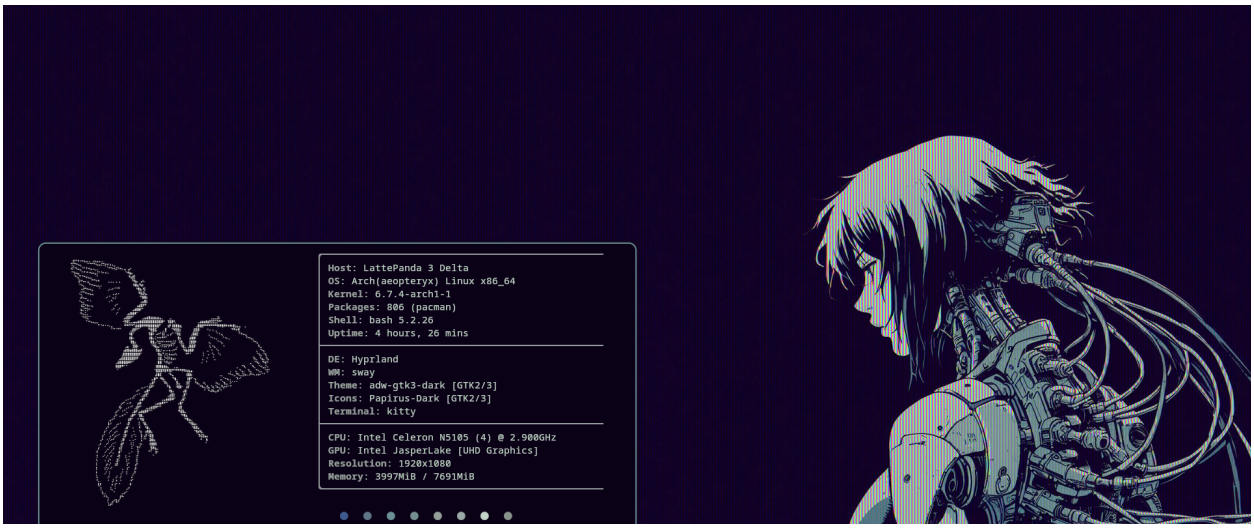


B22EE036_PRML_PA3_REPORT

AUTHOR : K. K. N. SHYAM SATHVIK

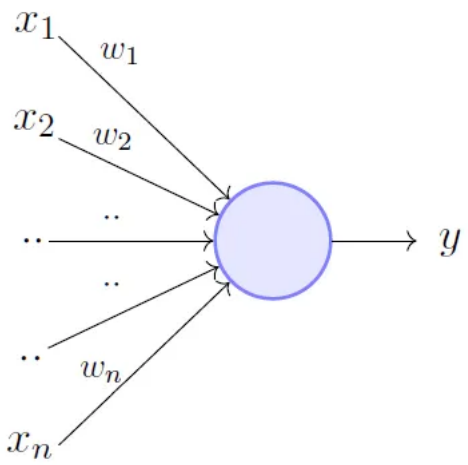
ROLL.NO : B22EE036

LINK TO THE COLAB FILE :

Google Colaboratory

https://colab.research.google.com/drive/1FbU3aQMtuohKM2IxTqsIDCdgmEfHz_kQ?usp=sharing

▼ PERCEPTRON



$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i \geq \theta$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i < \theta$$

Rewriting the above,

$$y = 1 \quad if \sum_{i=1}^{n} w_i * x_i - \theta \geq 0$$

$$= 0 \quad if \sum_{i=1}^{n} w_i * x_i - \theta < 0$$

Perceptron is a simplified computation model designed to learn to classify linearly seperable data into two categories. It takes the decision by scaling the inputs, summing them up and then passing them through an activation function to produce an output (0 or 1).

The mathematical model of a perceptron is given by the following equation.

$$f(x) = \begin{cases} 1 & \text{if } w \cdot x + b > 0 \\ 0 & \text{otherwise} \end{cases}$$

Here,

- `f(x)` represents the output of the Perceptron, given input vector `x`.
- `w` is the weight vector.
- `b` is the bias term.
- $w \cdot x$ denotes the dot product of `w` and `x`.

The learning rule for the perceptron model updates the weights based on whether the predicitons made for a randomly picked sample aligns with its original label.

Algorithm: Perceptron Learning Algorithm

$P \leftarrow inputs \quad with \quad label \quad 1;$
$N \leftarrow inputs \quad with \quad label \quad 0;$
Initialize $\mathbf{w}$ randomly;
while $!convergence$ **do**
  Pick random $\mathbf{x} \in P \cup N$;
  if $\mathbf{x} \in P \quad and \quad \mathbf{w}.\mathbf{x} < 0$ **then**
    $\mathbf{w} = \mathbf{w} + \mathbf{x}$;
  end
  if $\mathbf{x} \in N \quad and \quad \mathbf{w}.\mathbf{x} \geq 0$ **then**
    $\mathbf{w} = \mathbf{w} - \mathbf{x}$;
  end
end
//the algorithm converges when all the inputs are classified correctly

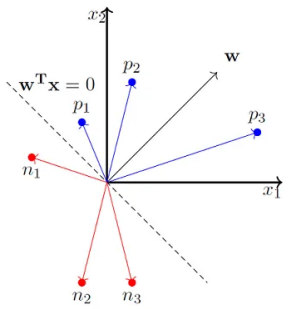
How it works:

- The key to understanding how a perceptron works is to understand the geometry behind the positive and negative classification.
- The dot product $w \cdot x$ tells us the angle a specific sample makes with the weights vector. A positive sample should have an angle less than or equal to 90 with the weights vector, whereas a negative vector align itself at an angle larger than 90 with w.

$$cos\alpha = \frac{\mathbf{w}^T\mathbf{x}}{||\mathbf{w}||||\mathbf{x}||} \quad \Big| \quad cos\alpha \propto \mathbf{w}^T\mathbf{x}$$

$$\text{So if } \mathbf{w}^T\mathbf{x} > 0 \quad \Rightarrow cos\alpha > 0 \quad \Rightarrow \alpha < 90$$
$$\text{Similarly, if } \mathbf{w}^T\mathbf{x} < 0 \quad \Rightarrow cos\alpha < 0 \quad \Rightarrow \alpha > 90$$

- The core of the Perceptron's learning algorithm adjusts w based on the classification accuracy for individual samples. If a positive instance is misclassified (i.e., it wrongly falls on the side of the decision boundary meant for negative instances), the algorithm modifies w to reduce the angle between w and the instance vector, thus pushing it towards the positive side. Conversely, for a misclassified negative instance, w is adjusted to increase the angle, moving the instance to the correct side of the boundary.



- In essence, through iterative updates, the Perceptron fine-tunes ww to align correctly with both positive and negative instances, facilitating accurate classification. This geometric interplay between w and instance vectors underpins the Perceptron's ability to learn and distinguish between the two classes efficiently.

(α_{new}) when $\mathbf{w_{new}} = \mathbf{w} + \mathbf{x}$	(α_{new}) when $\mathbf{w_{new}} = \mathbf{w} - \mathbf{x}$
$cos(\alpha_{new}) \propto \mathbf{w_{new}}^T\mathbf{x}$	$cos(\alpha_{new}) \propto \mathbf{w_{new}}^T\mathbf{x}$
$\propto (\mathbf{w}+\mathbf{x})^T\mathbf{x}$	$\propto (\mathbf{w}-\mathbf{x})^T\mathbf{x}$
$\propto \mathbf{w}^T\mathbf{x} + \mathbf{x}^T\mathbf{x}$	$\propto \mathbf{w}^T\mathbf{x} - \mathbf{x}^T\mathbf{x}$
$\propto cos\alpha + \mathbf{x}^T\mathbf{x}$	$\propto cos\alpha - \mathbf{x}^T\mathbf{x}$
$cos(\alpha_{new}) > cos\alpha$	$cos(\alpha_{new}) < cos\alpha$

- The perceptron learning algorithm has been proven to converge for linearly seperable dataset, i.e all samples will be correctly classified given that the model is trained for long enough and with a suitable learning rate.

▼ Task 0:

- Generate a synthetic 4-dimensional dataset:
  (a) Initial a linear function: f(x) = w0 + w1x1 + w2x2 + w3x3 + w4x4.
  (b) Choose w0, w1, w2, w3,and w4 randomly.

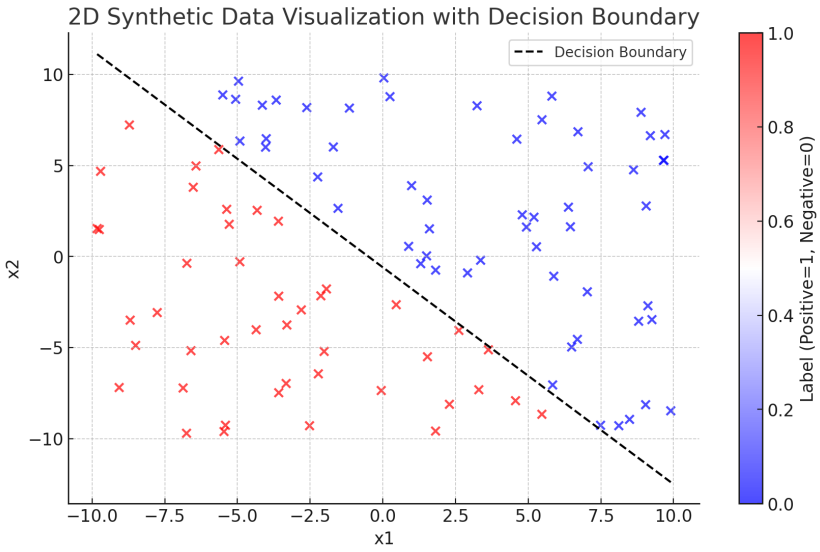
  (c) Evaluate f(x). If the result is greater than or equal to zero, then label it as a positive example, otherwise label it as a negative example.

- **Deliverables:**
  - Function Generated : $f(x) = 0.99 + 0.22x_1 + -0.25x_2 + -0.72x_3 + -0.83x_4$
  - Data Generated: (1st 5 rows.)

```
9.890019474506936,3.073119921307434,-9.091117917287,-1.7811909743531835,1
1.2769246382383415,-0.1634678295041514,-4.498895755688334,-1.2543828905765402,1
-1.2923076304006642,-4.74394924782805,-4.296842104743348,-8.346283974283,1
-4.568943731042305,0.15231333747208708,-8.940753787798556,-4.563170770133023,1
1.2619127197111073,2.66490199465923,0.4285979434147684,-1.294480514847237,1
-5.717766798822548,-4.832828165493163,4.073500473102298,5.229853681923837,0
```

- **The purpose of this task is to generate a linearly seperable dataset. Hence, any perceptron trained on a linearly seperable dataset generated by an arbitray equation such as this which labels the features as 0 and 1 depending on the sign of the dot product of w and x will give a 100% accuracy.**

- Following the exact approach on a 2d synthetic dataset generates a linearly seperable dataset such as shown below.



- Another approach to generate linearly separable data could be to use `np.random.randint` and then center half of the dataset around some random point by adding it to each point and centering the other half by subtracting the same random point from the whole dataset.

▼ Task 1 & 2:

- Write training code for the perceptron learning algorithm. (Do not forget to normalize the data both during training and testing). (Deliverable: train.py) (10 points)

- The following class defines a perceptron model, the model needs to be given data, but there's also a helper function which can do it for you, it needs to be called like-so : `perceptron.normalize_data()`

- **Perceptron (My implementation)**

```
import argparse
import numpy as np

class Perceptron:
    def __init__(self, X_train, y, weights=4, learning_rate=1, iterations=1000):
        self.X_train = self.normalize(X_train)       # normalizing step.
        self.y = y
```

```
        self.lr = learning_rate
        self.iter = iterations
        self.weights = np.random.randn(weights + 1)  # Including the bias term w0
        self.X_train_with_intercept = np.hstack((np.ones((self.X_train.shape[0], 1)), self.X_t

    def train(self):
        print("Weights : ", self.weights)
        print("Train :", self.X_train_with_intercept)
        iters = 0
        while not self.convergence() and iters < self.iter:
            print(self.convergence())
            iters += 1
            for i, (x, label) in enumerate(zip(self.X_train_with_intercept, self.y)):
                y_pred = self.predict(self.weights, x)
                if label == 0 and y_pred != 0:
                    self.weights = self.weights - self.lr * x
                elif label == 1 and y_pred != 1:
                    self.weights = self.weights + self.lr * x
        print(f"Iters: {iters}, Accuracy: {self.accuracy()}")

    def convergence(self):
        if self.accuracy() == 1.0:
          return True
        return False

    def accuracy(self):
        correct_predictions = 0
        for x, label in zip(self.X_train_with_intercept, self.y):
            y_pred = self.predict(self.weights, x)
            if y_pred == label:
                correct_predictions += 1
        return correct_predictions / len(self.y)

    def predict(self, w, feature):
        # print("w", w)
        # print("feeeee", feature)
        # print(np.dot(w, feature))
        t = np.dot(w, feature)
        return 1 if t >= 0 else 0

    def normalize(self, X):
        mean = X.mean(axis=0)
        std = X.std(axis=0)
        return (X - mean) / std

    def save_weights(self):
        weights_str = '\n'.join(map(str, self.weights))
        with open('weights.txt', 'w') as file:
            file.write(weights_str)
```

The above class includes the following functionality:

- `def __init__` - Instantiates the perceptron object with `X_train, y` : features, labels. Also takes care of normalizing the input data.
- `def train` : Trains the perceptron to fit the instantiated data.
- `def accuracy` : Computes the accuracy of the model at the current timestamp with respect to the labels y.
- `def predict` : Classifies one label.
- `def convergence` : Checks if all the samples are correctly classified.
- `def save_weights` : Saves the weights after convergence in `weights.txt`

Additional Functionality written in `train.py` and `test.py` :

- `def load_data` : Use to load data from the file passed to `train.py` in the command line.
- `argparse` : used argparse for taking cli inputs, not really sure why this is special but I used it anyways.

▼ Task 3:

- First, I split the X_train, y that I randomly generated using the following:

```
np.random.seed(42)

# Generate a linearly separable dataset
X_train_pos = np.random.randint(1, 10, (100, 4)) + 10 # 100 samples for class 1, slightly shi
X_train_neg = np.random.randint(1, 10, (100, 4)) - 10 # 100 samples for class 0, slightly shi
X_train = np.vstack((X_train_pos, X_train_neg))
y = np.array([1]*100 + [0]*100)                       # First 100 are class 1, next 100 are
```

- Second, I split the data into train and test with a 9:1 ratio. I'll keep this test data constant to test it on the perceptron after training it with different subsets of the data.

```
# Initial split into training (90%) and test data (10%)
X_train_90, y_train_90, X_test_10, y_test_10 = train_test_split(X_train, y, 0.1)
```

- Third, I split the X_train_90, y_train_90 into subsets of 20%, 70% and 50%, like so:

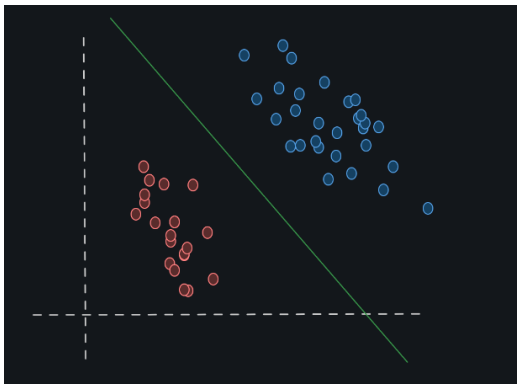
```
# Training subsets percentages of the 90% training data
training_percentages = [0.2, 0.5, 0.7]
test_results = []

for percentage in training_percentages:
```

```
    X_train_subset, y_train_subset = split_data(X_train_90, y_train_90, percentage)
    perceptron_new = Perceptron(X_train_subset, y_train_subset)
    perceptron_new.train()
    test_accuracy = perceptron_new.test(X_test_10, y_test_10)
    test_results.append({'Training Percentage': percentage*100, 'Test Accuracy': test_accuracy
```

RESULTS:

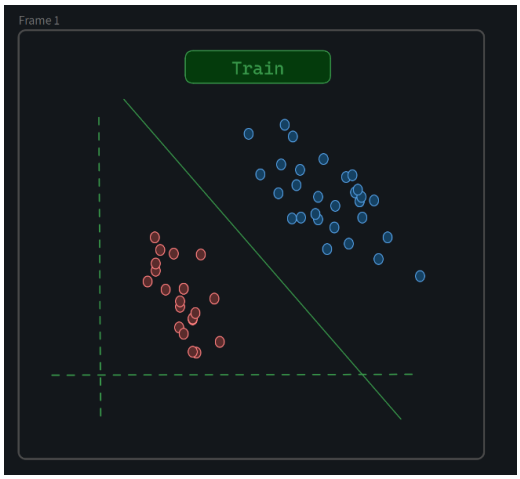
Training Percentage	Training Accuracy	Testing Accuracy
20%	100%	100%
50%	100%	100%
70%	100%	100%

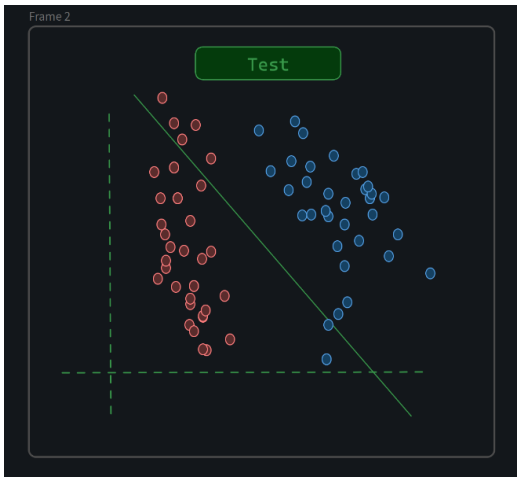
- The above results make sense since the decision boundary learnt is simple. From my understanding the training-testing process was something of this sort making the decision boundary really simple to learn.

- Hence, resulting in a 100% accuracy even after training with very less data (20%).
    - 

- In reality, even if the perceptron separates the data and finds a decision boundary that perfectly classifies every single data point, it won't necessarily translate to 100% accuracy on the test data since we don't know if the data combination of test+train is still linearly separable or not. Even if they were linearly separable, we still wouldn't know if the perceptron perfectly classifies all data points, since the decision boundary learnt was learnt on the train distribution and not on the test.

- Consider the following data for example, where a perceptron fails in test.
    -





▼ EIGENFACES

▼ Principal Component Analysis

- Principal Component Analysis is a Dimensionality Reduction technique which lets us transform a higher dimensional dataset into a lower dimensional one while keeping most of the data’s essence intact.
- It can tell us which of the features can have a higher discriminatory nature with respect to the other ones, thus letting us let go of the lower importance features while still capturing the essence

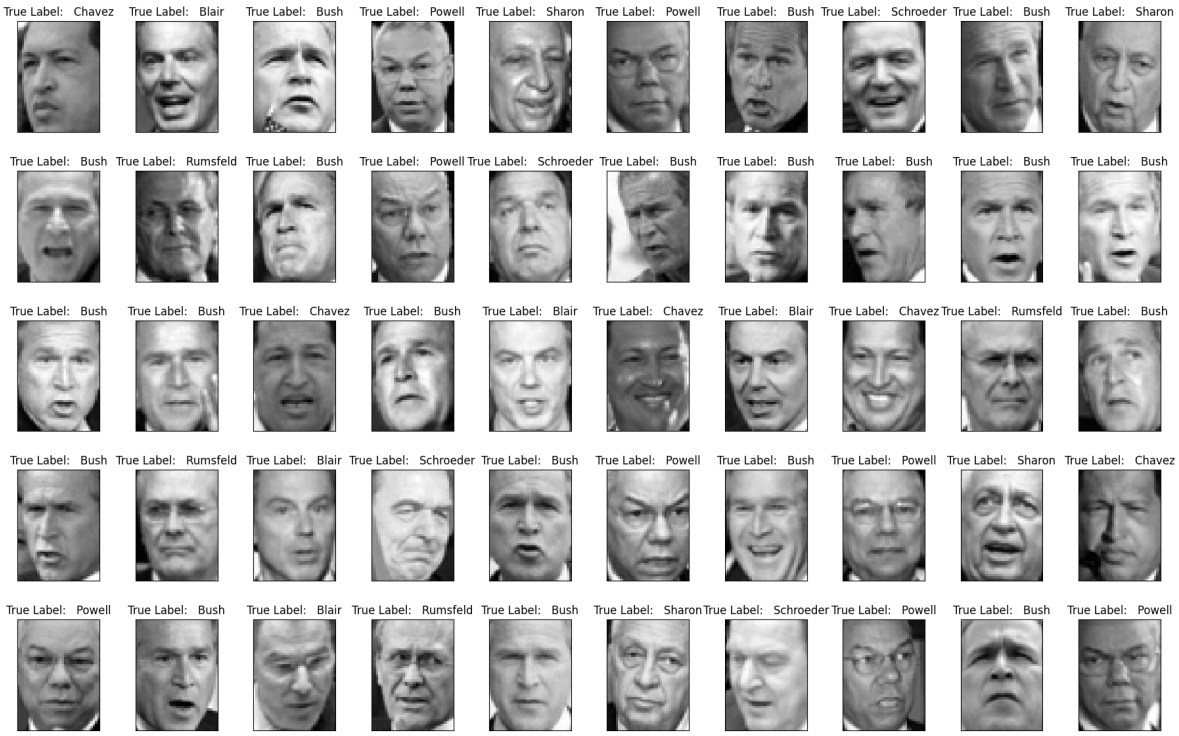
▼ Task 1: Data Preprocessing

- (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.

▼ Dataset / DataLoader : [Ref : `fetch_lfw_people.DESCR`]

Dataset : The Labeled Faces in the Wild face recognition dataset



This dataset is a collection of JPEG pictures of famous people collected over the internet, all details are available on the official website: http://vis-www.cs.umass.edu/lfw/ The dataset contains more than 13,000 images of faces. Each picture is centered on a single face.

The typical task performed on this dataset is Face Verification: given a pair of two pictures, a binary classifier must predict whether the two images are from the same person.

An alternative task which can be performed, Face Recognition or Face Identification is: given the picture of the face of an unknown person, identify the name of the person by referring to a gallery of previously seen pictures of identified persons.

Both Face Verification and Face Recognition are tasks that are typically performed on the output of a model trained to perform Face Detection. The most popular model for Face Detection is called Viola-Jones and is implemented in the OpenCV library. The LFW faces were extracted by this face detector from various online websites.

(a) Load the LFW dataset using the Scikit-learn's fetch lfw people function.

```
[3]  lfw_people = fetch_lfw_people(min_faces_per_person=70, resize=0.4)

     # Getting the dimensions of the images to plot them
     n_samples, h, w = lfw_people.images.shape

[5]  # print(lfw_people)
     print(dir(lfw_people))

     ['DESCR', 'data', 'images', 'target', 'target_names']
```

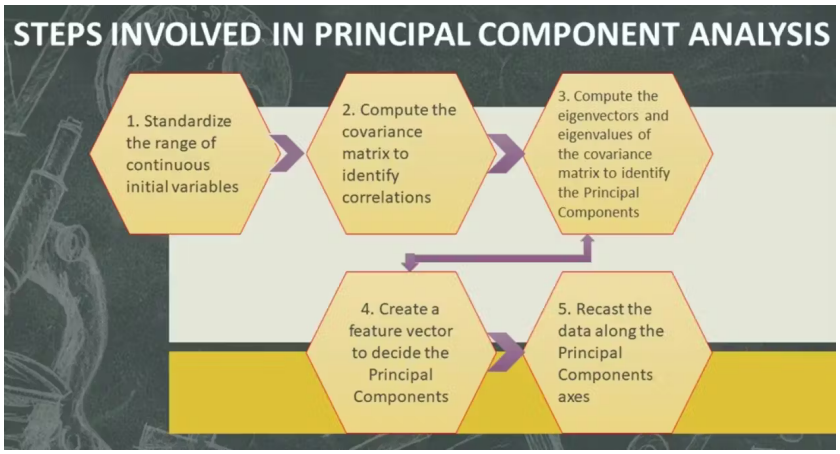
- Note : `min_faces_per_person` and `resize` have been given the same parameters as the sci-kit learn article since most of the work has been done before exploring the dataset more.

The dataset returned by `fetch_lfw_people` is a dictionary-like object with the following attributes:

- `data`: NumPy array of shape `(n_samples, n_features)`. Each row corresponds to a flattened face image (2D slices) of original size 62x47 pixels. When the `resize` parameter is used, the image size changes accordingly, and the data array shape changes too.
- `images`: NumPy array of shape `(1288, 50, 37)` containing the face images themselves. The dimensions are reported after performing the resizing operation.
- `target`: NumPy array of shape `(1288,)` with labels for each image. Each label is an integer index of the identity of the person in the image.
- `target_names`: NumPy array of shape `(7,)` containing the names of the people in the dataset. Here we only take the images of the people who's face appears at least 70 times in the dataset each.
- `DESCR`: A text description of the dataset.

💡 Most of the tasks performed in the task were directly followed from the reference sci-kit learn article provided here : https://scikit-learn.org/stable/auto_examples/applications/plot_face_recognition.html

▼ PCA Implementation



Source: https://www.turing.com/kb/guide-to-principal-component-analysis

For this task I've chosen to implement my own version of PCA using the algorithm provided by the Professor in Lecture 16. I've compared the results of my own PCA implementation and the PCA from sci-kit learn library.

▼ Implementation of Principal Component Analysis:

- **Step 1 :**
  - Given a data set of m samples and n features:

$$X = \{x_1, x_2, x_3, x_4, ..., x_n\}$$

where each x_i is a feature vector of n dimensions.

$$X = \begin{bmatrix} x_{11} & x_{12} & \cdots & x_{1n} \\ x_{21} & x_{22} & \cdots & x_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ x_{m1} & x_{m2} & \cdots & x_{mn} \end{bmatrix}$$

- **Step 2 : Data Normalization:**
  - Center the data about the origin by normalizing it about its mean.
  - Subtract each sample with the mean and dividing with the standard deviation we have the following result:
    - Standard Normalization:
      - $z = \frac{(x-\mu)}{\sigma}$

- **Step 3: Computing the Covariance Matrix:**
  - Calculate the covariance matrix to understand how each variable relates to the other. The covariance matrix is given by:

  - $\sigma_{ij} = \frac{1}{m-1} \sum_{k=1}^{m} (x_{ik} - \mu_i)(x_{jk} - \mu_j)$

- The covariance matrix is then given by:

$$\Sigma = \begin{bmatrix} \sigma_{11} & \sigma_{12} & \cdots & \sigma_{1n} \\ \sigma_{21} & \sigma_{22} & \cdots & \sigma_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ \sigma_{n1} & \sigma_{n2} & \cdots & \sigma_{nn} \end{bmatrix}$$

- This step is crucial for PCA because the eigenvectors of the covariance matrix represent the directions of maximum variance in the data, and the eigenvalues represent the magnitude of these directions.

- **Step 3 : Computing the Eigenvalues and the Eigenvectors.**
  - Computing the eigenvalues and eigenvectors of the covariance matrix calculated in the previous step to identify the principal components.
  - The eigenvectors of the covariance matrix are orthogonal and represent the directions of maximum variance, known as principal components. The eigenvalues indicate the amount of variance captured by each principal component.
  - Reference: https://numpy.org/doc/stable/reference/generated/numpy.linalg.eig.html

- **Step 4: Sorting the Eigenvectors and Eigenvalues:**
  - After computing the eigenvalues and eigenvectors of the covariance matrix, sort them by decreasing eigenvalues. This step ensures that the eigenvectors are ordered by their importance in explaining the variance within the dataset.
  - Sorting is crucial because it helps in **selecting the top k eigenvectors based on the sorted eigenvalues, which contribute most to the data's variance**. This selection forms the basis for dimensionality reduction in PCA.

- **Step 5: Calculating the Reduced Dimensionality Dataset X_r (X-reduced):**
  - Select the top k eigenvectors forming a matrix W of dimensions $n \times k$, where $k < n$, based on the sorted eigenvalues. This matrix Wrepresents the transformation matrix to map the data from the original space to the reduced space.
  - The reduced dimensionality dataset, X_r, is obtained by projecting the original data X onto the space spanned by the selected eigenvectors. This projection is achieved through matrix multiplication:

$$X_r = X * W$$

  - X_r now represents the dataset in the reduced dimensionality, retaining most of the original dataset's variance.

$$X_{\text{reduced}} = \begin{bmatrix} x'_{11} & x'_{12} & \cdots & x'_{1k} \\ x'_{21} & x'_{22} & \cdots & x'_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ x'_{m1} & x'_{m2} & \cdots & x'_{mk} \end{bmatrix}$$

    - The final matrix is a transformed dataset with less dimensions than the original which still captures most of the underlying patterns necessary for the learning task at hand.

▼ Task 2 and Parts of Task 5: EigenFaces Implementation

- (a) Implement Eigenfaces using Principal Component Analysis (PCA). Set an appropriate value for ncomponents and explain your choice in the report (refer explained variance ratio attribute for PCA in the Scikit-learn documentation to justify your choice).

The Key-Points to be noted from the Eigenfaces paper are:

- **Objective**: To recognize faces using a lower-dimensional representation of face images.
- **Principle**: Faces are represented in a high-dimensional space. Eigenfaces reduce this dimensionality by capturing the most significant variations among face images.
- **Process**:
    - Collect and standardize a set of face images.
    - Use Principal Component Analysis (PCA) to find the eigenfaces, which are the principal components of the face dataset.
    - Represent each face as a combination of these eigenfaces.
- **Recognition**:
    - Project a new face onto the eigenface space and compare its representation with known faces to recognize it.
- **Advantages**: Reduces computational complexity and is robust to certain variations in faces.
- **Limitations**: Sensitivity to alignment, lighting, and pose variations.
- **Applications**: Used in face recognition and related fields in digital image processing.

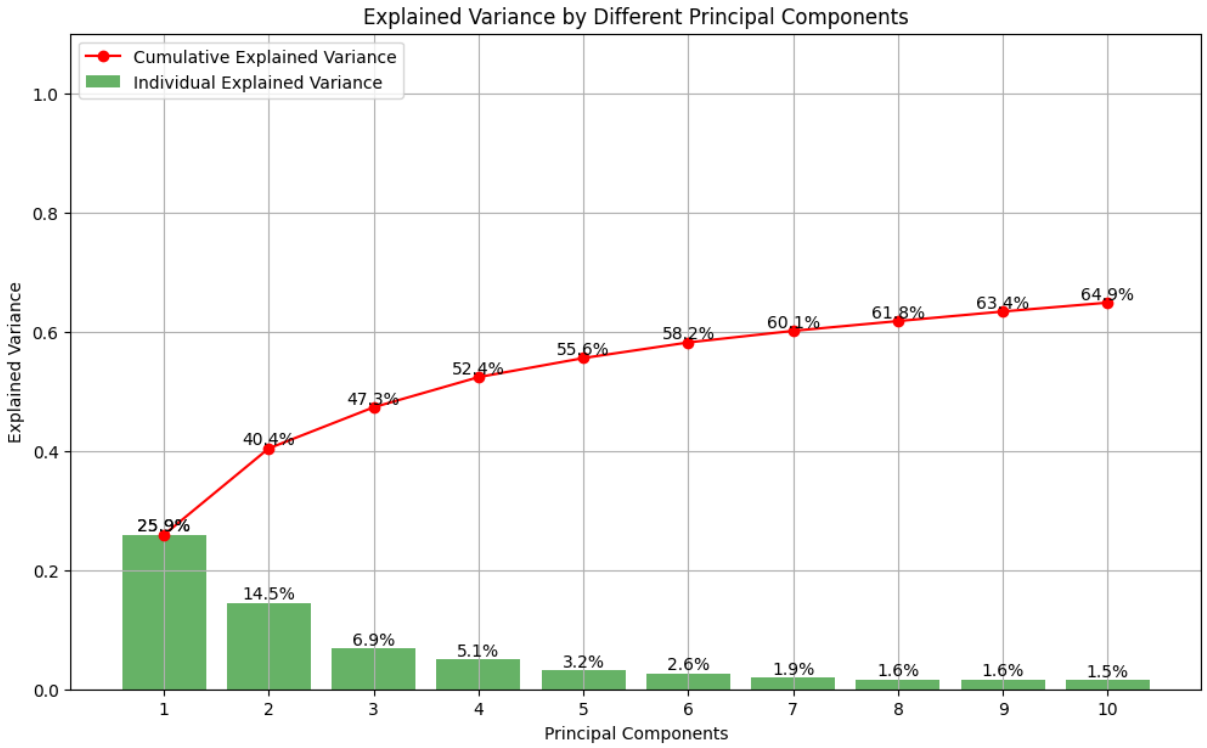
Explained-Variance Ratio:

- The explained variance ratio in Scikit-learn, accessed via the `explained_variance_ratio_` attribute in PCA, indicates the proportion of the dataset's total variance that each principal component accounts for. It's calculated by dividing the variance captured by each principal component by the total variance of the dataset.
- Eigenvalues indicate the variance along each component.

> The Cumulative Explained Variance plot is a graphical representation that shows the proportion of the dataset’s variance that is cumulatively explained by each component. When you perform PCA, you’re transforming the

> data into a new coordinate system with axes ranked by how well they capture the variance in the data. Each axis (principal component) can explain a certain amount of the variance.

Reference: https://medium.com/@megha.natarajan/understanding-cumulative-explained-variance-in-pca-with-python-653e3592a77c



Indicates the decreased contribution to from the principal components to the explained variance (which cannot directly be discarded lest we lose the discriminatory features, ideally we'd pick the number of components which explain at least 95% of the variance for a competent model. PCA works better when there's a huge amount of data, since the discriminatory features are represented well enough to not be discarded by PCA, and the discarded data mostly comes from the outliers. Source: My intuition.)

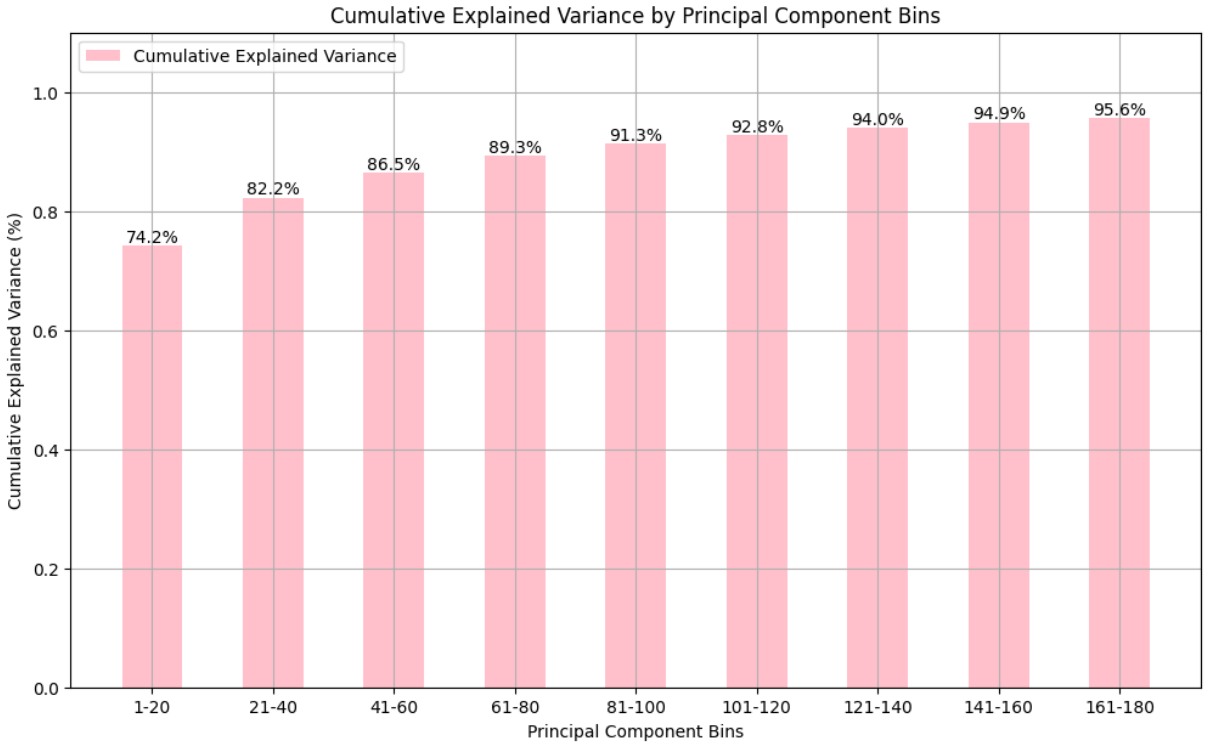
The above given plot show the variance explained by each individual component and from the above graph it can be clearly inferred that most of the data can be explained or represented with the first few principal components itself, but to have a model with a good performance, we would want to ideally at least capture 90% of the explained variance of the data, since that is where the **discriminatory features** are. Picking only the first 10 components since they're explaining abut 70% of the data would lead to a poor performance.

`Method used to pick the n_components parameter:`

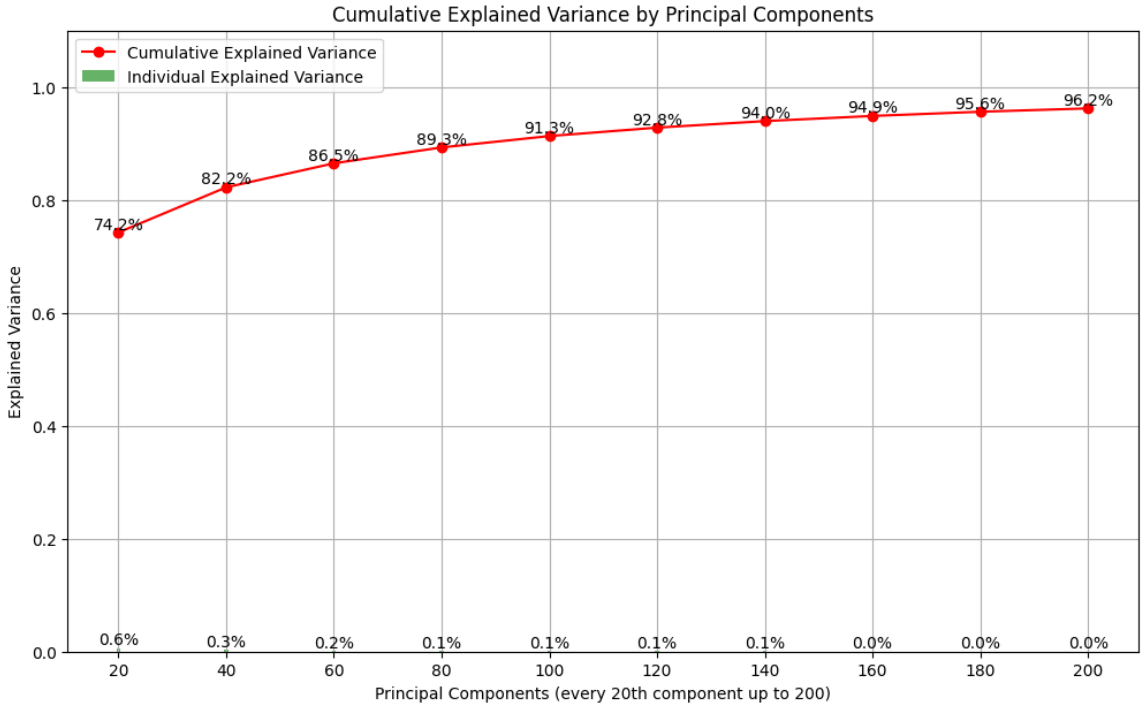
```
pca = PCA().fit(X_train)
# Calculate cumulative explained variance
cumulative_variances = np.cumsum(pca.explained_variance_ratio_)
# Finding the number of components that explain at least 95% of the variance
n_components = np.where(cumulative_variances >= 0.95)[0][0] + 1
print(f'Number of components chosen: {n_components}')
```

```
Number of components chosen: 163
```

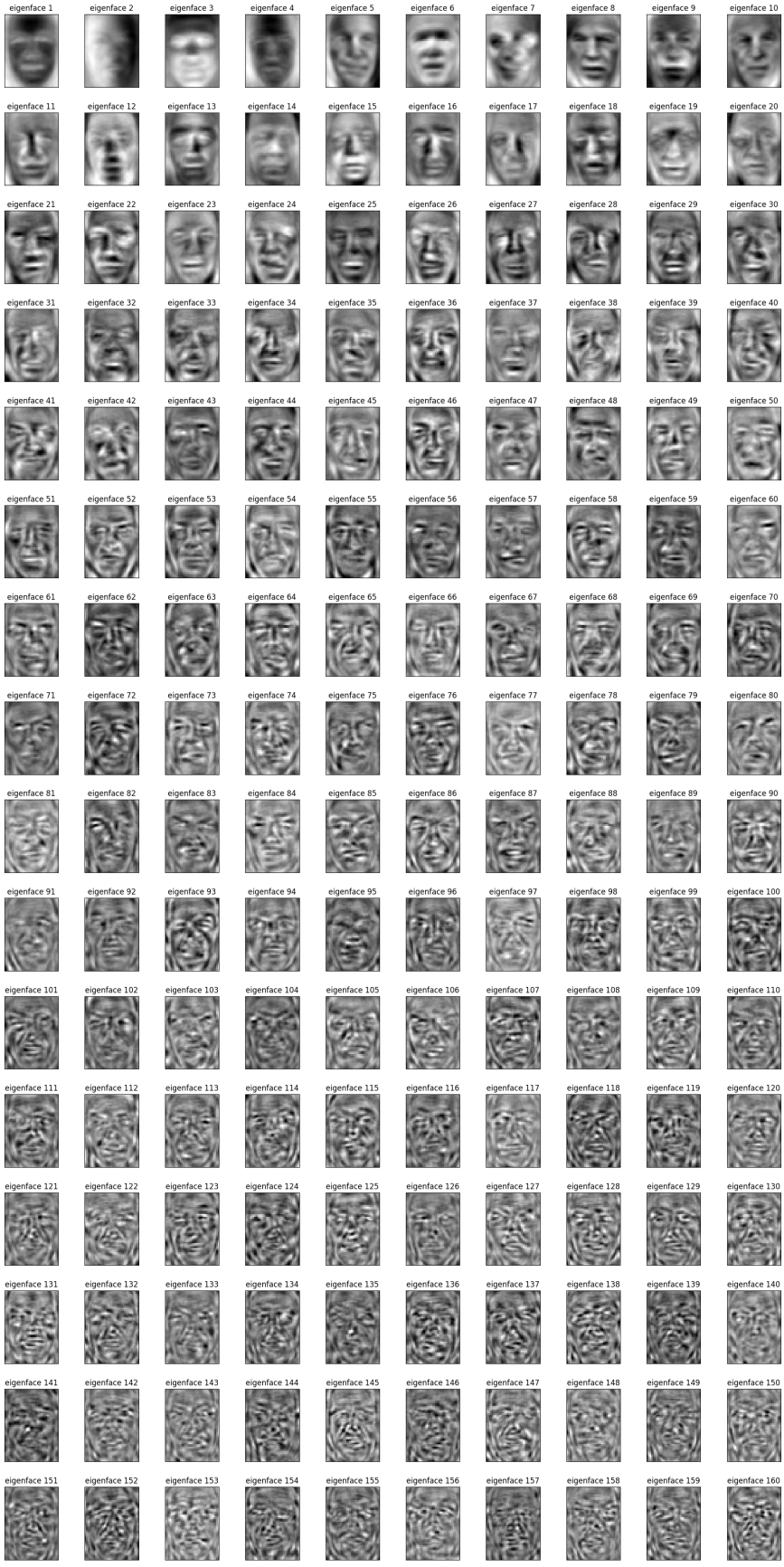
The n_components parameter was picked such that the transformed data represents at least 95% of the data, that is, such that the components explain at least 95% of the variance of the data.



A plateau can be observed around the 150s for the explained variance.



- **EigenFaces : These are the principal components (or eigenvectors) of the covariance matrix of the face dataset, which can be visualized as images. Each eigenface represents a basis feature of the faces in the dataset.**
- These images form the template (basis vectors) over which other images are reconstructed/transformed to the space of.


eigenface 1
eigenface 2
eigenface 3
eigenface 4
eigenface 5
eigenface 6
eigenface 7
eigenface 8
eigenface 9
eigenface 10
eigenface 11
eigenface 12
eigenface 13
eigenface 14
eigenface 15
eigenface 16
eigenface 17
eigenface 18
eigenface 19
eigenface 20
eigenface 21
eigenface 22
eigenface 23
eigenface 24
eigenface 25
eigenface 26
eigenface 27
eigenface 28
eigenface 29
eigenface 30
eigenface 31
eigenface 32
eigenface 33
eigenface 34
eigenface 35
eigenface 36
eigenface 37
eigenface 38
eigenface 39
eigenface 40
eigenface 41
eigenface 42
eigenface 43
eigenface 44
eigenface 45
eigenface 46
eigenface 47
eigenface 48
eigenface 49
eigenface 50
eigenface 51
eigenface 52
eigenface 53
eigenface 54
eigenface 55
eigenface 56
eigenface 57
eigenface 58
eigenface 59
eigenface 60
eigenface 61
eigenface 62
eigenface 63
eigenface 64
eigenface 65
eigenface 66
eigenface 67
eigenface 68
eigenface 69
eigenface 70
eigenface 71
eigenface 72
eigenface 73
eigenface 74
eigenface 75
eigenface 76
eigenface 77
eigenface 78
eigenface 79
eigenface 80
eigenface 81
eigenface 82
eigenface 83
eigenface 84
eigenface 85
eigenface 86
eigenface 87
eigenface 88
eigenface 89
eigenface 90
eigenface 91
eigenface 92
eigenface 93
eigenface 94
eigenface 95
eigenface 96
eigenface 97
eigenface 98
eigenface 99
eigenface 100
eigenface 101
eigenface 102
eigenface 103
eigenface 104
eigenface 105
eigenface 106
eigenface 107
eigenface 108
eigenface 109
eigenface 110
eigenface 111
eigenface 112
eigenface 113
eigenface 114
eigenface 115
eigenface 116
eigenface 117
eigenface 118
eigenface 119
eigenface 120
eigenface 121
eigenface 122
eigenface 123
eigenface 124
eigenface 125
eigenface 126
eigenface 127
eigenface 128
eigenface 129
eigenface 130
eigenface 131
eigenface 132
eigenface 133
eigenface 134
eigenface 135
eigenface 136
eigenface 137
eigenface 138
eigenface 139
eigenface 140
eigenface 141
eigenface 142
eigenface 143
eigenface 144
eigenface 145
eigenface 146
eigenface 147
eigenface 148
eigenface 149
eigenface 150
eigenface 151
eigenface 152
eigenface 153
eigenface 154
eigenface 155
eigenface 156
eigenface 157
eigenface 158
eigenface 159
eigenface 160

- On projecting the input data on the eigenfaces orthonormal basis, we get a transformed dataset, can't show it in an image format since the feature vectors get transformed to a `163*1` vector and 163 cannot be reshaped into a image format, since 163 is a prime.

▼ Task 3 & 4: Model Training and Model Evaluation

3.(a) Choose a classifier for Eigenfaces (e.g., K-Nearest Neighbors) and train the classifier using the transformed trainingdata. (10 points)

The task at hand is a multi-class classification task, and the following models were trained on the transformed dataset:

```
models = [
    ('KNN', KNeighborsClassifier(n_neighbors=5)),
    ('Logistic Regression', LogisticRegression()),
    ('SVM', SVC()),
    ('Decision Tree', DecisionTreeClassifier()),
    ('Random Forest', RandomForestClassifier()),
    ('Gradient Boosting', GradientBoostingClassifier())
]
```

The above listed models were trained on the data and were tested based on the following metrics. The results are tabulated below.

4.(a) Use the trained Eigenfaces classifier to make predictions on the Eigenfaces-transformed testing data.

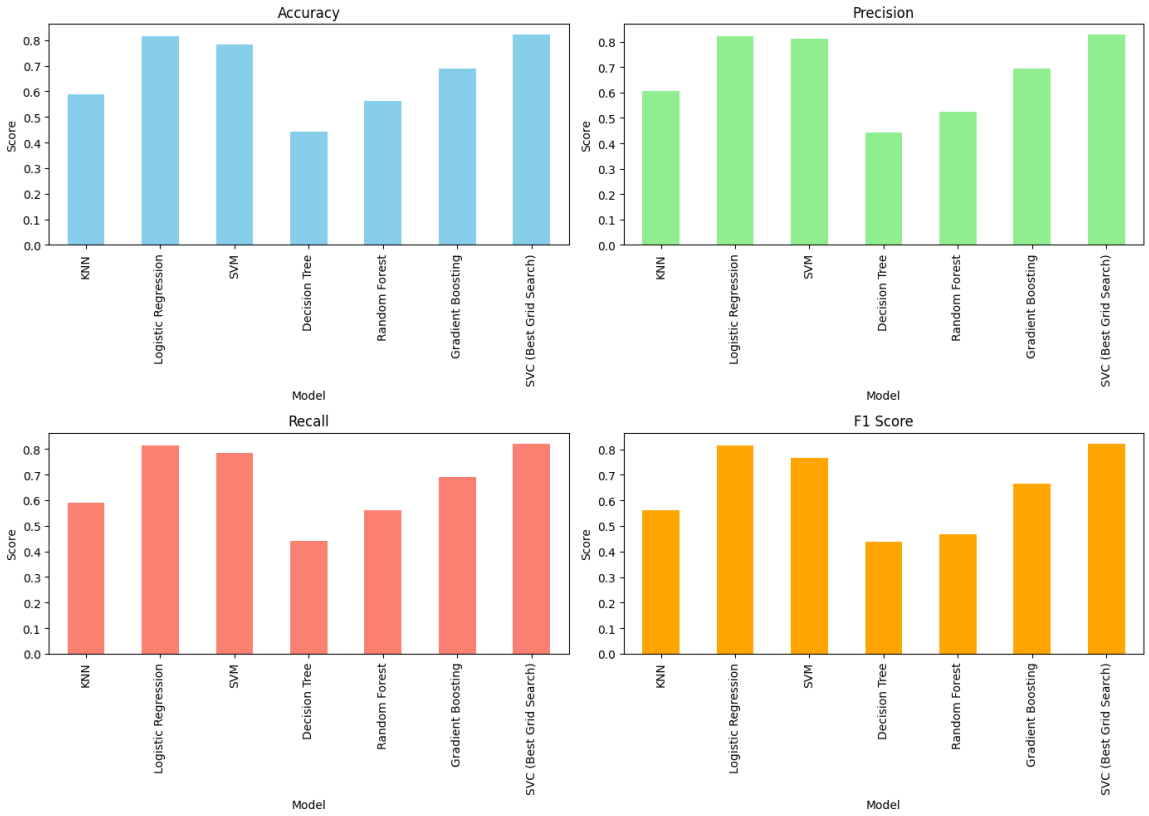
4.(b) Calculate and report accuracy.

▼ COMPARISION TABLE

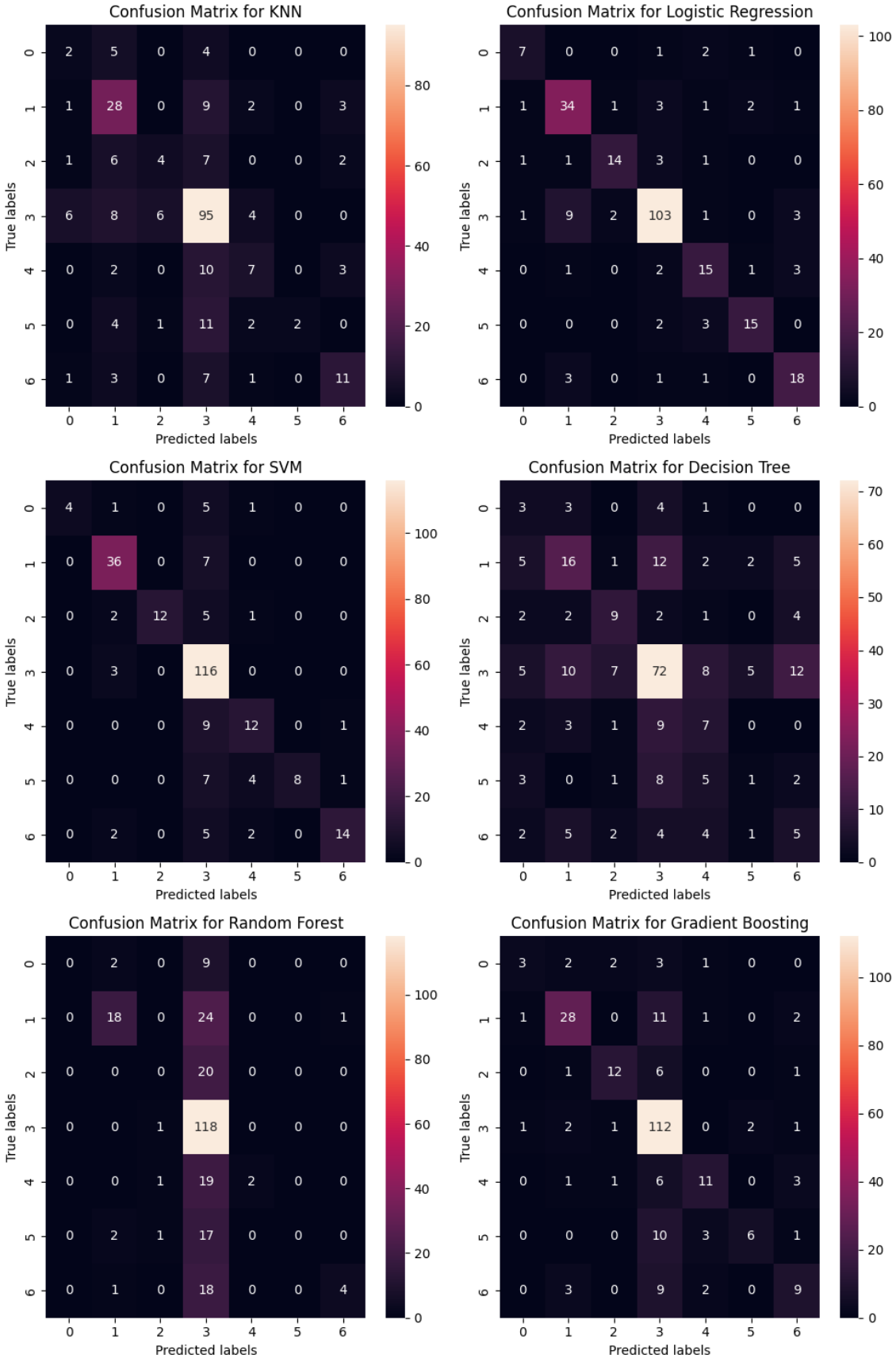
Model	Accuracy	Precision	Recall	F1 Score
KNN	0.581395	0.596736	0.581395	0.552701
Logistic Regression	0.833333	0.840738	0.833333	0.834726
SVM	0.786822	0.812903	0.786822	0.772183
Decision Tree	0.414729	0.427630	0.414729	0.418676
Random Forest	0.562016	0.63179	0.562016	0.464934
Gradient Boosting	0.720930	0.719402	0.720930	0.700341
SVC (Best Grid Search)	0.821705	0.827955	0.821705	0.823187

💡 Please note that the performance metrics in collab have changed on rerun, but they’re more or less around the same value.

▼ ACCURACY PLOTS



Performance Comparision across all models.


Confusion Matrix for KNN
2 5 0 4 0 0 0
1 28 0 9 2 0 3
1 6 4 7 0 0 2
6 8 6 95 4 0 0
0 2 0 10 7 0 3
0 4 1 11 2 2 0
1 3 0 7 1 0 11
True labels
Predicted labels
0 1 2 3 4 5 6
0 20 40 60 80
Confusion Matrix for Logistic Regression
7 0 0 1 2 1 0
1 34 1 3 1 2 1
1 1 14 3 1 0 0
1 9 2 103 1 0 3
0 1 0 2 15 1 3
0 0 0 2 3 15 0
0 3 0 1 1 0 18
True labels
Predicted labels
0 1 2 3 4 5 6
0 20 40 60 80 100
Confusion Matrix for SVM
4 1 0 5 1 0 0
0 36 0 7 0 0 0
0 2 12 5 1 0 0
0 3 0 116 0 0 0
0 0 0 9 12 0 1
0 0 0 7 4 8 1
0 2 0 5 2 0 14
True labels
Predicted labels
0 1 2 3 4 5 6
0 20 40 60 80 100
Confusion Matrix for Decision Tree
3 3 0 4 1 0 0
5 16 1 12 2 2 5
2 2 9 2 1 0 4
5 10 7 72 8 5 12
2 3 1 9 7 0 0
3 0 1 8 5 1 2
2 5 2 4 4 1 5
True labels
Predicted labels
0 1 2 3 4 5 6
0 10 20 30 40 50 60 70
Confusion Matrix for Random Forest
0 2 0 9 0 0 0
0 18 0 24 0 0 1
0 0 0 20 0 0 0
0 0 1 118 0 0 0
0 0 1 19 2 0 0
0 2 1 17 0 0 0
0 1 0 18 0 0 4
True labels
Predicted labels
0 1 2 3 4 5 6
0 20 40 60 80 100
Confusion Matrix for Gradient Boosting
3 2 2 3 1 0 0
1 28 0 11 1 0 2
0 1 12 6 0 0 1
1 2 1 112 0 2 1
0 1 1 6 11 0 3
0 0 0 10 3 6 1
0 3 0 9 2 0 9
True labels
Predicted labels
0 1 2 3 4 5 6
0 20 40 60 80 100

▼ INDIVIDUAL RESULTS

Parameters the results were evaluated on:

- **Precision:** The ratio of true positives in all positive predictions for a specific class, indicating the model's accuracy in identifying that class.
- **Recall (Sensitivity):** The ratio of true positive predictions in all actual instances of a specific class, reflecting the model's ability to remember all relevant cases.
- **F1-Score:** Harmonic mean of recall and precision, providing a balanced measure of the model's performance in terms of both accuracy and completeness.
- **Support:** The number of actual occurrences of a specific class in the dataset, which indicates the class's representation and importance in evaluating model performance.
- **Macro Average:** Treats each class equally, averages the performance metrics (like precision, recall, or f1-score) across all classes.
- **Weighted Average:** Weights classes based on their number of instances, averaging performance metrics based on class size.
- **Accuracy:** The proportion of all correct predictions made by the model out of all predictions.

1. KNN `(n=5)` :

```
KNN Model
Accuracy: 0.5813953488372093

Classification Report:
              precision    recall  f1-score   support

           0       0.25      0.27      0.26        11
           1       0.50      0.63      0.56        43
           2       0.40      0.20      0.27        20
           3       0.67      0.81      0.73       119
           4       0.41      0.32      0.36        22
           5       1.00      0.10      0.18        20
           6       0.58      0.48      0.52        23

    accuracy                           0.58       258
   macro avg       0.54      0.40      0.41       258
weighted avg       0.60      0.58      0.55       258
```

2. Logistic Regression :

```
Logistic Regression Model
Accuracy: 0.8333333333333334

Classification Report:
              precision    recall  f1-score   support

           0       0.88      0.64      0.74        11
           1       0.76      0.86      0.80        43
           2       0.88      0.75      0.81        20
           3       0.92      0.89      0.91       119
           4       0.64      0.73      0.68        22
           5       0.84      0.80      0.82        20
           6       0.72      0.78      0.75        23

    accuracy                           0.83       258
   macro avg       0.81      0.78      0.79       258
weighted avg       0.84      0.83      0.83       258
```

3. SVM :

```
SVM Model
Accuracy: 0.7868217054263565

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.36      0.53        11
           1       0.82      0.84      0.83        43
           2       1.00      0.65      0.79        20
           3       0.76      0.97      0.85       119
           4       0.60      0.55      0.57        22
           5       1.00      0.40      0.57        20
           6       0.88      0.61      0.72        23

    accuracy                           0.79       258
   macro avg       0.86      0.63      0.69       258
weighted avg       0.81      0.79      0.77       258
```

4. Decision Tree :

```
Decision Tree Model
Accuracy: 0.41472868217054265

Classification Report:
              precision    recall  f1-score   support

           0       0.15      0.27      0.19        11
```

```
           1       0.33      0.33      0.33        43
           2       0.23      0.30      0.26        20
           3       0.64      0.57      0.60       119
           4       0.38      0.45      0.42        22
           5       0.00      0.00      0.00        20
           6       0.21      0.26      0.24        23

    accuracy                           0.41       258
   macro avg       0.28      0.31      0.29       258
weighted avg       0.43      0.41      0.42       258
```

5. Random Forest :

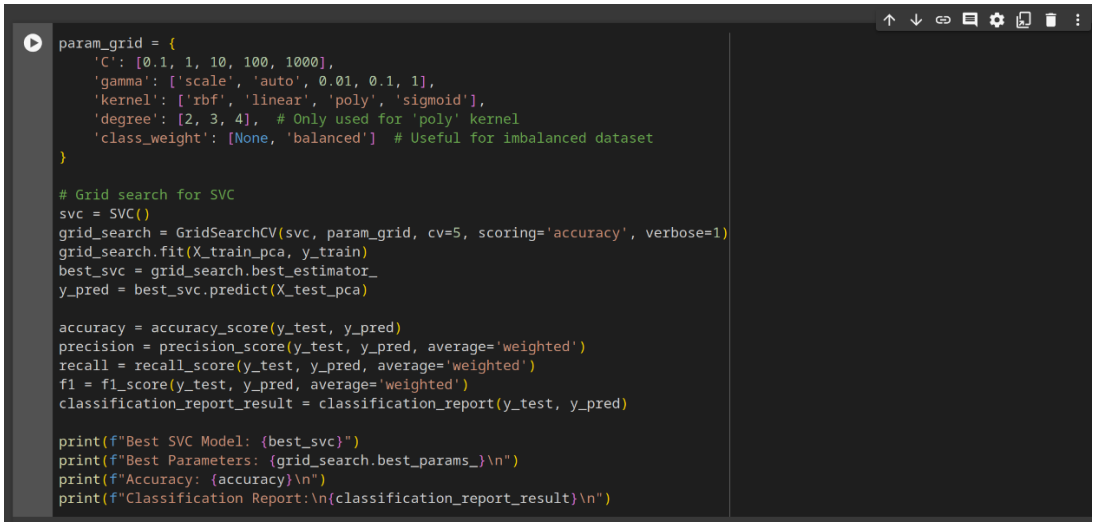
```
Random Forest Model
Accuracy: 0.562015503875969

Classification Report:
              precision    recall  f1-score   support

           0       1.00      0.09      0.17        11
           1       0.68      0.44      0.54        43
           2       1.00      0.05      0.10        20
           3       0.53      0.98      0.69       119
           4       1.00      0.05      0.09        22
           5       0.00      0.00      0.00        20
           6       0.75      0.26      0.39        23

    accuracy                           0.56       258
   macro avg       0.71      0.27      0.28       258
weighted avg       0.63      0.56      0.46       258
```

6. Gradient Boosting :

```
Gradient Boosting Model
Accuracy: 0.7209302325581395

Classification Report:
              precision    recall  f1-score   support

           0       0.80      0.36      0.50        11
           1       0.81      0.70      0.75        43
           2       0.86      0.60      0.71        20
           3       0.72      0.93      0.81       119
           4       0.60      0.55      0.57        22
           5       0.57      0.20      0.30        20
           6       0.65      0.57      0.60        23
```

```
    accuracy                           0.72       258
   macro avg       0.72      0.56      0.61       258
weighted avg       0.72      0.72      0.70       258
```

7. Grid Search on an SVM:



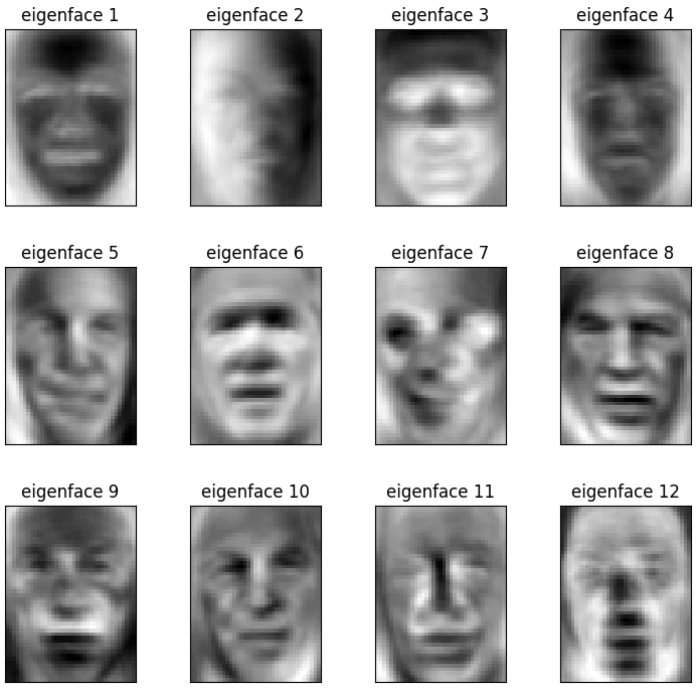
```python
param_grid = {
    'C': [0.1, 1, 10, 100, 1000],
    'gamma': ['scale', 'auto', 0.01, 0.1, 1],
    'kernel': ['rbf', 'linear', 'poly', 'sigmoid'],
    'degree': [2, 3, 4],  # Only used for 'poly' kernel
    'class_weight': [None, 'balanced']  # Useful for imbalanced dataset
}

# Grid search for SVC
svc = SVC()
grid_search = GridSearchCV(svc, param_grid, cv=5, scoring='accuracy', verbose=1)
grid_search.fit(X_train_pca, y_train)
best_svc = grid_search.best_estimator_
y_pred = best_svc.predict(X_test_pca)

accuracy = accuracy_score(y_test, y_pred)
precision = precision_score(y_test, y_pred, average='weighted')
recall = recall_score(y_test, y_pred, average='weighted')
f1 = f1_score(y_test, y_pred, average='weighted')
classification_report_result = classification_report(y_test, y_pred)

print(f"Best SVC Model: {best_svc}")
print(f"Best Parameters: {grid_search.best_params_}\n")
print(f"Accuracy: {accuracy}\n")
print(f"Classification Report:\n{classification_report_result}\n")
```

Results:

```
Fitting 5 folds for each of 600 candidates, totalling 3000 fits
Best SVC Model: SVC(C=0.1, class_weight='balanced', degree=2, kernel='linear')
Best Parameters: {'C': 0.1, 'class_weight': 'balanced', 'degree': 2, 'gamma': 'scale', 'ker

Accuracy: 0.8217054263565892

Classification Report:
              precision    recall  f1-score   support

           0       0.64      0.64      0.64        11
           1       0.72      0.84      0.77        43
           2       0.82      0.70      0.76        20
           3       0.91      0.87      0.89       119
           4       0.68      0.77      0.72        22
           5       0.84      0.80      0.82        20
           6       0.82      0.78      0.80        23

    accuracy                           0.82       258
   macro avg       0.78      0.77      0.77       258
weighted avg       0.83      0.82      0.82       258
```

4.(c) Visualize 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)

▼ EIGENFACES ANALYSIS



Subset of the first few eigenfaces (in order of highest variance to lowest)

1. Observations:

- Eigenfaces appear ghost-like and resemble a mask.
- They seem to capture the underlying common aspects of a face, such as slits for eyes, a general structure for the nose, mouth and the eyebrows seem to all be present from the very first ***eigenface.***
- The Eigenfaces also seem to capture various lighting conditions as evident in ***eigenface2*** and ***eigenface3*** where the light appears to be from the left and downwards respectively, thus throwind a shadow on the rest of the face.
- ***eigenface 5*** seems to resemble a mask the most and ***eigenface 6*** almost looks like the face of **Bush**.
- The rest of the eigenfaces seem to appear more crooked (***eigenface 7***) and seem to be capturing the less represented faces in the dataset.

- They also seem to capture expressions, beards and the features captured become more and more subtle as we look into more eigenfaces with less captured variance.

2. Fails:

These are the images that the transformed data falsely classifies:


True: Bush
Pred: Powell
True: Bush
Pred: Powell
True: Chavez
Pred: Schroeder
True: Powell
Pred: Bush
True: Powell
Pred: Bush
True: Sharon
Pred: Bush
True: Rumsfeld
Pred: Schroeder
True: Powell
Pred: Schroeder
True: Chavez
Pred: Schroeder
True: Powell
Pred: Chavez
True: Bush
Pred: Sharon
True: Powell
Pred: Sharon
True: Rumsfeld
Pred: Powell
True: Sharon
Pred: Schroeder
True: Powell
Pred: Bush
True: Bush
Pred: Powell
True: Schroeder
Pred: Bush
True: Bush
Pred: Powell
True: Bush
Pred: Blair
True: Chavez
Pred: Schroeder
True: Rumsfeld
Pred: Sharon
True: Bush
Pred: Powell
True: Bush
Pred: Powell
True: Bush
Pred: Blair
True: Rumsfeld
Pred: Bush
True: Blair
Pred: Bush
True: Bush
Pred: Powell
True: Schroeder
Pred: Bush
True: Blair
Pred: Powell
True: Bush
Pred: Rumsfeld
True: Bush
Pred: Rumsfeld
True: Bush
Pred: Powell
True: Bush
Pred: Powell
True: Bush
Pred: Blair
True: Schroeder
Pred: Bush
True: Powell
Pred: Bush
True: Sharon
Pred: Schroeder
True: Blair
Pred: Powell
True: Rumsfeld
Pred: Powell
True: Sharon
Pred: Bush
True: Blair
Pred: Schroeder
True: Rumsfeld
Pred: Bush
True: Bush
Pred: Rumsfeld
True: Schroeder
Pred: Chavez
True: Blair
Pred: Powell

Falsely Classified Faces.

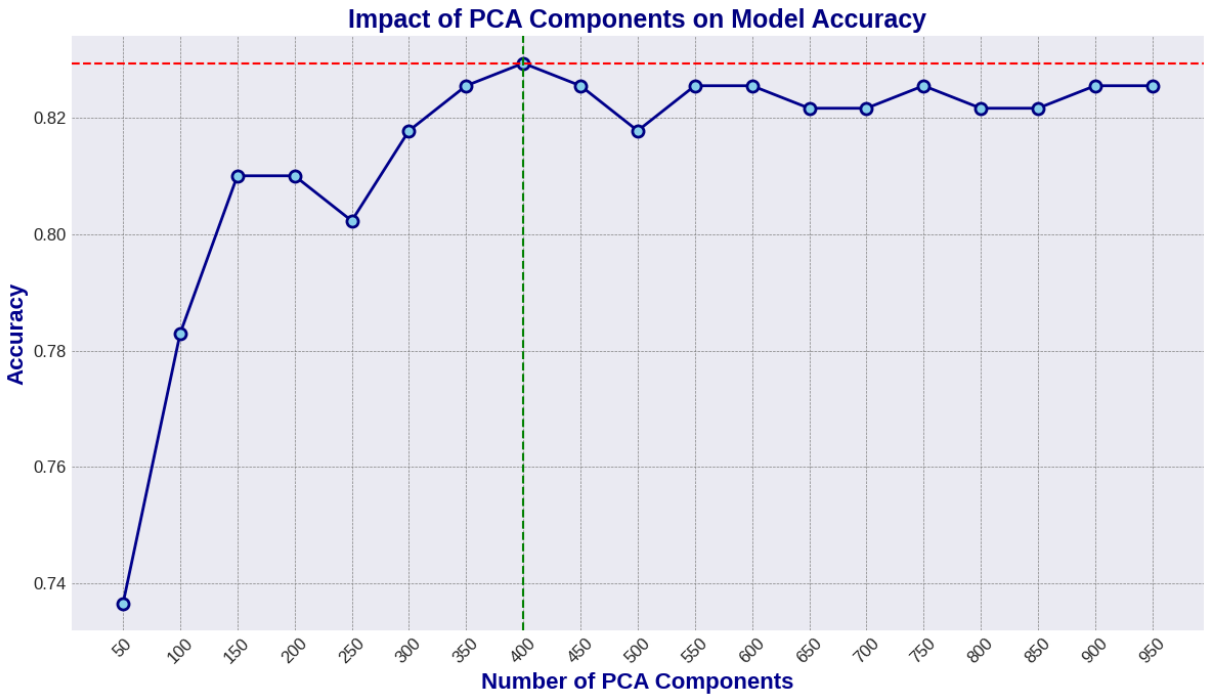
- It can be observed that even the best model appears to fail on a few faces, the causes as inferred from the above can be:
  - The main cause appears to be the difference in pose, most of the failed images contain faces which aren't centered and straight.
  - The other huge cause appears to be the expression in these images.
  - Both of the above reasons make the face divert from the facial geometry learned from the eigenfaces by the model, thus incorrectly classifying them.
  - One more, reason also appears to be lighting conditions, and another reason appears to be the image quality, some images are just not great in quality and look very pixelated (albeit in a less number).

3. Ways to make the model better:

- The model can be made more robust to lighting and pose variations by applying data augmentation techniques on the train data, such as resizing the image, rotating the image, adding some noise into the image, blurring the image slightly etc.
- The above approach can make the model more robust to:
  - **Sensitivity to Lighting and Shadow**
  - **Pose Variations**
  - **Other Foreign Elements such as scars, beards, accessories**
- The model's fundamental approach can be altered to better preseve the spacial relation between pixels rather than straight up flattening the images and then applying eigenfaces on them. Maybe eigenfaces could be applied on a 30*30 matrix itself, not very sure how the mathematics of that would turn out.
- Modern Computer Vision techniques can always be used. (Ex: CNNs, Vision Transformers etc)

▼ Task 5:

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



- The relationship between the number of PCA components and the model accuracy, tends to increase first as the PCA captures more variance in the data as the components increase until a certain spot.
- After a certain maxima, PCA incorporates more noise and learns the more specific features which fail to generalize to the dataset as a whole, here the model's performance almost plateaus and decreases as well as the model tends to overfit on the noise, while not learning anything meaningful.
- From the above plot we observe that the optimal point seems to be at `n_components = 400`; we can infer from this that PCA is a very useful technique, as we can just drop more than half of the features and only retain the more useful data, thus, saving compute while also extracting maximum performance from our model.

▼ Acknowledgments

References:

- Perceptron:
  - https://towardsdatascience.com/perceptron-learning-algorithm-d5db0deab975
  - https://cs.fit.edu/~dmitra/Cplus/2019Spr/assignment5-2.pdf
  - https://www.scaler.com/topics/machine-learning/perceptron-learning-algorithm/

- PCA:

- https://youtu.be/FgakZw6K1QQ?si=xv9_BRgamdhpwga-
- https://medium.com/@megha.natarajan/understanding-cumulative-explained-variance-in-pca-with-python-653e3592a77c
- https://www.jcchouinard.com/pca-explained-variance/

- Eigenfaces:
  - https://scikit-learn.org/stable/auto_examples/applications/plot_face_recognition.html
  - https://scikit-learn.org/stable/modules/generated/sklearn.datasets.fetch_lfw_people.html#sklearn.dataset

---

- THE END -