Face Recognition By Principal Component Analysis

Zahra Mojtahedin

***Abstract*— In this research report, we consider face recognition as a two-dimensional problem, avoiding the need to recover three-dimensional geometry. It takes advantage of the fact that faces are typically upright and can be described by a small set of characteristic 2-D views. The system projects face images onto a feature space that captures significant variations among known faces, using "eigenfaces" as the key features. These eigenfaces are the eigenvectors of the face image set and do not necessarily correspond to specific facial features. To recognize a face, the system compares the weights of its eigenface features with those of known individuals. Notably, this approach allows for unsupervised learning and recognition of new faces and can be easily implemented using a neural network architecture.**

Keywords: ORL Dataset | Noise | Rotation | Robustness

I. INTRODUCTION

Face recognition is useful in the realm of neuroscience for several reasons. Studying how the brain processes and recognizes faces can provide valuable insights into the mechanisms of visual perception, memory, and social cognition. By using face recognition technology, researchers can track how the brain responds to different facial features, expressions, and emotional cues, helping to understand the neural basis of facial perception and social interaction.

Moreover, face recognition can be used in neuroscience research to investigate disorders related to facial processing, such as prosopagnosia (face blindness), autism spectrum disorders, and other conditions affecting social cognition. By studying how the brain processes faces in both healthy individuals and those with neurological or psychiatric conditions, researchers can gain a better understanding of the underlying neural mechanisms and potentially develop new interventions or treatments.

Furthermore, face recognition technology can be utilized in neuroimaging studies, such as functional magnetic resonance imaging (fMRI) and electroencephalography (EEG), to monitor brain activity while individuals process and recognize faces. This can help in identifying specific brain regions involved in facial recognition and understanding how neural networks are activated during social interactions.

II. MATERIALS AND METHODS

A. Dataset

In this Homework we use ORL dataset. The term "ORL dataset" commonly refers to the Olivetti Research Laboratory dataset, a prominent face database. This dataset comprises a collection of face images taken at the Olivetti Research Laboratory in Cambridge, UK, spanning the period between April 1992 and April 1994. Encapsulated within are images of 40 different subjects, each depicted from various angles and illuminated under different lighting conditions.

Frequently utilized in the realms of face recognition, machine learning, and computer vision, the ORL dataset serves as a popular resource for testing. Researchers leverage this dataset to pioneer and assess algorithms for tasks such as facial recognition, face detection, and image processing.

The ORL dataset has played a pivotal role in propelling advancements in face recognition technologies and has established itself as a standard for appraising the efficacy of diverse algorithms. (Fig 1)

Fig. 1. Samples of the Dataset

III. PRINCIPAL COMPONENT ANALYSIS

Principal Component Analysis (PCA) is a statistical technique used to reduce the dimensionality of a dataset while retaining as much variance as possible. Mathematically, PCA can be described as follows:

Given a dataset of n observations with p variables represented by an $n \times p$ matrix X, where each row represents an observation and each column represents a variable, PCA seeks to find a set of p linearly uncorrelated variables, called principal components, that are ordered by the amount of variance they explain in the original data.

The first principal component, denoted as Z_1, is a linear combination of the original variables given by:

$$Z_1 = \phi_{11}X_1 + \phi_{21}X_2 + \ldots + \phi_{p1}X_p$$

where ϕ_{i1} represents the weights or loadings of the original variables on the first principal component, and X_j represents the jth variable.

The weights are chosen such that they maximize the variance of Z_1, subject to the constraint that $\sum_{j=1}^{p} \phi_{j1}^2 = 1$.

Subsequent principal components $Z_2, Z_3, \ldots, Z_p$ are chosen in a similar manner, subject to the constraints that they

are uncorrelated with the previous principal components and that they maximize the remaining variance.

Collecting the loadings $\phi_{i1}, \phi_{i2}, \ldots, \phi_{ip}$ into an $p \times p$ matrix called Φ, and $Z_1, Z_2, \ldots, Z_p$ into an $n \times p$ matrix called Z, the transformed dataset can be represented as:

$$Z = X\Phi$$

The principal components capture the directions of maximum variance in the original data, and by choosing to keep only the first k principal components which explain most of the variance, the dimension of the data can be reduced to k dimensions.

This process can be achieved using the eigen-decomposition of the covariance matrix of X or through the singular value decomposition (SVD) of X.

Principal Component Analysis (PCA) is a popular technique used for dimensionality reduction and data visualization. It works by transforming the original dataset into a new coordinate system where the axes are ranked in order of importance, with the first axis capturing the most variance in the data.

The mathematical expression for PCA can be written in latex as follows:

Given a dataset of n data points, each represented as a d-dimensional vector $\mathbf{x_i} = (x_{i1}, x_{i2}, \ldots, x_{id})^T$, where $i = 1, 2, \ldots, n$, the goal of PCA is to find a new set of d' orthonormal basis vectors $\mathbf{v_1}, \mathbf{v_2}, \ldots, \mathbf{v_{d'}}$ such that the variance of the projected data onto these basis vectors is maximized.

The first principal component, denoted as $\mathbf{v_1}$, is the unit vector in the direction of the highest variance in the data. The second principal component, $\mathbf{v_2}$, is orthogonal to the first one and points in the direction of the second highest variance, and so on.

The data can then be projected onto the new coordinate system spanned by these principal components, given by $\mathbf{z_i} = (z_{i1}, z_{i2}, \ldots, z_{id'})^T = \mathbf{V}^T\mathbf{x_i}$, where $\mathbf{V}$ is the matrix containing the d' principal components as its columns.

PCA has numerous applications, including but not limited to:

- Dimensionality reduction for high-dimensional data visualization
- Noise reduction and feature extraction
- Data compression and reconstruction
- Identifying patterns and underlying structure in data
- Preprocessing for machine learning tasks such as classification and clustering.

It's important to note that PCA operates under the assumption that the data's relationships are linear and thus works best with continuous variables. This technique aims to reduce the dimensionality of the data while preserving as much variability as possible. However, one crucial aspect to keep in mind is that PCA is highly sensitive to the scaling of the data. Variables measured on different scales can heavily influence the principal components if they are not standardized. Therefore, it is often recommended to standardize the variables before applying PCA to ensure that each variable contributes equally to the analysis, thereby providing more accurate and meaningful results. (Fig 2)

Fig. 2. Original and Reconstructed Image by PCA(0.87 explained Variance)

IV. MLP CLASSIFIER

MLP (Multi-Layer Perceptron) classifier is a type of artificial neural network that's commonly used for classification tasks. It's composed of multiple layers of nodes, including an input layer, one or more hidden layers, and an output layer. The key steps in building a MLP classifier are as follows:

- **Input Layer**: Receives the input data. Each node in this layer represents a feature in the dataset.
- **Hidden Layers**: Process the inputs from the previous layer using weights and biases. The result is passed through an activation function and sent to the next layer.
- **Output Layer**: Produces the final output. For classification, this might be the predicted class; for regression, it might be a continuous value.
- **Forward Propagation**: Data flows from the input layer through the hidden layers to the output layer, with each neuron processing the inputs and passing the results forward.
- **Loss Function**: Compares the predicted output with the actual output to calculate the error (loss).
- **Back-propagation**: The error is propagated back through the network to update the weights and biases, reducing the error in future predictions. This process uses gradient descent or other optimization algorithms.
- **Training**: The network is trained over multiple iterations (epochs), adjusting weights and biases each time to minimize the loss function.

V. ADAM OPTIMIZER

The Adam (Adaptive Moment Estimation) optimizer is an optimization algorithm designed specifically for training deep learning models. It combines the advantages of two other popular optimization algorithms: AdaGrad and RMSProp.

- **Adaptive Learning Rates:** Adam adjusts the learning rate for each parameter individually, which helps the model converge faster and more efficiently.
- **Combines Methods:** It combines the benefits of both Adaptive Gradient Algorithm (AdaGrad) and Root

Mean Square Propagation (RMSProp) to improve the optimization process.

- **Momentum:** Adam uses the concept of momentum to accelerate the convergence of the training process. It calculates an exponentially decaying average of past gradients and past squared gradients.

The key steps in building Adam Optimizer are as follows:

1) **Initialization:**
   Initialize parameters (weights) and their respective moment estimates, typically set to zero. Set hyperparameters: learning rate (α), two exponential decay rates (β_1 and β_2), and a small constant (ϵ) to prevent division by zero.
2) **Compute Gradients:** Compute the gradients of the loss function with respect to the parameters (weights).
3) **Update Biased First Moment Estimate:** Calculate the moving average of the gradient (first moment estimate) using β_1:

$$m_t = \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$$

   Here, m_t is the first moment estimate at time step t and g_t is the gradient at time step t.
4) **Update Biased Second Moment Estimate:** Calculate the moving average of the squared gradient (second moment estimate) using β_2:

$$v_t = \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$$

   Here, v_t is the second moment estimate at time step t.
5) **Compute Bias-Corrected Estimates:** Correct the bias in the first and second moment estimates:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

6) **Update Parameters:** Update the parameters (weights) using the bias-corrected estimates and the learning rate:

$$\theta_t = \theta_{t-1} - \alpha \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

   Here, θ_t represents the parameters at time step t.

Key Points

- **Efficiency:** Adam combines the benefits of AdaGrad (which works well with sparse gradients) and RMSProp (which works well with non-stationary objectives) to achieve better optimization performance.
- **Robustness:** Adam is generally robust to different hyperparameter settings, making it easier to tune and use effectively across various types of models and datasets.

By using Adam, you often achieve faster convergence and better performance in training deep learning models.

VI. Gaussian Noise

To add Gaussian noise to an image, each pixel in the image is modified by adding a random value drawn from a Gaussian distribution. The process of adding Gaussian noise can be described as follows:

1. Generate random numbers: For each pixel in the image, generate a random number from a Gaussian distribution with a specified mean and standard deviation. The mean controls the average value of the noise, while the standard deviation controls the spread or intensity of the noise.

2. Add noise to the pixel values: Add the generated random numbers to the corresponding pixel values in the image. This effectively perturbs the pixel values with random noise.

Mathematically, if I(x, y) represents the intensity of the pixel at position (x, y) in the original image, and N(x, y) represents the Gaussian noise value at the same position, the pixel value in the noisy image, I'(x, y), can be computed as:

$$I'(x, y) = I(x, y) + N(x, y)$$

Adding Gaussian noise to images can be useful for various purposes, such as simulating the effect of noise in real-world imaging conditions, testing the robustness of image processing algorithms, or augmenting datasets for training machine learning models. (Figs 3, 4, and 5)

Fig. 3. Added Gaussian Noise to the Test Images(STD = 25)

Fig. 4. Added Gaussian Noise to the Test Images(STD = 80)

Fig. 5. Added Gaussian Noise to the Test Images(STD = 110)

VII. Salt and Pepper Noise

To add salt and pepper noise to the Persian digital dataset, we can apply the following formula to each pixel in the image:

$$f(x, y) = \begin{cases} 0 & \text{with probability } P \\ 1 & \text{with probability } P \\ I(x, y) & \text{otherwise} \end{cases}$$

Where $I(x, y)$ represents the original intensity of the pixel, and P is the probability of the pixel being corrupted. By

applying this formula, we can simulate the effect of salt and pepper noise on the Persian digital dataset.(Figs 6, 7, and 8)

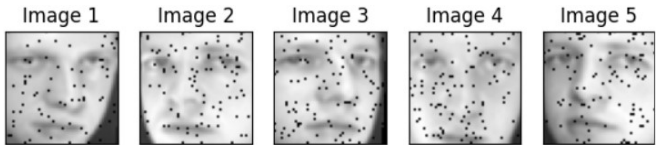

Fig. 6. Added Salt and Pepper Noise to the Test Images(density = 0.02)

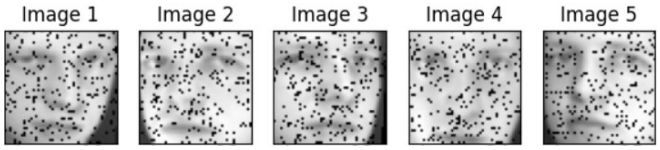

Fig. 7. Added Salt and Pepper Noise to the Test Images(density = 0.04)

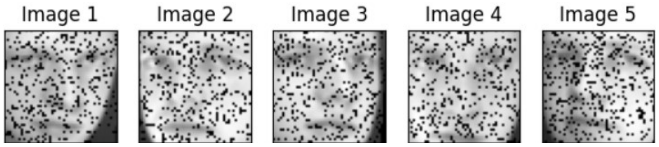

Fig. 8. Added Salt and Pepper Noise to the Test Images(density = 0.08)

VIII. Rotating Images

Here we rotated test images with the angles of 5, 10, and 15. (Figs 9, 10, and 11)

Fig. 9. Rotated test images(5 deg-rates)

Fig. 10. Rotated test images(10 deg-rates)

Fig. 11. Rotated test images(15 deg-rates)

Results

Without using any feature extractor (as shown in Figures 18 and 19), results demonstrate that the system or process accuracy improves as the number of PC components increases. The data indicates a positive correlation between the number of PC components and accuracy, up to a certain point. Beyond this threshold, the benefits of adding more components diminish.

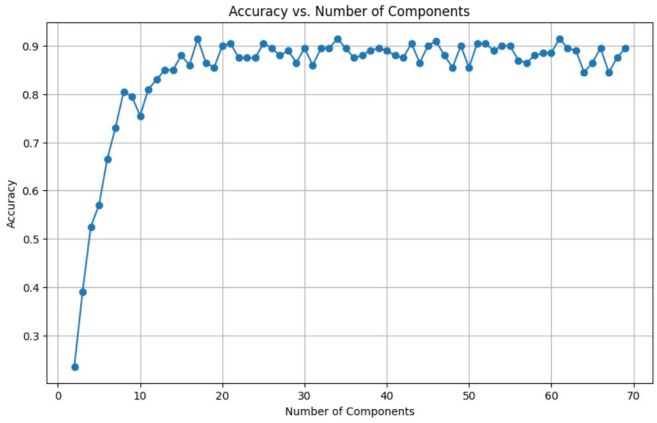

Fig. 12. Accuracy Per Number of Components Without Using any Feature Extractor

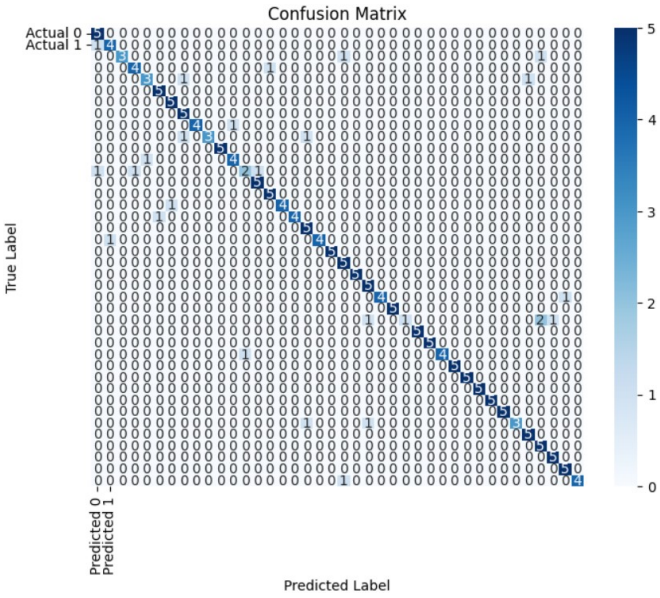

Fig. 13. Confusion Matrix with 17 PCs(ACC = 94.00)

Information gain is used in machine learning and feature extraction to gauge the importance of a feature in making predictions. When creating a classification model, selecting the most relevant features from input data is crucial for accurate predictions.

Information gain feature extraction involves using this metric to select the most informative features for the classification task. It measures how much a feature reduces uncertainty about the classification when its values are known, quantifying the contribution of knowing a feature's value in predicting the class label.

In practical terms, information gain feature extraction typically involves the following steps:

- Calculate Entropy of Target Variable: Measure the uncertainty or randomness in the target variable. High entropy indicates high uncertainty; low entropy indicates

low uncertainty.

- Calculate Conditional Entropy Given a Feature: For each feature, determine the entropy of the target variable given the values of that feature. This shows the uncertainty in the target when feature values are known.
- Compute Information Gain: Compare the original entropy of the target variable with the entropy after splitting the data based on feature values. Higher information gain means the feature is more important.
- Select Top Features: Choose the features with the highest information gain as the most informative for the classification task.

  In Figures 20 and 21(this one is the best performance) you can observe the results of extracting information gain feature before PCA.

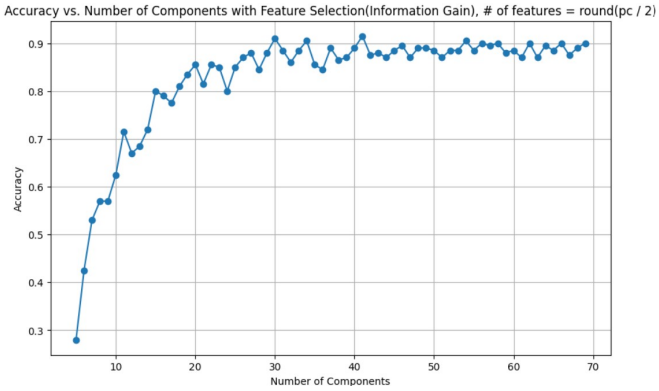

Fig. 14. Accuracy Per Number of Components With Using Information Gain Feature Extractor

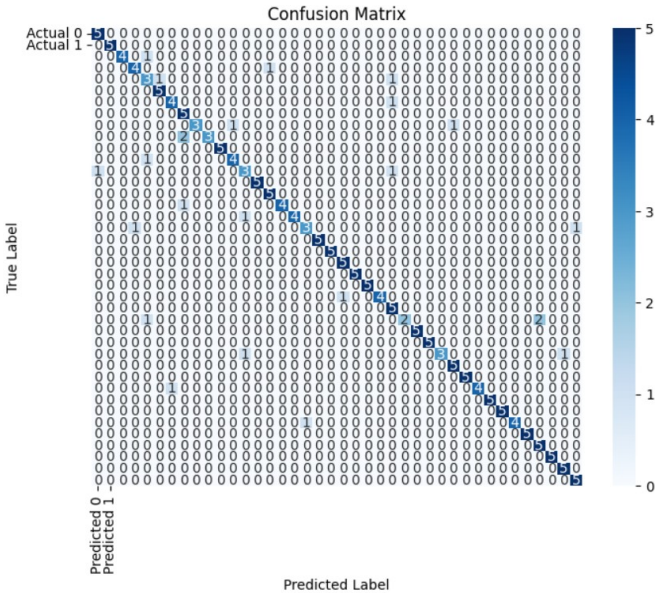

Fig. 15. Confusion Matrix with 41 PCs and Information Gain Feature Extractor(ACC = 92.00)

The Fisher Score feature extractor, or Fisher Discriminant Analysis (FDA), is used for feature selection and dimensionality reduction in machine learning and pattern recognition. It aims to find features that best separate different classes of data.

Maximizes the ratio of between-class variance to within-class variance, identifying features with low variance within each class and high variance between classes. This enhances class distinctions while minimizing within-class variations.

In Figures 22 and 23 (this one is the best performance) you can observe the results of extracting Fisher Score feature before PCA.

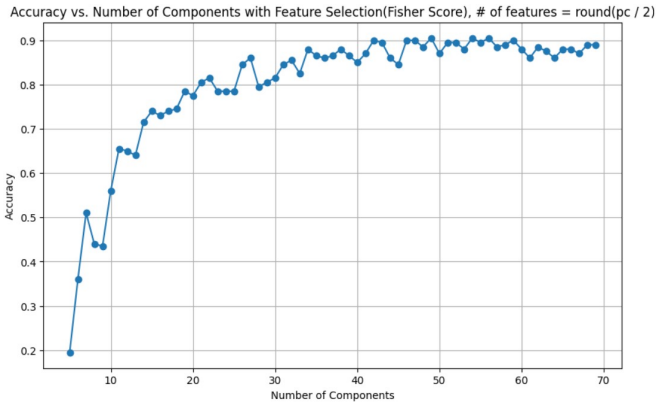

Fig. 16. Accuracy Per Number of Components With Using Fisher Score Feature Extractor

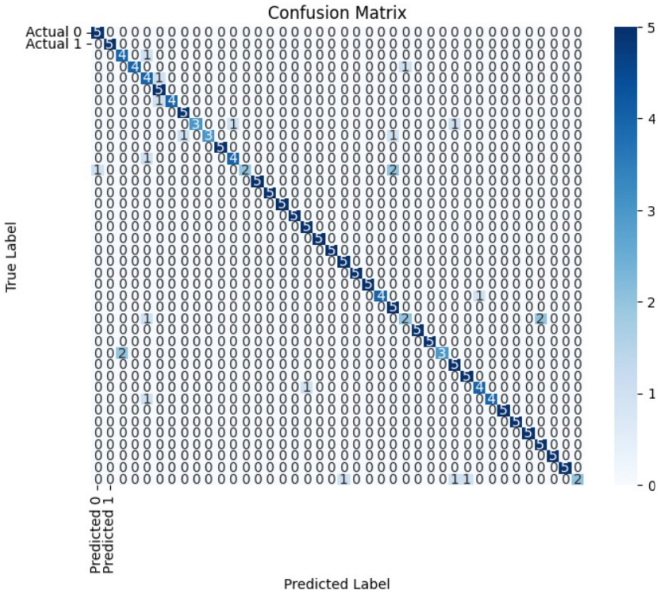

Fig. 17. Confusion Matrix with 49 PCs and Fisher Score Feature Extractor(ACC = 90.00)

A correlation coefficient measures the strength and direction of the relationship between two variables. In feature extraction, it helps identify how strongly each feature in a dataset is related to the target variable, aiding in the selection of the most relevant features for predictive modeling.

Here's how it works:

- Calculate the Correlation Coefficient: Typically, the Pearson correlation coefficient is used to measure the

linear relationship between two variables. It ranges from -1 to 1, where 1 indicates a perfect positive relationship, -1 indicates a perfect negative relationship, and 0 indicates no linear relationship.

- Feature Selection: After calculating the correlation coefficient between each feature and the target variable, select the features with the highest absolute correlation coefficients as the most relevant for your predictive model.
- Feature Importance: The correlation coefficient also provides insights into the relative importance of different features in the dataset. Features with higher absolute correlation coefficients are more important in explaining the variation in the target variable.

In Figures 24 and 25(this one is the best performance) you can observe the results of extracting correlation coefficient feature before PCA.

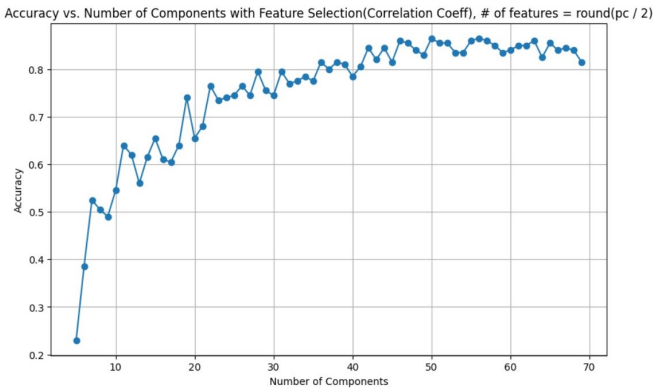

Fig. 18. Accuracy Per Number of Components With Using Correlation Coefficient Feature Extractor

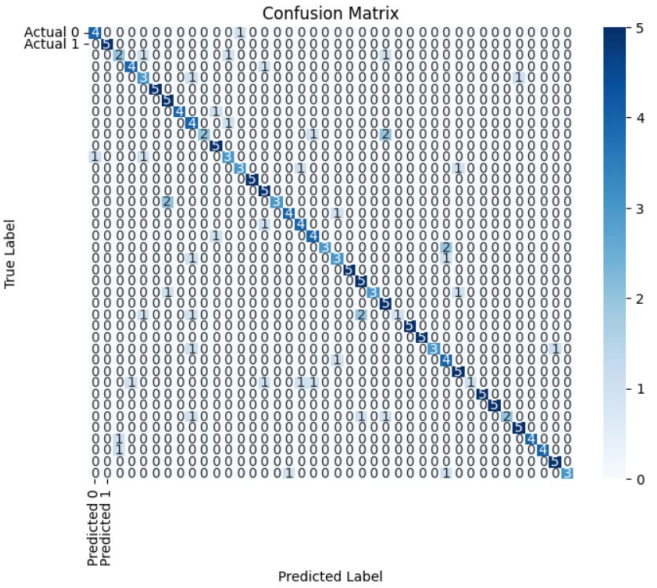

Fig. 19. Confusion Matrix with 50 PCs and Correlation Coefficient Feature Extractor(ACC = 88.50)

Variance threshold feature extractor is a technique used in feature selection to remove low-variance features from a dataset. This method calculates the variance of each feature, and any feature whose variance doesn't meet a certain threshold is removed from the dataset.

Here's how it works:

- Calculate the variance of each feature in the dataset.
- Remove the features whose variance doesn't surpass the specified threshold.
- The remaining features are considered to be the ones with higher variance and are retained for further analysis.

In Figures 26 and 27(this one is the best performance) you can observe the results of extracting variance threshold feature before PCA.

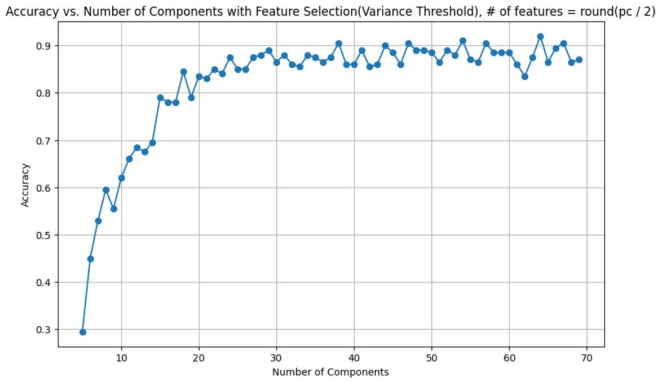

Fig. 20. Accuracy Per Number of Components With Using Variance Threshold Feature Extractor

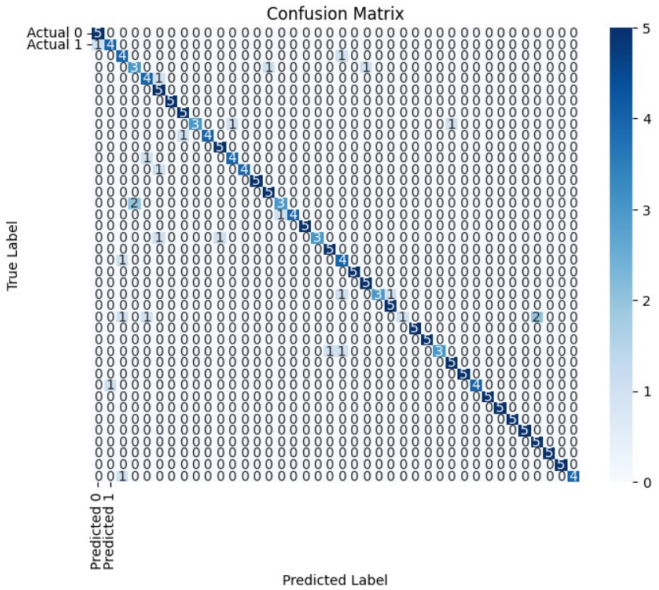

Fig. 21. Confusion Matrix with 64 PCs and Variance Threshold Feature Extractor(ACC = 92.50)

Here we have added Gaussian Noise to the test data and plotted accuracy per number of components. As you see in the figures 28 to 38, accuracy will decrease while increasing standard deviation.

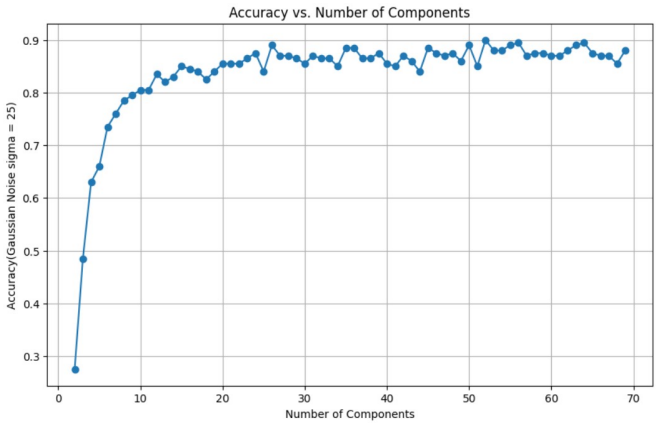

Fig. 22. We have added Gaussian Noise(STD = 25) to the Test Images and Tested the Dataset

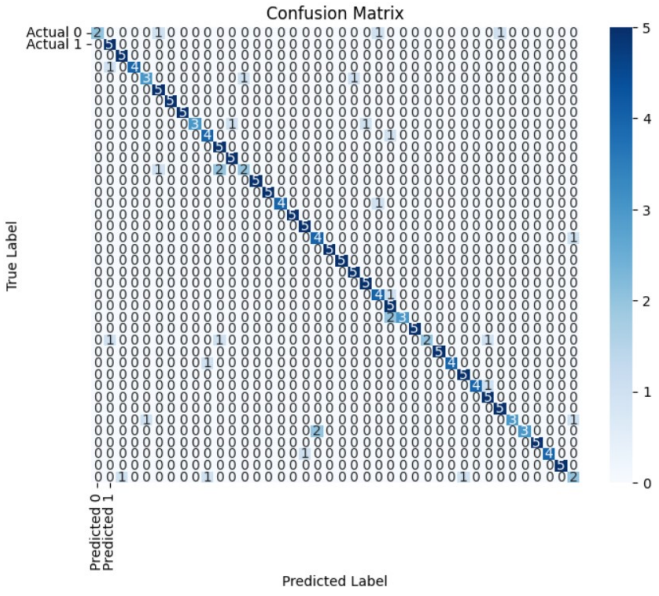

Fig. 23. Confusion Matrix with 52 PCs for STD = 25 (ACC = 90.00)

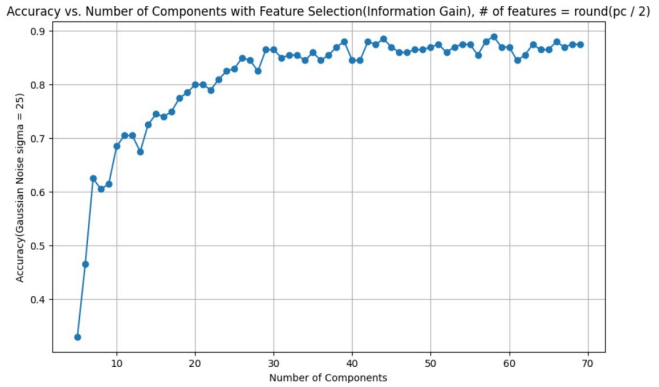

Fig. 24. We have added Gaussian Noise(STD = 25) to the Test Images and Tested the Dataset and Used Information Gain Feature Extractor

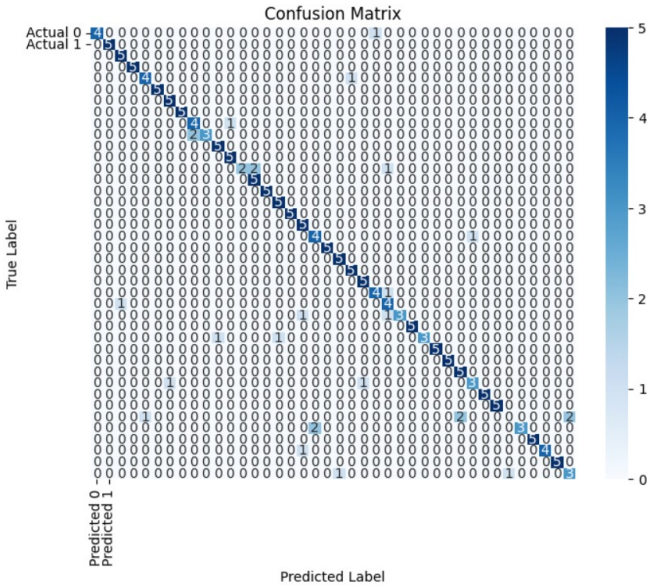

Fig. 25. Confusion Matrix with 58 PCs for STD = 25 and Used Information Gain Feature Extractor(ACC = 89.00)

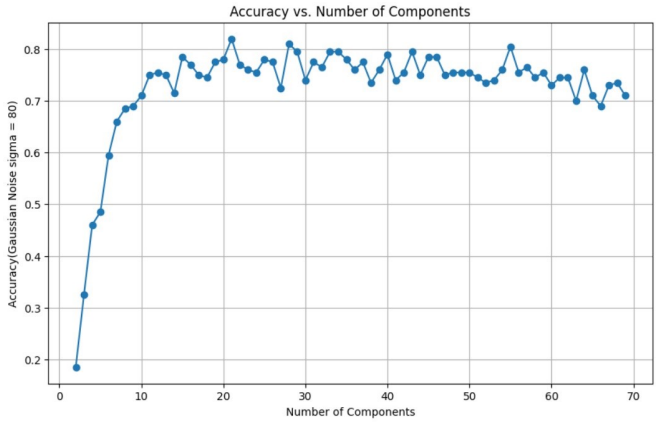

Fig. 26. We have added Gaussian Noise(STD = 80) to the Test Images and Tested the Dataset

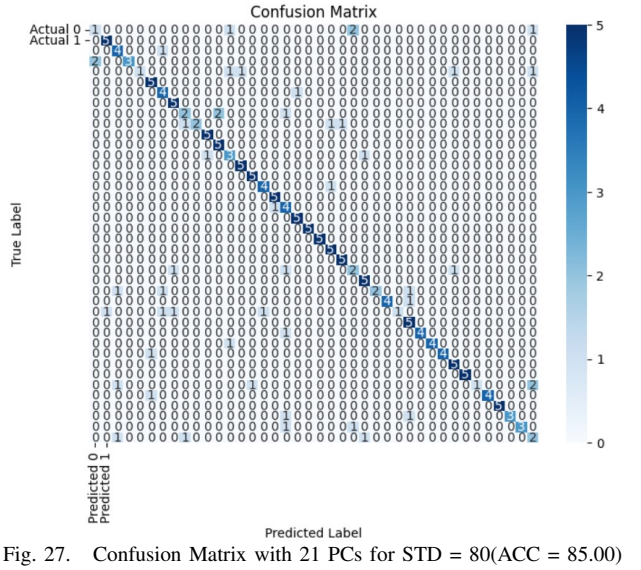

Fig. 27. Confusion Matrix with 21 PCs for STD = 80(ACC = 85.00)

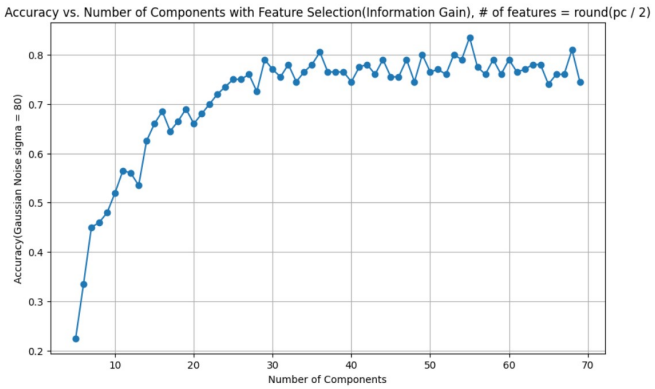

Fig. 28. We have added Gaussian Noise(STD = 80) to the Test Images and Tested the Dataset and Used Information Gain Feature Extractor

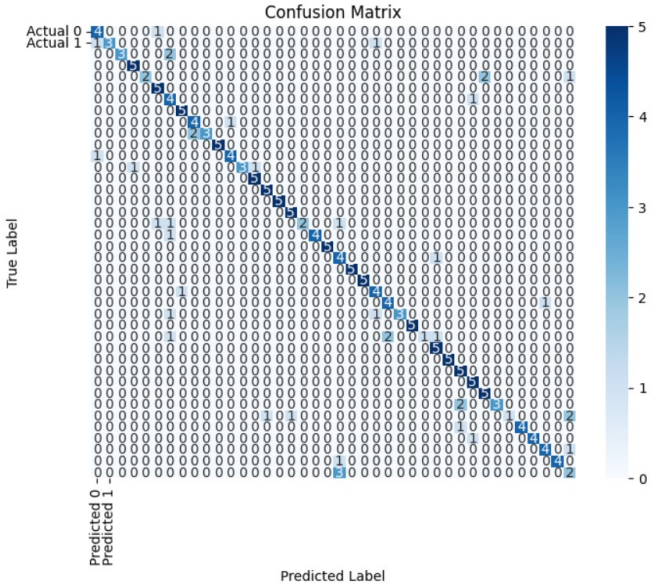

Fig. 29. Confusion Matrix with 55 PCs for STD = 80(ACC = 86.50)

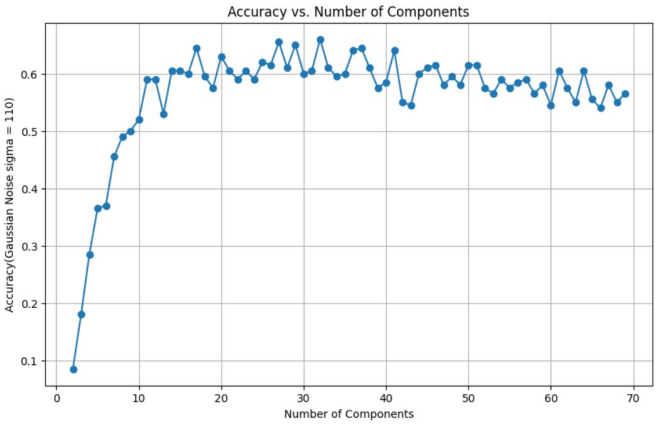

Fig. 30. We have added Gaussian Noise(STD = 110) to the Test Images and Tested the Dataset

Here we have added salt and pepper noise to the test data and plotted accuracy per number of components. As you see in the figures 39 to 45, accuracy will decrease while increas-

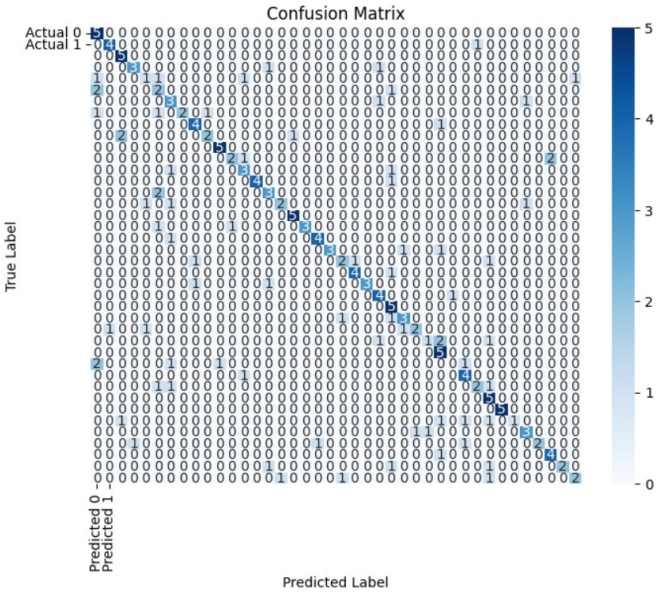

Fig. 31. Confusion Matrix with 32 PCs for STD = 110(ACC = 67.50)

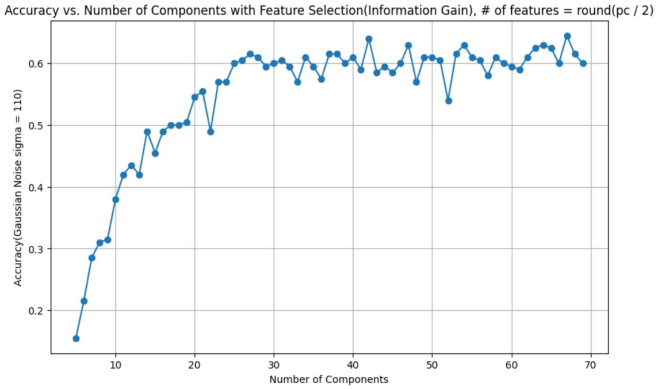

Fig. 32. We have added Gaussian Noise(STD = 110) to the Test Images and Tested the Dataset and Used Information Gain Feature Extractor

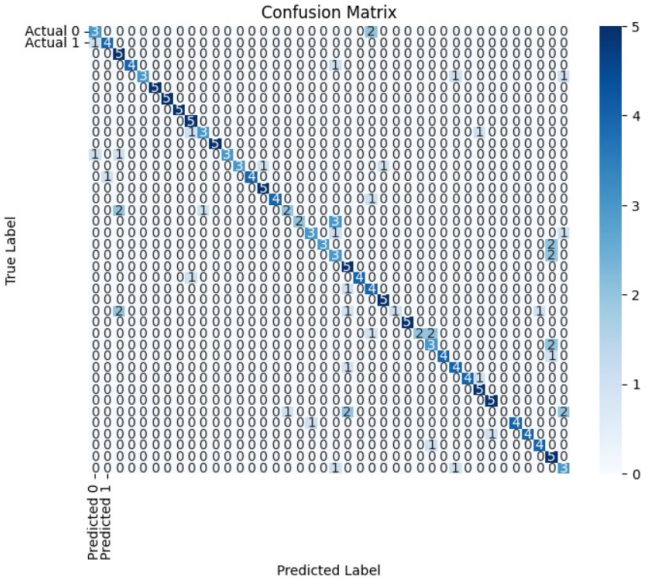

Fig. 33. Confusion Matrix with 67 PCs for STD = 110(ACC = 68.00)

ing salt and pepper density. For more outputs and confusion matrix, please refer to the file `HW2_part3(.ipynb)`.

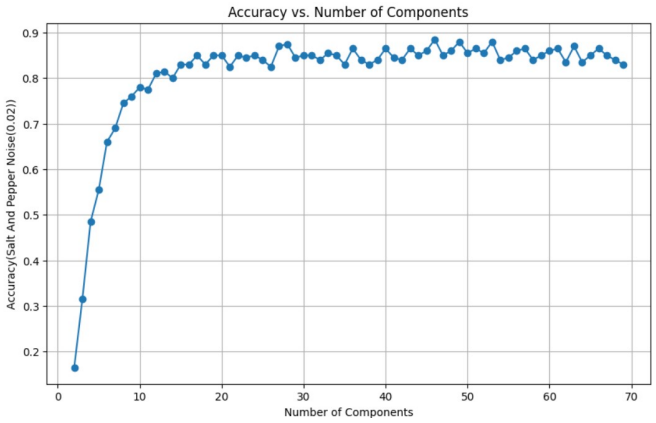

Fig. 34. Added Salt and Pepper Noise(Den = 0.02)

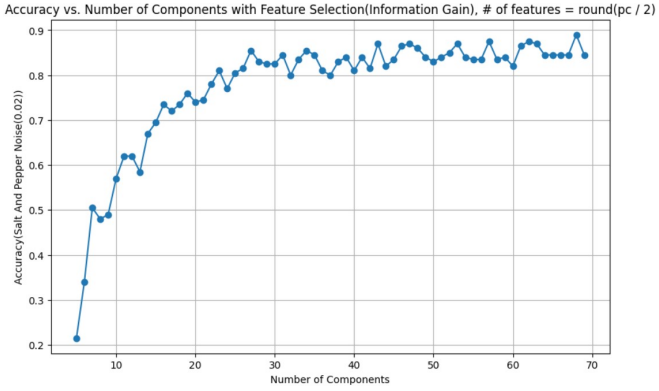

Fig. 35. Added Salt and Pepper Noise(Den = 0.02) to the Test Images and Tested the Dataset and Used Information Gain Feature Extractor

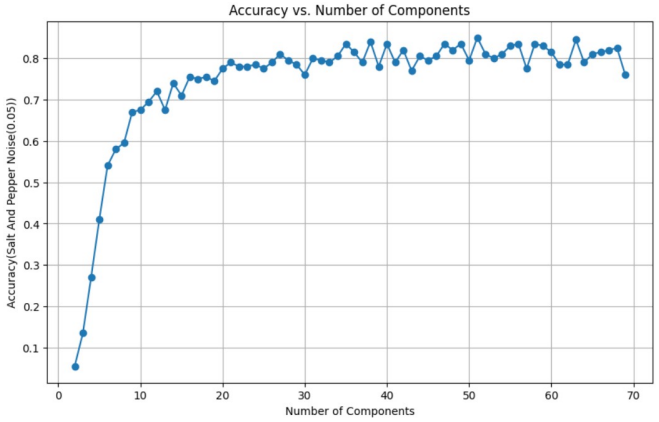

Fig. 36. Added Salt and Pepper Noise(Den = 0.04)

Here we have rotated the test data and plotted accuracy per number of components. As you see in the figures 46 to 52 accuracy will decrease while increasing rotation angle.

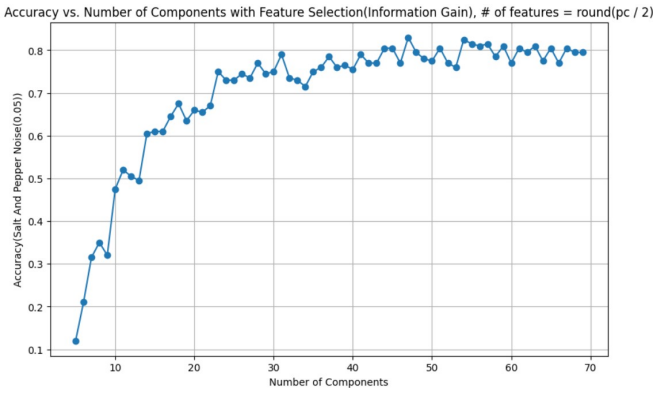

Fig. 37. Added Salt and Pepper Noise(Den = 0.04) to the Test Images and Tested the Dataset and Used Information Gain Feature Extractor

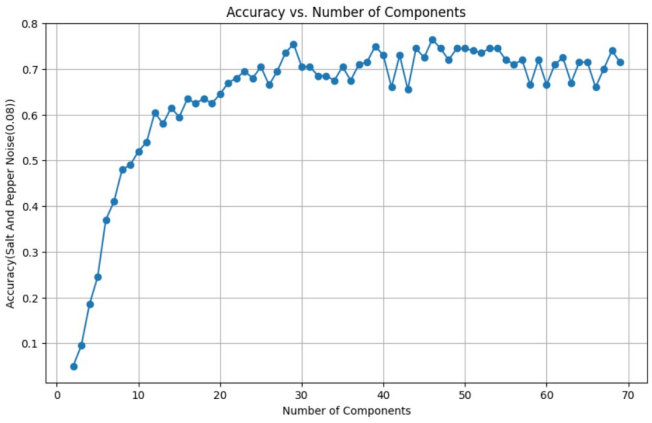

Fig. 38. Added Salt and Pepper Noise(Den = 0.08)

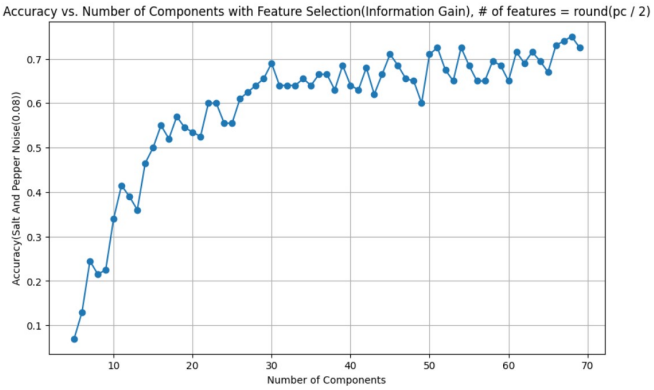

Fig. 39. Added Salt and Pepper Noise(Den = 0.08) to the Test Images and Tested the Dataset and Used Information Gain Feature Extractor

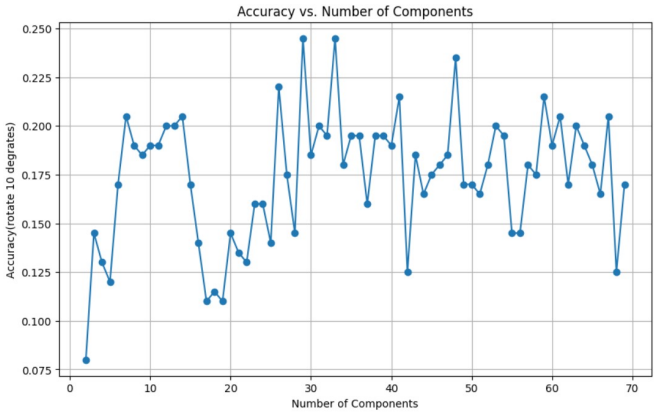

Fig. 40. Rotated test images(rotation angle = 5)

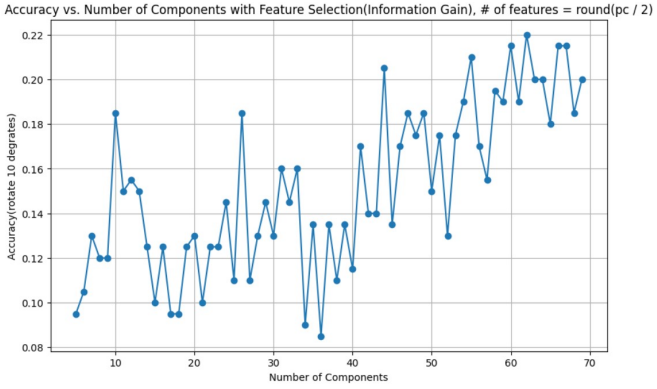

Fig. 41. Rotated test images(rotation angle = 5) and Tested the Dataset and Used Information Gain Feature Extractor

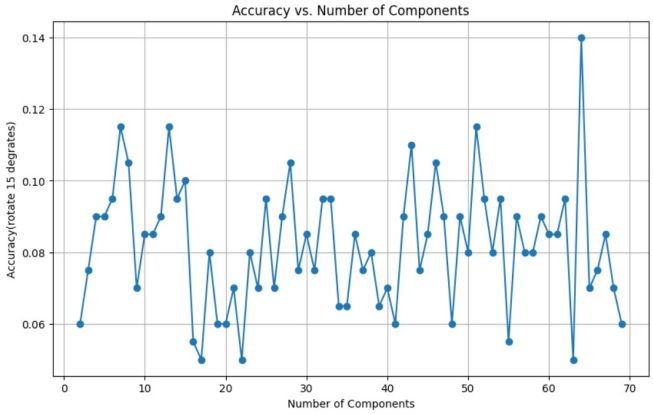

Fig. 42. Rotated test images(rotation angle = 10)

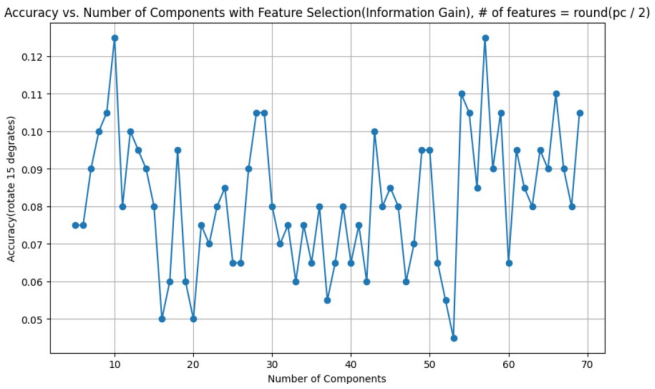

Fig. 43. Rotated test images(rotation angle = 10) and Tested the Dataset and Used Information Gain Feature Extractor

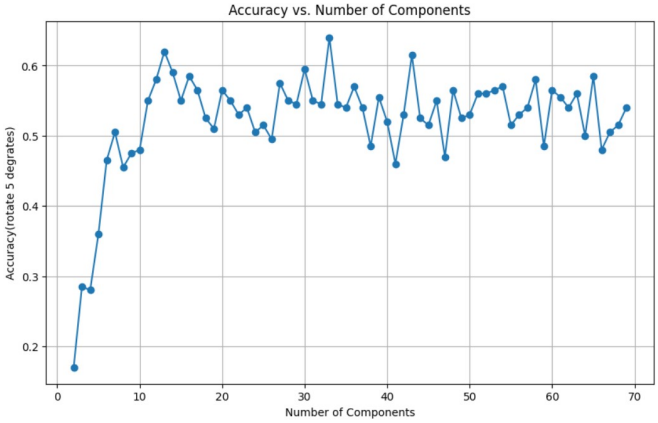

Fig. 44. Rotated test images(rotation angle = 15)

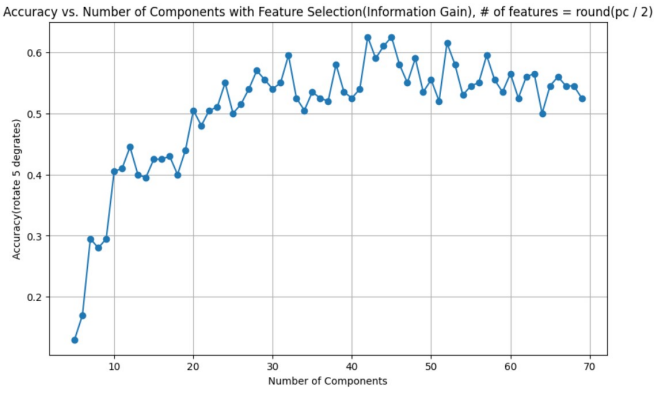

Fig. 45. Rotated test images(rotation angle = 15) and Tested the Dataset and Used Information Gain Feature Extractor

IX. Conclusion

- Adding Gaussian or salt and pepper noise to test data is likely to decrease the accuracy of a model that has been trained on clean data. Noise generally introduces variations and distortions to the data, making it more challenging for the model to accurately predict the correct labels.
  Gaussian noise typically adds random values that follow a Gaussian distribution to the data, while salt and pepper noise randomly corrupts some of the pixels in the images by setting them to the extreme values of the data range (i.e., salt for setting to maximum, and pepper for setting to minimum).
  When a model trained on clean data is tested on noisy data, it may struggle to make accurate predictions due to the unexpected variations introduced by the noise. This could lead to a decrease in accuracy, as the model was not trained to handle noisy inputs.
  So, in general, adding Gaussian or salt and pepper noise

to the test data is likely to decrease the accuracy of the model's predictions. In this report we observed that accuracy will decrease by adding gaussian or salt and pepper noise.

- Using less training data can often lead to a decrease in accuracy when training machine learning models. With less data, the model may not be able to capture the full complexity and variability of the underlying patterns in the data, resulting in a model that is less accurate in making predictions.
  There are several reasons for this:
  - Generalization: With less training data, the model may not be able to generalize well to new, unseen examples, leading to over-fitting or under-fitting.
  - Noise: More data can help in capturing the underlying signal amidst the noise in the data. With less data, the noise could have a larger impact on the model's performance.
  - Complexity: In some cases, particularly for complex models, a smaller dataset might not provide enough information to fully capture the complexity of the underlying relationship in the data.

  In this report we observed that accuracy will decrease by decreasing the number of train data.
- If you rotate the test data, the accuracy of the model should not change substantially. The purpose of the test data is to evaluate the performance of the model on unseen examples, and by rotating the data, you are essentially changing its orientation but not its underlying patterns. However, drastic transformations or distortions in the test data could potentially affect the accuracy if the model was not trained to handle such variations.
  Our random forest model does not handle rotation variation. So the accuracy will decrease by rotating with greater angles.
- If the brightness adjustment does not significantly alter the visual appearance of the images or the data being used, the change in accuracy may be minimal. However, if the adjustments are substantial, it could affect how well the model generalizes to the adjusted data, potentially leading to a decrease in accuracy. It's important to note that any changes to the test data should be done carefully, as they should reflect the real-world conditions the model will encounter. Also for future works, we can note that the effect of brightness adjustment on accuracy might depend on the type of machine learning model being used, the specific data, and the nature of the task at hand.

REFERENCES

[1] M. Turk and A. Pentland, "Eigenfaces for Recognition", Journal of Cognitive Neuroscience, vol. 3, no. 1, pp. 71-86, 1991