IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

A Comparative Analysis of PCA for Dimensionality Reduction in Diverse Datasets

PRABIN BOHARA[1], PRABIN SHARMA POUDEL[2], AND SANTOSH PANDEY.[3]

[1]Institute of Engineering, Thapathali Campus, Kathmandu, Nepal (e-mail: prabinbohara10@gmail.com)
[2]Institute of Engineering, Thapathali Campus, Kathmandu, Nepal (e-mail: prabinsharmapoudel@gmail.com)
[3]Institute of Engineering, Thapathali Campus, Kathmandu, Nepal (e-mail: suntoss.pandey@gmail.com)

* All Authors contributed equally to this work.

ABSTRACT Decision tree analysis is a popular and widely used machine learning technique for classification tasks. This project applied decision tree analysis to the Dry Bean dataset, aiming to develop an accurate classification model for different dry bean varieties based on their physical properties and shape characteristics. The dataset was pre-processed, and a decision tree algorithm was trained and optimized using cross-validation. Performance evaluation metrics, including accuracy, precision, recall, and F1-score, were used to assess the model. The decision tree model achieved high accuracy and favourable results across multiple metrics. Visualizing the decision tree structure provided insights into important features and decision paths. Overall, this project highlights the effectiveness of decision tree analysis in classifying dry bean varieties based on their physical properties and shape characteristics. The developed decision tree model can be a useful tool for automatic classification of dry bean varieties in agricultural and food industry applications, facilitating quality control and product differentiation.

INDEX TERMS Decision Tree, cross-validation

I. INTRODUCTION

A Decision tree analysis is a visual representation that illustrates possible outcomes resulting from a sequence of interconnected choices. It provides a framework for evaluating different factors and decisions to determine the most favourable outcome. Typically, a decision tree analysis begins with a central decision and expands into branches representing additional choices or potential results. It provides a structured and intuitive approach to analyse complex datasets. This report focuses on applying decision tree analysis to the Dry Bean dataset obtained from the UCI Machine Learning Repository. The dataset comprises physical properties and shape characteristics of various dry bean varieties consumed globally. The primary goal is to develop a decision tree model capable of accurately classifying dry bean varieties based on their attributes. To achieve this, we perform crucial pre-processing steps, including handling missing values, normalizing numerical features, and encoding categorical variables. Subsequently, the dataset is divided into training and testing sets to evaluate the model's performance.

The report then proceeds with training the decision tree algorithm and optimizing its hyperparameters through cross-validation. Various evaluation metrics such as accuracy, precision, recall, and F1-score are utilized to assess the effectiveness of the decision tree analysis on the Dry Bean dataset. Additionally, the decision tree structure is visualized to gain valuable insights into the significant attributes and decision paths contributing to the classification process. By showcasing the applicability of decision tree analysis, this report emphasizes its potential in accurately classifying dry bean varieties based on their physical properties and shape characteristics. It highlights the relevance of decision tree analysis in agricultural and food industry applications, including quality control and product differentiation.

II. METHODOLOGY

A. THEORY

1) Background

The Decision tree is an intuitive non-parametric supervised machine-learning model that can be used for classification as well as regression tasks by creating a tree-like flowchart structure to make predictions based on input features. It mimics the human decision-making process as the decision

rules and splits in the tree structure make it clear how the input features influence the final predictions or classifications. Each internal node in the decision tree represents a test on an attribute/feature, each leaf node denotes the class label, and each branch represents the outcome of the test leading to the class label in the leaf node. The classification rule can be obtained by tracing the paths from root to leaf. Decision trees offer a measure of feature importance, which can help us identify the most influential factors driving the predictions or classifications.

A decision tree algorithm called Iterative Dichotomizer (ID3) was developed in the late 1970s. A successor of ID3 called C4.5 was presented later which became a standard. Likewise, CART (Classification and Regression Trees) was put forward which described the generation of binary trees. These algorithms are called greedy algorithms where the top-down approach of recursively dividing and conquering is done to construct the tree.

2) Decision Tree Induction

Decision tree induction starts with a dataset containing training examples with their corresponding class labels. The greedy algorithms mentioned identify the best feature to split the dataset based on certain criteria like information gain. The selected feature is used as the root (topmost node) of the decision tree, and the dataset is split into subsets based on the feature values. The process is repeated recursively for each subset, creating branches and sub-trees until a stopping condition is met. The resulting decision tree can be visualized as a flowchart-like structure, with each node representing a test on a specific feature. To make predictions on unseen data, the algorithm traverses the decision tree by following the path based on the feature values of the input instance until it reaches a leaf node. During the induction of the decision trees, there might be various stopping criteria which helps the tree become less prone to being complex and avoiding overfitting.

3) Attribute Selection Criterion

It's a heuristic to select the splitting criteria that can separate the given data into individual classes in the best way possible. Different attribute selection measures have their own calculation methods and criteria for evaluating the importance of attributes. Some of them include Information gain, Gini Index, Gain ratio, Chi-Square, Relieff, etc.

a: Information Gain

Information gain is a widely used attribute selection measure in decision tree algorithms such as ID3. It quantifies the amount of information gained about the target variable by splitting the data based on a specific attribute. It is based on the concept of entropy, which measures the impurity or uncertainty of a dataset.

$$E = -\sum_{i=1}^{N} p_i \log_2(p_i) \tag{1}$$

Considering a dataset with N classes, 'pi' is the probability of randomly selecting an example in class 'i'. The formula is used to calculate the initial entropy before split (Ebefore) and the entropy after the split (Esplit). Then the information gain is calculated using the formula:

$$IGain = E_{\text{before}} - E_{\text{split}} \tag{2}$$

The more the entropy removed, the greater the information gain. The higher the information gain, the better the split. The attribute with the highest information gain is selected as the best attribute for splitting.

b: Gini Index

The Gini index is a measure of impurity or inequality often used in decision tree algorithms such as CART. The formula to calculate the Gini index is:

$$Gini = 1 - \sum_{i=1}^{n} (p_i)^2 \tag{3}$$

The Gini index ranges from 0 to 1, where 0 indicates a perfectly pure node (all instances belong to the same class) and 1 indicates maximum impurity (instances are evenly distributed across classes). To calculate the Gini index for a specific attribute, the dataset is split based on the attribute's values, and the Gini index is calculated for each resulting subset. The attribute with the lowest Gini index (i.e., highest purity) is selected as the best attribute for splitting. Unlike information gain, the Gini index does not explicitly consider the information content of attributes but focuses solely on the distribution of class labels.

c: Gain Ratio

The Gain ratio is a modification of the information gain measure that addresses the bias towards attributes with many distinct values. The C4.5 algorithm utilizes the Gain ratio as the attribute selection measure. The formula for calculating the Gain ratio is as follows:

$$Gain\_ratio = \frac{Gain(Attribute)}{Split\_info(Attribute)} \tag{4}$$

d: Chi-Square

It measures the statistical significance of the relationship between the attribute and the class variable. The formula for calculating Chi-square is:

$$\chi^2 = \sum \left(\frac{(O_i - E_i)^2}{E_i} \right) \tag{5}$$

where, Oi is the observed value and Ei is the expected value.

The Chi-Squared statistic follows a Chi-Squared distribution, and its value indicates the strength of association between the attribute and the class variable. The higher the value of chi-square more the sub-nodes are different from the parent node and hence the homogeneity is more. Thus, the attribute with the highest Chi-Squared value is selected as

the best attribute for splitting the data. This test is particularly useful for datasets with categorical attributes and categorical class variables.

e: Relieff

This is an algorithm rather than a mathematical formula such as information gain. Relieff, unlike other greedy algorithms, can detect attribute dependencies. This algorithm uses a sampling-based approach to estimate attribute weights, making it computationally efficient for large datasets. It can handle noisy or incomplete data and is robust against irrelevant attributes. Relieff assigns higher weights to attributes that have consistent values among the nearest neighbors of the same class and different values among the nearest neighbors of different classes. The weights assigned to attributes represent their importance in discriminating between different classes. The main purpose of including Relieff in this section is to mention the use of this algorithm to delete the irrelevant attributes for the classification so that we can obtain an improved decision tree.

4) Decision Tree Induction combinations

The attributes can be discrete-valued or continuous-valued. Likewise, depending on the target, the problem at hand might be classification or regression. The following points clarify the induction process for each of the cases that may arise depending on the dataset chosen:

a: Classification for Discrete-valued attributes

In the context of classification problems in decision trees, categorical attributes and discrete attributes are handled similarly, and thus used interchangeably although they are not exactly the same in meaning. Examples of categorical attributes include "color" (with categories like red, blue, green) or "gender" (with categories like male, female). The best discrete-valued attribute is chosen from the available attributes as the splitting attribute. Then the dataset is partitioned into subsets. The previously chosen attribute need not be considered for any future partitioning, that's why it is removed from the attribute list. The recursive partitioning is carried out for each child node.

b: Classification for Continuous-valued attributes

Examples of continuous (or numeric) attributes include "age," "income," or "temperature." The continuous-valued attributes can be discretized using buckets, intervals, percentiles, or clusters to divide the range of attribute values. Instances falling within each bucket are assigned to the corresponding subset. An impurity measure, such as the Gini index or entropy, for each potential split is calculated to maximize the information gain (minimize impurity). Besides discretizing using buckets, the threshold-based approach can be used to select a threshold value for the attribute. Instances with attribute values greater than or equal to the threshold are assigned to one subset, while instances with values below the threshold are assigned to the other subset. Similarly, the splitting process for each subset created by the split is done recursively, considering only the remaining attributes.

c: Classification for a mixture of Categorical and numeric attributes

The approach is like handling discrete and continuous attributes separately. For numerical attributes, discretization techniques can be applied to convert them into categorical attributes. This can be done through methods such as binning, percentile-based discretization, or clustering-based discretization. Alternatively, binary decisions can be made by selecting a threshold value to separate the data based on whether the attribute value is above or below the threshold. The obtained categorical data is partitioned based on the attribute values, and the process is repeated recursively for each partition.

d: Regression for numeric data

Numerical data for regression refers to the case where the attribute values are numeric, and the target variable is also numeric. Both the attribute and the target are continuous variables. For the regression problem, we can't use the splitting criteria like calculating entropy like we do in classification problems. Thus, we use an appropriate attribute selection measure, such as mean squared error or mean absolute error, to determine the best attribute for splitting the data.

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y}_i)^2 \tag{6}$$

For the regression decision tree, we try to minimize the MSE (Mean Square Error) instead of entropy as we do in the Classification trees. For numerical attributes, binary decisions are made based on a threshold value to determine the direction of the split. The attribute values are compared to the threshold, and the data is partitioned accordingly. Splitting of the data is based on the selected attribute and continued recursively until a stopping condition is met or a desired level of accuracy is achieved. Once the decision tree is built, it can be used to make predictions on new instances by traversing the tree and assigning the predicted numeric value based on the path followed. To reduce the risk of overfitting, we can specify the minimum number of children per leaf node in advance and use cross-validation to find this value.

5) Tree Pruning

Tree pruning is a technique used in decision tree algorithms to reduce the complexity and overfitting of the tree model. Overfitting usually occurs when a decision tree captures the noise and irregularities in the training data, leading to poor generalization on unseen data. The goal of tree pruning is to find the right balance between complexity and accuracy by removing unnecessary branches and nodes from the tree. Two common approaches to tree pruning include: pre-pruning and post-pruning. Pre-pruning is a pruning method that stops the tree growth early based on predefined criteria, such as

maximum depth or minimum number of instances in a leaf. Post-pruning, also known as error-based pruning, involves growing the tree to its entirety and then removing branches based on their estimated error rate. Another could be Rule Post-Pruning, where the decision tree is converted into a set of if-then rules and their accuracy is evaluated on the validation data. Rules that do not contribute significantly to the model's performance are pruned. CART uses Cost complexity pruning, also called minimal cost complexity pruning or alpha-beta pruning, employing a parameter called the complexity parameter to control the pruning process. The pruning process involves evaluating the error rate or other metrics on a validation set or using cross-validation to estimate the performance of the pruned tree. Pruning reduces the risk of overfitting by simplifying the decision tree model and improving its generalization on unseen data. Thus, Tree pruning is an essential technique in decision tree learning, contributing to improved accuracy, reduced complexity, and enhanced model interpretability.

B. SYSTEM BLOCK DIAGRAM

The system architecture in Figure 1 demonstrates the sequence of blocks involved in the decision tree analysis. Initially, diverse datasets were collected, and data visualization techniques such as scatter plots, histograms, and heatmaps were utilized to gain insights into the data's distribution and relationships. For decision tree analysis, the preprocessing stage involves steps such as handling missing values, normalizing numerical features, and encoding categorical variables. This ensures that the data is brought to a common scale, enabling all features to contribute equally to the analysis. The dataset is split into training and testing sets, and the decision tree algorithm is trained using the training set. Hyperparameters are optimized, and the algorithm's performance is evaluated using metrics such as accuracy, precision, recall, and F1-score. Once the decision tree model is trained, it can be visualized to gain insights into the significant attributes and decision paths that contribute to the classification process. The visualization provides a clear representation of the decision tree structure, aiding in understanding the model's behavior and interpreting its results. In summary, the decision tree analysis follows a similar sequence to the architecture in Figure 1, but with distinct steps focusing on feature selection, model training, hyperparameter optimization, and visualization of the decision tree structure.

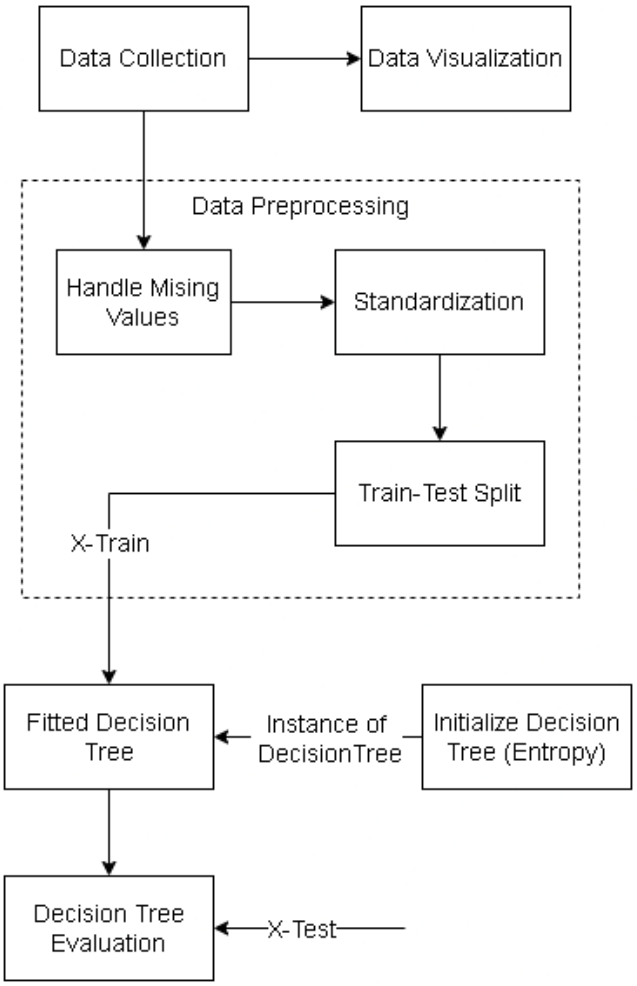


FIGURE 1: System Architecture of Decision Tree

C. INSTRUMENTATION

The implementation of Decision Tree in our study involved the use of several libraries and tools to facilitate data visualization and mathematical operations.

1) Pandas

Pandas, a powerful Python library for data manipulation, plays a crucial role in the decision tree pipeline. It provides essential functions and data structures for data handling, exploration, and analysis. Pandas is used to load and read data, inspect its structure, handle missing values, encode categorical variables, scale numerical data, and perform descriptive statistics. It enables efficient data pre-processing and feature extraction, making it an invaluable tool for decision tree analysis.

2) Matplotlib

Matplotlib is a powerful data visualization library in Python. It contributes to the instrumentation of decision tree models by offering essential plotting capabilities. While it doesn't directly construct decision trees, matplotlib enables the visualization of decision tree structures as hierarchical diagrams or tree plots, aiding in understanding the decision paths. Additionally, matplotlib facilitates the creation of feature importance plots, allowing users to identify influential attributes. Moreover, it enables the visualization of decision boundaries, aiding in comprehending how the model classifies different regions of the feature space. By leveraging matplotlib's visualization capabilities, users can gain valuable insights into decision tree models and effectively communicate their findings.

3) NumPy

NumPy, also known as Numerical Python, is essential in the implementation of decision tree models, offering efficient array operations and mathematical functions. With its multidimensional array object and comprehensive set of functions, NumPy enables high-performance computations, including mathematical operations, shape manipulation, and linear algebra operations. In the context of decision tree analysis, NumPy is instrumental in performing array operations such as element-wise calculations, matrix multiplication, transposition, and the computation of eigenvalues and eigenvectors. Key NumPy functions like np.dot(), np.transpose(), and np.linalg.eig() are utilized for these tasks. Specifically, np.linalg.eig() is particularly valuable in decision tree analysis as it determines the direction and magnitude of components crucial for constructing the decision tree. By leveraging NumPy's capabilities, the implementation of decision tree models becomes more efficient, allowing for seamless manipulation and analysis of arrays required for accurate classification.

4) Seaborn

Seaborn, a Python data visualization library, enriches the analysis and interpretation of decision tree models by providing powerful visualization capabilities. While not involved in constructing decision trees directly, seaborn's plotting functions allow for visualizing the decision tree structure through tree plots and dendrograms, aiding in understanding the decision paths. Seaborn further facilitates the visualization of feature importance, enabling the identification of influential attributes and their impact on classification. Additionally, seaborn offers a variety of plotting options, such as pair plots, scatter plots, and heatmaps, which assist in exploring feature relationships and their effects on the target variable. By leveraging seaborn's visualization capabilities, users can gain deeper insights into decision tree models, effectively communicate their findings, and enhance the overall analysis and interpretation process.

5) Dtreeviz

Dtreeviz is a Python library that enhances decision tree analysis through interactive and visually appealing visualizations. It allows users to explore the decision tree structure interactively, gaining insights into the decision paths and splits. Dtreeviz provides feature importance plots to identify influential features and offers other visualizations such as class prediction histograms and sample-based explanations. By leveraging dtreeviz, users can gain a deeper understanding of decision tree models, improving interpretation and decision-making.

By utilizing these libraries, we were able to seamlessly integrate data visualization and mathematical operations into our Decision Tree Analysis. These libraries enabled effective data visualization, allowing us to analyse datasets, visualize decision tree structures, and gain valuable insights. By leveraging these libraries, we combined data analysis and visualization to improve decision tree analysis and interpretation.

D. WORKING PRINCIPLE

The decision tree Analysis consists a the pipeline of steps involving data collection, preprocessing, training the algorithm, hyperparameter optimization, performance evaluation, and decision tree visualization. The dataset is collected, preprocessed to handle missing values and normalize features, and split into training and testing sets. The decision tree algorithm is trained using the training set, creating a structure that recursively splits the data based on informative attributes. Hyperparameters are optimized to enhance the model's performance. The model's accuracy, precision, recall, and F1-score are evaluated using the testing set. Decision tree visualization provides insights into significant features and decision paths. Overall, this pipeline enables accurate classification of dry bean varieties based on their physical properties and shape characteristics, benefiting applications in agriculture and the food industry.

1) Data Pre-processing

The input dataset is pre-processed to ensure its suitability for decision tree analysis. This includes handling missing values, scaling the data, and addressing outliers. In our implementation, specific data pre-processing techniques were applied to each dataset, such as standardization or normalization. Categorical target variables were encoded to convert them into a numerical format. For instance, label encoding was used to represent the seven classes (e.g., 'SEKER', 'BARBUNYA', 'BOMBAY', 'CALI', 'HOROZ', 'SIRA', 'DERMASON') with numeric values. The processed dataset can be visualized using the provided figures. This pre-processing step is essential to ensure that the decision tree algorithm can effectively analyse the data and make accurate classifications based on the attributes. By pre-processing the data, we address any inconsistencies or anomalies that could affect the performance of the decision tree model. The visualization of the processed dataset provides insights into the transformed data and allows for a better understanding of its distribution and relationships. Overall, data pre-processing is a crucial step in preparing the input dataset for decision tree analysis, ensuring that it meets the necessary requirements for accurate classification.

2) Train-Test Split

After pre-processing, the dataset is partitioned into two subsets: a training set and a testing set. The training set is utilized to train the decision tree algorithm. By leveraging the provided features and their associated labels, the algorithm learns to construct a decision tree structure. This structure is formed through a recursive process of splitting the data based on informative attributes, creating branches and leaf nodes that represent different classification outcomes. The decision tree algorithm learns from the training set to capture patterns and relationships between the features and labels, enabling it to make informed decisions during classification.

3) Model Initialization

The initialization of a decision tree model involves setting up the parameters and configurations necessary to create the decision tree structure. This includes specifying the splitting criterion (e.g., entropy or Gini index), setting the maximum depth of the tree, and defining other hyperparameters like the minimum number of samples required for a node split. By initializing the decision tree model with appropriate parameters, we lay the foundation for the subsequent training and optimization processes.

4) Hyperparameter tuning

Hyperparameter tuning optimizes the decision tree model by fine-tuning parameters like tree depth, minimum sample split, and splitting criterion. Techniques like cross-validation determine the optimal parameter values. This iterative process aims to strike a balance between model complexity and generalization ability, ensuring accurate predictions on unseen data. Through hyperparameter optimization, the decision tree model achieves enhanced performance and optimal results.

5) Model Evaluation

The performance of the decision tree model is assessed using the testing set, where multiple metrics, including accuracy, precision, recall, and F1-score, are computed to evaluate its classification capabilities. These metrics provide insights into the model's accuracy in correctly predicting the class labels of the dry bean varieties. By analysing these performance measures, we can assess the effectiveness of the decision tree model in accurately classifying the dry bean varieties based on their attributes.

6) Model Visualization

Visualizing the decision tree structure allows to gain valuable insights into the important features and decision paths within the model. By employing visualization techniques, we can represent the decision tree's decision-making process graphically, enabling easier interpretation and comprehension of the key factors contributing to the classification outcomes. These visual representations aid in understanding how the decision tree makes decisions based on different attributes, shedding light on the underlying logic and helping us identify the most influential features in the classification process.

To summarize, the decision tree pipeline involves collecting the data, pre-processing it, training the algorithm, optimizing the hyperparameters, evaluating the model's performance, and visualizing the decision tree. This comprehensive process enables the accurate classification of dry bean varieties by considering their physical properties and shape characteristics. The application of this pipeline holds significant potential in agricultural and food industry domains, providing valuable insights for quality control and differentiation purposes. By following these steps, the decision tree analysis becomes a powerful tool in efficiently and effectively classifying dry bean varieties, contributing to advancements in the agricultural sector.

III. RESULTS

A. DISTRIBUTION OF DATA

The distribution of the dataset feature was analysed using a histogram plot. The histogram provided insights into the frequency distribution of the feature values, allowing us to understand the underlying distribution pattern and identify any potential outliers or data skewness.

B. PRE-PROCESSING

The original dataset contained the class labels as 'SEKER', 'BARBUNYA', 'BOMBAY', 'CALI', 'HOROZ', 'SIRA', 'DERMASON' class. As part of the pre-processing phase, label encoding was applied to transform categorical data into numerical values. These labels were encoded using a range from 0 to 6, assigning a unique numerical representation to each category. This transformation enabled us to effectively utilize the categorical feature in subsequent analysis or modelling tasks, as many machine learning algorithms require numerical input.

C. MODEL PERFORMANCE

1) General model performance

This is the initial performance of the model without doing any processing or hyperparameter tuning. The general decision tree model achieved an accuracy of 0.888. The classification report showed precision, recall, and F1-score values for each class, along with the support count. The macro-average of 0.9 and the weighted-average scores of 0.89 indicate the overall performance of the model. Similarly, the confusion matrix shown is useful in visualizing the performance of the classification model for given set of test data.

[insert classification report figure here]

[insert confusion matrix report figure here]

2) Model performance after PCA

The model is expected to perform better after the application of PCA (Principal Component Analysis). It is observed that after applying PCA with 7 components covering 99.60

[insert Feature importance (having both cumulative and individual importance curve) figure here]

[insert scree plot here]

Likewise, the classification report for this model shows a macro average of 0.91 and a weighted average of 0.90. The confusion matrix shown can further help in visualizing the performance of this model for given test data.

[insert classification report here]

[insert confusion matrix here]

3) Model performance after Pre-Pruning

Similarly, the decision tree model's performance was evaluated after applying pre-pruning by specifying the maximum depth of the tree. The accuracy, precision, recall, and F1-score values reflect the model's performance after pre-pruning. When a maximum depth of 3 was chosen, the model gave an accuracy of 83.54%. It was also seen that the macro

average was 0.73 and the weighted average was 0.82. The confusion matrix for this model shows a different kind of performance in the test data.

[insert classification report/ confusion matrix figure here]

Another pre-pruning technique of minimum sample split was done as well to evaluate the model performance. Here we specified the minimum number of samples required to do a spilt. Keeping the criteria as Entropy and a minimum sample split of 1000 gave an accuracy of 87.64%. Likewise, the macro average and weighted average of 0.89 and 0.88 was obtained respectively. It is further explained from the obtained classification report and confusion matrix.

[insert report/confusion matrix]

4) Model performance after Post-Pruning

Likewise, the decision tree model's performance was evaluated after applying post-pruning techniques, such as Grid SearchCV and Cost Complexity pruning path. GridSearchCV is a technique for finding the optimal parameter values from a given set of parameters in a grid., which is essentially a cross-validation technique. The best parameters were obtained which showed that the optimal maximum depth was 5 and the optimal criterion was log-loss. Doing so gave an accuracy of 85.90

[insert report/confusion matrix here]

Likewise, another post-pruning technique called Cost Complexity Pruning Path helps in finding the optimal pruning parameter by exploring the trade-off between tree complexity and accuracy. The accuracy, precision, recall, and F1-score values reflect the improved performance of the model after Cost Complexity Pruning Path.

[insert effective alpha and impurity figure here]

IV. DISCUSSION AND ANALYSIS

The dry bean dataset contains a collection of 13611 instances belonging to 7 classes of beans. The data were normally distributed with respect to each feature, and some were skewed to the left or right, as seen in the histogram. The successful application of the Decision Tree Classifier to the dry bean dataset demonstrated its effectiveness in accurately classifying the different varieties of dry beans. By constructing a hierarchical structure of decision rules based on the dataset's features, the decision tree model achieved distinct separations between the dry bean varieties, providing reliable class predictions.

A noteworthy finding from the analysis is the impact of the decision tree's depth on its performance. Deeper trees generally exhibit higher accuracy in classifying the dry bean varieties. However, it is crucial to find the right balance to avoid overfitting, where the model captures irrelevant or noisy features. Therefore, selecting an optimal depth for the decision tree is essential to achieve optimal performance without sacrificing generalization. Methods like pruning, Increasing the minimum sample split, fixed depth, etc can be used to limit the depth of the Decision Tree.

The change in the decision tree model's performance after Grid SearchCV post-pruning can be attributed to the optimization of hyperparameters. By fine-tuning the pruning parameters and exploring different combinations of hyperparameters to find the best configuration Grid SearchCV helps to reduce overfitting and enhance the generalization capabilities of the model. Likewise, the Cost Complexity Pruning Path allows for the identification of the optimal pruning parameter by striking a balance between simplicity and accuracy. Overall, both Grid SearchCV and Cost Complexity Pruning Path techniques have shown to be effective in improving the performance of the decision tree model through appropriate post-pruning. The various curve in relation to alpha showed different insights, where alpha refers to the pruning parameter used in the decision tree pruning process. It determines the level of pruning applied to the tree. Initially, as alpha increases, the accuracy of the model shows a gradual improvement. However, after a certain threshold, further increasing alpha leads to a plateau in accuracy, indicating that additional pruning has a limited impact on enhancing model performance. Similarly, as alpha increases, the depth of the tree decreases sharply, resulting in a simpler and less complex model. However, beyond a certain alpha value, the depth curve levels off, indicating that additional pruning has minimal impact on reducing the tree's depth. Also, as alpha increases initially, the total impurity sharply increases, indicating a reduction in model performance and an increase in impurity. However, beyond a certain alpha value, the impurity curve levels off, suggesting that further pruning does not significantly affect the overall impurity. Thus, a higher alpha value results in more aggressive pruning, leading to a simpler model with potentially reduced overfitting. The curves show the effect of different alpha values on various aspects of the decision tree model, providing insights into the trade-off between complexity and performance.

Furthermore, the Decision Tree Classifier allows for the assessment of feature importance. By examining the top-splitting nodes and their associated features, we can identify the most significant attributes in classifying the dry bean varieties. This insight provides valuable information for further analysis and understanding of the factors influencing the classification process. Its hierarchical decision-making structure simplifies the classification process, while the selection of an appropriate tree depth and consideration of feature importance contribute to achieving accurate and interpretable results. By employing pre-pruning and post-pruning techniques, we have successfully optimized the model's complexity and achieved a balance between accuracy and generalization.

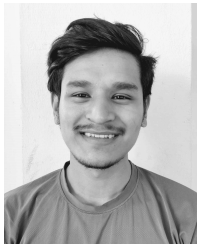
V. CONCLUSION

In conclusion, the implementation of the decision tree algorithm on the dry bean dataset has provided us with valuable insights into its classification capabilities. The objective of implementing the decision tree algorithm using Information gain as the attribute selection measure on the dry bean dataset

was successfully achieved. By utilizing the decision tree classifier with the entropy criterion and a maximum depth of five, we obtained a commendable accuracy score of 89.36% on the test data. The decision tree algorithm demonstrated its ability to effectively handle the classification task, enabling it to identify the most informative features for classification, and therefore showcasing its power in distinguishing between different classes based on the dataset's attributes. The implementation of pre-pruning techniques, such as limiting the maximum depth, helped prevent overfitting and ensured the model's generalization capability. Thus, Decision tree algorithms offer a powerful and interpretable approach to solving classification and regression problems by creating a hierarchical structure of decision rules based on input features. With their ability to handle both discrete and continuous attributes, decision trees can provide flexibility in modeling a wide range of datasets and can effectively capture complex patterns and relationships. With further exploration and fine-tuning of hyperparameters, decision tree models hold great potential for solving a wide range of classification problems in various domains.

REFERENCES

[1] Abdi, H., & Williams, L. J. (2010). Principal component analysis. Wiley interdisciplinary reviews: computational statistics, 2(4), 433-459. doi:10.1002/wics.101.
[2] I. T. Jolliffe, "Principal component analysis and factor analysis," Principal Component Analysis, pp. 115-128, 1986.
[3] K. Pearson, "On lines and planes of closest fit to systems of points in space," Philosophical Magazine, vol. 2, no. 11, pp. 559-572, 1901.
[4] H. Abdi and L. J. Williams, "Principal component analysis," Wiley Interdisciplinary Reviews: Computational Statistics, vol. 2, no. 4, pp. 433-459, 2010.
[5] M. Ringnér, "What is principal component analysis?," Nature Biotechnology, vol. 26, no. 3, pp. 303-304, 2008.
[6] scikit-learn developers, "sklearn.decomposition.PCA," scikit-learn documentation, [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html.
[7] J. Brownlee, "How to Use Principal Component Analysis for Feature Selection," Machine Learning Mastery, [Online]. Available: https://machinelearningmastery.com/principal-components-analysis-for-feature-selection-in-python/.
[8] D. Pattnaik, "Principal Component Analysis (PCA) - Step by Step Tutorial," Towards Data Science, [Online]. Available: https://towardsdatascience.com/principal-component-analysis-pca-step-by-step-tutorial-81b6387c55c1.
[9] A. Dey, "PCA — Machine Learning Mystery," Machine Learning Mystery, [Online]. Available: https://machinelearningmastery.com/pca-machine-learning-mystery/.
[10] A. Géron, "Principal Component Analysis (PCA) Explained," Medium, [Online]. Available: https://towardsdatascience.com/principal-component-analysis-pca-explained-and-implemented-eeab7cb73b72.



PRABIN BOHARA is an enthusiastic individual currently pursuing a Bachelor's degree in Computer Engineering, set to graduate in 2024. He has a genuine passion for coding, web technologies, and machine learning. Prabin's primary focus revolves around AI research and exploring the various applications of data science in general. He actively contributes to open-source projects and continuously expands his knowledge through academic endeavors. Prabin actively participates in webinars, both nationally and internationally, as well as competing in diverse technology competitions. He also shares his expertise by conducting workshops on various technology topics. With a strong drive for excellence, Prabin aims to make valuable contributions in the field of AI while keeping up with the latest advancements.

PRABIN SHARMA POUDEL , an undergraduate senior at IOE's Thapathali Campus, has forever been in pursuit of knowledge and wisdom. His involvement extends beyond technical confines, with involvement in non-technical organizations like AYON. With a vision to become a scholarly figure, he aims to bridge multiple disciplines of knowledge. His current field of interest encompasses AI in conflict resolution, Computer vision, and AI in creative industries Beyond academics, Prabin finds joy in playing chess, composing music, and journaling his thoughts through poetry.

SANTOSH PANDEY is a fourth-year Bachelor's student at IOE, Thapathali Campus, with a passion for coding and technology. Currently working as an intern at NAAMII organization, he is driven by the goal of applying his skills to earn a living through coding. With a keen interest in full-stack development, graphic designing, mobile app development, and computer vision, Santosh is constantly exploring new technologies and honing his skills in these areas. In his free time, he actively participates in coding competitions and enjoys participating in marathons. Santosh's dedication, enthusiasm, and continuous learning make him a promising professional in the field of technology and coding.

CODE

A. RANDOM NUMBERS DATASET

```
import numpy as np
import matplotlib.pyplot as plt
from numpy import random

# Generates and visualizes a scatter plot of two sets of 50 random data points each. [randn=from normal distribution.]
# x1 and x2 represent the 2 axes
x1 = np.random.randn(50)
x2 = np.random.randn(50)
print(x1)
print(x2)
plt.scatter(x1,x2)
plt.xlabel("X1")
plt.ylabel("X2")
plt.text(-2, 2.8, "'THA076BCT026','THA076BCT027','THA076BCT041'", fontsize=8,color='red')
plt.title('Scatter Plot for Random Values')
plt.ylim(-3,3)
plt.xlim(-3,3)
plt.savefig('Scatter Plot for Random Values', bbox_inches='tight')

# Mean and standard deviation
mean1 = np.mean(x1)
mean2 = np.mean(x2)
sd1 = np.std(x1)
sd2 = np.std(x2)
print(mean1,sd1)
print(mean2,sd2)

# 2x2 random matrix with elements sampled from a uniform distribution (range [0, 1]).
random_matrix = np.random.rand(2,2)
print(random_matrix)

# Converts the lists x1 and x2 into NumPy arrays, then creates a data matrix by column stacking the arrays.
x1_array = np.array(x1)
x2_array = np.array(x2)
data_matrix = np.column_stack((x1_array, x2_array))
print(data_matrix.shape)
print(data_matrix)


# **STEP 1: Reshape the data by multiplication with a random matrix .** To visualize the linear transformation, click on the link : https://www.geogebra.org/m/YCZa8TAH?fbclid=IwAR3LK2VOQooxb-qzH-LLhlyzVEhi8Qsb8Of_V1zeiqWb3FcjYibVPbL0NGo

transformed_data_by_random_matrix = data_matrix @ random_matrix
print(transformed_data_by_random_matrix.shape)

# first column [:,0] as x-axis value. similarly 2nd column as y-axis
plt.scatter(
    transformed_data_by_random_matrix[:,0],
    transformed_data_by_random_matrix[:,1])
plt.xlabel("X1-transformed")
plt.ylabel("X2-transformed")
plt.ylim(-3,3)
plt.xlim(-3,3)
plt.text(-1.5, 2.8, "'THA076BCT026','THA076BCT027','THA076BCT041'", fontsize=8,color='red')
plt.title('Scatter Plot for transformed Random Values')
plt.savefig('transformed Random Values', bbox_inches='tight')


# **STEP 2: Calculate covariance matrix** The covariance matrix provides insights into the relationships between different dimensions of the transformed_data_by_random_matrix.
# See if elements in main diagonal are max and other diagonal are min. If not, we have to proceed further .
covariance_matrix = np.cov(transformed_data_by_random_matrix.T)
covariance_matrix
#same as above
covariance_matrix = np.cov(np.transpose(transformed_data_by_random_matrix))
print(covariance_matrix)


# **STEP 3: Eigen values and vector calculation**
# The eigenvalues represent the variance explained by each principal component (magnitude of new feature space)
# The eigenvectors define the directions of the principal components. ( direction of new feature space)
eigen_values, eigen_vectors = np.linalg.eig(covariance_matrix)
print(eigen_values)
print(eigen_vectors)

# Just To check if the eigenvalues really represent the variance, lets find out the proportion of variance for each components
proportion1 = eigen_values[0] / np.sum(eigen_values)
print(proportion1*100)
proportion2 = eigen_values[1] / np.sum(eigen_values)
print(proportion2*100)


# **STEP 4: Transform the data with the selected eigen vector.**
#first write the eigenvectors row wise ( do the transpose, from column wise to row wise)
```

```
arranged_eigen_vector = np.transpose(
    eigen_vectors)
print(arranged_eigen_vector.shape)
print(arranged_eigen_vector)

#also transpose the (50,2) data matrix to
     (2,50)
transposed_data_matrix =
    transformed_data_by_random_matrix.T
print(transposed_data_matrix.shape)

#Project data into first principal
    component (by matrix multiplication)
new_data = np.dot(arranged_eigen_vector,
    transposed_data_matrix)
print(new_data.shape)
#print(new_data)

#transpose back to (50,2)
trans_new_data=new_data.T
print(trans_new_data.shape)

# again see if the elements in main
    diagonal are max
covariance_of_new_data = np.cov(new_data)
print(covariance_of_new_data)

#lets select the best eigen vectors as
    specified in the step 4
best_eigen_vector = np.transpose(
    eigen_vectors[:,0])
print(best_eigen_vector.shape)
print(best_eigen_vector)
new_new_data = np.dot(best_eigen_vector,
    transposed_data_matrix)
print(new_new_data.shape)
print(new_new_data)


# **STEP 6: plot the new data**
plt.scatter(new_data[0,:], new_data[1,:]
    )
plt.ylim(-3,3)
zeros = np.zeros(50)
plt.scatter(new_data[1,:],zeros)
plt.xlabel("X1_after_PCA")
plt.ylabel("X2_after_PCA")
plt.title('Scatter Plot After PCA')
plt.text(-0.8, 0.9, "'THA076BCT026','
    THA076BCT027','THA076BCT041'",
    fontsize=8,color='red')
plt.ylim(-1,1)
plt.xlim(-1,1)
plt.grid()
plt.savefig('scatter plot after PCA',
    bbox_inches='tight')

print(new_data.shape)
print(new_data)
```

B. IRIS DATASET

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
from numpy import random
import sklearn


# **STEP 1: Load the iris flower dataset
    **

from sklearn.datasets import load_iris
iris = load_iris()

X = iris.data    # X is 2d array, where
    row=samples, column=features
y = iris.target  # y is 1d array,
    representing class label for all
    samples
print(X.shape)
print(y.shape)
print(iris.feature_names)
print("Target names:", iris.target_names)
print(X)

# **STEP 2: Create a dataframe and see
    the data statistics**

# assign meaningful columns(feature_names
    ) to feature values for 150 samples(
    iris.data) in the dataframe.
df = pd.DataFrame(data=iris.data,columns=
    iris.feature_names)
# add a new target column.
df['target']=iris.target
df.head(2)   # view 1st 2 data, both are
    setosa(numbering starts from 0)

#view 49th to 52nd dataset to see 2
    classes(setosa and vesicolor)
subset = df.iloc[48:52]
print(subset)

# statistics computed for each column
summary_stats = df.describe()
print(summary_stats)

# data visualization, box plot for each
    feature
df.plot(kind='box', figsize=(10, 6))
plt.title('Box Plot of Iris Dataset')
plt.text(1.5, 8, "'THA076BCT026','
    THA076BCT027','THA076BCT041'",
    fontsize=8,color='red')

plt.savefig('Box Plot of Iris Dataset',
    bbox_inches='tight')
plt.show()

import seaborn as sns
# Set the size of the figure
plt.figure(figsize=(12, 7))
df['target'] = df['target'].replace({0: '
    Setosa', 1: 'Versicolor', 2: '
    Virginica'})
# Create a pair plot with hue based on
    the 'target' column in the dataframe
g=sns.pairplot(df, hue='target',plot_kws
    ={'legend': False})
g._legend.set_title('')
# # Set the title of the plot
plt.suptitle('Attribute Correlation Pair
    Plot', fontsize=16, y=0.98)
```

```
# Set the style and color of the text box
textbox_props = dict(boxstyle='round', facecolor='white', edgecolor='gray', alpha=0.8)

# Add a text box with the highlighted feature names
plt.text(0.1, 8, "THA076BCT026\nTHA076BCT027\nTHA076BCT041", fontsize=10, color='red', ha='center', bbox=textbox_props)
legend_elements = [plt.Line2D([0], [0], marker='o', color='w', label='Setosa', markerfacecolor='royalblue', markersize=8),
                   plt.Line2D([0], [0], marker='o', color='w', label='Versicolor', markerfacecolor='g', markersize=8),
                   plt.Line2D([0], [0], marker='o', color='w', label='Virginica', markerfacecolor='orange', markersize=8)]

# Add the legend to the plot
plt.legend(handles=legend_elements, title='target')

# Adjust the spacing between subplots
plt.subplots_adjust(top=0.9)
# Set custom tick labels for the legend
# Save the plot to a file
plt.savefig('./plots/pair_plot.png', bbox_inches='tight')

# Display the plot
plt.show()

# Visualization of data points based on the target class (purple=setosa,green=Versicolour,yellow=Virginica )
df['target']=iris.target
plt.figure(figsize=(10, 6))
plt.scatter(df['sepal length (cm)'], df['sepal width (cm)'], c=df['target'], label='Sepal')
plt.scatter(df['petal length (cm)'], df['petal width (cm)'], c=df['target'], marker='x', label='Petal')
plt.xlabel('Length (cm)')
plt.ylabel('Width (cm)')
plt.title('Scatter Plot of Sepal and Petal Features')
plt.colorbar(label='Species')
plt.text(1, 4.3, "'THA076BCT026','THA076BCT027','THA076BCT041'", fontsize=8,color='red')
plt.legend()
plt.savefig('./plots/Sepal and Petal Features', bbox_inches='tight')
plt.show()

# **STEP 3: Data preprocessing**

# Handling missing values:
# df.isna() returns a boolean, where True means missing values(NaN) and .sum() sums the values along each column
missing_values = df.isna().sum()
print("Columns with missing values:")
print(missing_values[missing_values > 0])

# Standardizing to eliminate the chance of PCA being influenced by magnitude of some features' values.
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
X = df.iloc[:,1:].values
df_scaled = pd.DataFrame(scaler.fit_transform(df), columns=df.columns)

# IMPORTANT STEP!
# Create new input and output after preprocessing X and y.
X_train = df_scaled.drop('target', axis=1).values
y_train = df['target']
print("shape of new input X",X_train.shape)
print("shape of new output Y",y_train.shape)
print(X_train)
print(y_train)

df_plot=df_scaled.copy()
df_plot['target'] = df['target'].replace({0: 'Setosa', 1: 'Versicolor', 2: 'Virginica'})
plt.figure(figsize=(12, 8))
# Create a pair plot with hue based on the 'target' column in the dataframe
g=sns.pairplot(df_plot, hue='target', plot_kws={'legend': False})
g._legend.set_title('')

plt.suptitle('Attribute Correlation Pair Plot of Transformed Data', fontsize=16, y=0.98)

# Set the style and color of the text box
textbox_props = dict(boxstyle='round', facecolor='white', edgecolor='gray', alpha=0.8)

# Add a text box with the highlighted feature names
plt.text(1.5, 6, "THA076BCT026\nTHA076BCT027\nTHA076BCT041", fontsize=10, color='red', ha='center', bbox=textbox_props)

# Adjust the spacing between subplots
plt.subplots_adjust(top=0.9)

legend_elements = [plt.Line2D([0], [0], marker='o', color='w', label='Setosa', markerfacecolor='royalblue',
```

```
    markersize=8),
                    plt.Line2D([0], [0],
                        marker='o', color=
                        'w', label='
                        Versicolor',
                        markerfacecolor='g
                        ', markersize=8),
                    plt.Line2D([0], [0],
                        marker='o', color=
                        'w', label='
                        Virginica',
                        markerfacecolor='
                        orange',
                        markersize=8)]

# Add the legend to the plot
plt.legend(handles=legend_elements, title
    ='target')
# Save the plot to a file
plt.savefig('./plots/
    pair_plot_transformed.png',
    bbox_inches='tight')

# Display the plot
plt.show()

# lets see the new plot after
    standardization
plt.figure(figsize=(10, 6))
plt.scatter(df_scaled['sepal length (cm)'
    ], df_scaled['sepal width (cm)'], c=
    df_scaled['target'], label='Sepal')
plt.scatter(df_scaled['petal length (cm)'
    ], df_scaled['petal width (cm)'], c=
    df_scaled['target'], marker='x',
    label='Petal')
plt.xlabel('Length (cm)')
plt.ylabel('Width (cm)')
plt.title('Scatter Plot of Sepal and
    Petal Features')
plt.colorbar(label='Species')
plt.ylim(-3,4)
plt.xlim(-3,3)
plt.text(-1, 3.5, "'THA076BCT026','
    THA076BCT027','THA076BCT041'",
    fontsize=8,color='red')
plt.legend()
plt.savefig('./plots/processed Sepal and
    Petal Feature', bbox_inches='tight')
plt.show()

# **STEP 4: Covariance matrix calculation
    .**

# X_train is already an array so no need
    to do data_matrix = np.array(X)
data_matrix = X_train
data_matrix.shape

# covariance matrix will be 4*4 since x
    =(150,4)
covariance_matrix = np.cov(X_train,
    rowvar=False)              # row of
    X_train != variable(so column of
    X_train means variable, and rows
    means observation)
print("Covariance matrix shape:",
    covariance_matrix.shape)
print(covariance_matrix)

# visualize covariance matrix
plt.imshow(covariance_matrix, cmap='Blues
    ', interpolation='nearest')
plt.colorbar()
plt.title('Covariance Matrix')
plt.text(0, -0.8, "'THA076BCT026','
    THA076BCT027','THA076BCT041'",
    fontsize=8,color='red')
plt.xticks(np.arange(4), np.arange(4))
    #tick labels on axes changed from 0
    to 4
plt.yticks(np.arange(4), np.arange(4))
plt.savefig('Iris Covariance Matrix',
    bbox_inches='tight')
plt.show()

# **STEP 5: Eigen values and eigen vector
     calculation**

eigenvalues, eigenvectors = np.linalg.eig
    (covariance_matrix)
#descending sort
sorted_indices = np.argsort(eigenvalues)
    [::-1]
eigenvalues = eigenvalues[sorted_indices]
eigenvectors = eigenvectors[:,
    sorted_indices]
print("eigen values are",eigenvalues)
print("eigen vectors are",eigenvectors)
print(eigenvalues.shape)
print(eigenvectors.shape)

# **STEP 6: See the variance explained by
     each eigen values.**

# Proportion of variance for each eigen
    values
pov_list=[]
for i in range (0,4):
  pov=eigenvalues[i]/sum(eigenvalues)
  pov_list.append(pov)
  print("Proportion of variance for",
      eigenvalues[i],"eigen value is",
      pov)

# Plot the scree plot
plt.plot(range(1, len(pov_list) + 1),
    pov_list, marker='o',color='blue')
varlegend= "'THA076BCT026','THA076BCT027
    ','THA076BCT041'"
plt.text(2.5,0.7,varlegend,color='red')
      #plt.text(x_position, y_position,
    varlegend)
plt.xlabel('Principal Component')
plt.ylabel('Proportion of Variance')
plt.title('Scree Plot')
plt.show()


# **STEP 7: Select the eigen vectors as
    needed to compute the final output**

print("Shape of x train is",X_train.shape
    )
```

```
transposed_X_train = np.transpose(X_train)
transposed_X_train.shape
eigenvectors_transposed=np.transpose(eigenvectors)

final_data = {}        # Dictionary to store the final projected data

# Define the combinations of selected components
selected_components = [[0], [0, 1], [0,2], [0, 3], [1], [1,2], [1,3], [0, 1, 2, 3]]
count=len(selected_components)

# Iterate over the selected components
for i in range (0,count):
  selected_eigenvectors = eigenvectors_transposed[selected_components[i]]
                  #select the combination of components from above list
  final_data[i] = np.transpose(selected_eigenvectors @ transposed_X_train)  #projection
  print(final_data[i].shape)

eigenvectors_transposed[selected_components[2]]

# **STEP 8: Computing and visualizing covariance for the new data.**

new_covariance_matrices_dictionary = {}
# Compute covariance matrix for each projected data
for i in range(0, count):
    print(" * Covariance matrix for combination no", i)
    new_covariance_matrix = np.cov(final_data[i], rowvar=False)
    new_covariance_matrices_dictionary[i] = new_covariance_matrix
    # Print the shape of the covariance matrix
    print("  --->Covariance matrix shape: ", new_covariance_matrix.shape)

#Lets access the first combination(combination 0), which is (eigen[0])                        (best component)
new_covariance_matrices_dictionary[0]

#Lets access combination 1, which is eigen[0] and eigen[1]                (2 best components)
new_covariance_matrices_dictionary[1]

#Lets access combination 3, which is eigen[0] and eigen[3]                (best and worst component)
new_covariance_matrices_dictionary[3]

#Lets access combination 5, which is eigen[1] and eigen[2]        (2 medium medium best components)
new_covariance_matrices_dictionary[5]

#Lets access combination 7, which is eigen[0],eigen[1],eigen[2] and eigen[3]      (all components combined)
new_covariance_matrices_dictionary[7]

# Heatmap plot, darker shade indicate higher values. (combination starts from 0 to 7)
plt.imshow(new_covariance_matrices_dictionary[1], cmap='Reds', interpolation='nearest')  #for smoother interpolation of color on pixel values, instead of 'nearest', we can use 'bilinear','bicubic','spline16', 'spline36', 'hanning', 'hamming', 'hermite',etc..
plt.colorbar()
plt.title("New Covariance Matrix Combination 1 (2 best PC)")
plt.text(-0.3, -0.66, "'THA076BCT026','THA076BCT027','THA076BCT041'", fontsize=8,color='red')
plt.savefig('Covariance Matrix for 2 best PC', bbox_inches='tight')
plt.show()

plt.imshow(new_covariance_matrices_dictionary[3], cmap='Reds', interpolation='nearest')
plt.colorbar()
plt.title("New Covariance Matrix Combination 3 (best and worst PC)")
plt.text(-0.3, -0.66, "'THA076BCT026','THA076BCT027','THA076BCT041'", fontsize=8,color='red')
plt.savefig('Covariance Matrix for best and worst PC', bbox_inches='tight')
plt.show()

plt.imshow(new_covariance_matrices_dictionary[5], cmap='Reds', interpolation='nearest')
plt.colorbar()
plt.title("Covariance Matrix for two medium PCs")
plt.text(-0.3, -0.66, "'THA076BCT026','THA076BCT027','THA076BCT041'", fontsize=8,color='red')
plt.savefig('Covariance Matrix for two medium PCs', bbox_inches='tight')
plt.show()

plt.imshow(new_covariance_matrices_dictionary[7], cmap='Reds', interpolation='nearest')
plt.colorbar()
plt.title("New Covariance Matrix Combination 7 (All Components)")
plt.show()
```

```
# **STEP 9: Obtain final data and visualize**

# Two best PCs
final_data[1].shape
#final_data[1]

plt.ylim(-3,3)
plt.xlim(-3,3)
# plt.title("2 best PCs")
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.title("Scatter Plot after PCA for two best PCs on IRIS Dataset")
plt.text(-1.8, 2.8, "'THA076BCT026','THA076BCT027','THA076BCT041'", fontsize=8,color='red')
plt.scatter(final_data[1][:,0],final_data[1][:,1],c=y)
plt.savefig('Scatter Plot after PCA for two best PCs', bbox_inches='tight')

#Best and worst PCs
final_data[3].shape
#final_data[3]

plt.ylim(-3,3)
plt.xlim(-3,3)
plt.xlabel('PC1')
plt.ylabel('PC4')
plt.title("Scatter Plot after PCA for Best and worst PCs on IRIS Dataset")
plt.text(-1.8, 2.8, "'THA076BCT026','THA076BCT027','THA076BCT041'", fontsize=8,color='red')
plt.scatter(final_data[3][:,0],final_data[3][:,1],c=y)
plt.savefig('Scatter Plot after PCA for Best and worst PCs', bbox_inches='tight')

#Best and mediumly worse PCs
final_data[5].shape
#final_data[5]

plt.ylim(-3,3)
plt.xlim(-3,3)
plt.title("Scatter Plot after PCA for two Medium PCs on IRIS Dataset")
plt.text(-1.8, 2.8, "'THA076BCT026','THA076BCT027','THA076BCT041'", fontsize=8,color='red')
plt.xlabel('PC2')
plt.ylabel('PC3')
plt.scatter(final_data[5][:,0],final_data[5][:,1],c=y)
plt.savefig('Scatter Plot after PCA for two Medium PCs', bbox_inches='tight')

#All components
final_data[7].shape
#final_data[7]

plt.ylim(-3,3)
plt.xlim(-3,3)
plt.title("All PCs")
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.scatter(final_data[7][:,0],final_data[7][:,1],c=y)

# Create a 3D scatter plot
from mpl_toolkits.mplot3d import Axes3D

fig = plt.figure()
fig = plt.figure(figsize=(8, 8))
ax = fig.add_subplot(111, projection='3d')
ax.scatter(final_data[7][:,0],final_data[7][:,1],final_data[7][:,2],c=y, marker='o')
ax.set_xlabel('PC1')
ax.set_ylabel('PC2')
ax.set_zlabel('PC3')
ax.set_title('3D Scatter Plot of 3 PCs')
ax.text(-6, 2.8,1, "'THA076BCT026','THA076BCT027','THA076BCT041'", fontsize=8,color='red')
plt.tight_layout()
plt.savefig('3D Scatter Plot of 3 PCs', bbox_inches='tight',dpi=300)
plt.show()


# **STEP 10: PCA with library.**

from sklearn.decomposition import PCA
# Create instances of PCA with the desired number of components. (pca1 = 2components. pca2=so as to retain 95% info)
pca1 = PCA(n_components=2)
pca2 = PCA(0.95)

# Fit the PCA model to the data and transform the data
reduced_data1 = pca1.fit_transform(X)
reduced_data2 = pca2.fit_transform(X)

# Print the shape of the reduced data
print("Shape of reduced data:", reduced_data1.shape)
print("reduced through first", reduced_data1)

print("Shape of reduced data:", reduced_data2.shape)
print("reduced through second", reduced_data2)


# **STEP 11: Lets train 4 models and visualize the results.** 2 models are implemented after applying PCA using library. The third model is implemented after applying PCA using Math(using all the earlier above steps). The fourth model is trained without PCA.

# Create 4 sets of test-train for 4 models.
from sklearn.model_selection import train_test_split
```

```
x_train1,x_test1,y_train1,y_test1 = train_test_split(reduced_data1,y, test_size=0.2)   # using library, 2 components
x_train2,x_test2,y_train2,y_test2 = train_test_split(reduced_data2,y, test_size=0.2)   # using library, 0.95 info
x_train3,x_test3,y_train3,y_test3 = train_test_split(final_data[1],y, test_size=0.2)   # from scratch, 2 components
x_train4,x_test4,y_train4,y_test4 = train_test_split(X_train,y,test_size=0.2)          # no PCA

#training of the model
# model 1: using library, 2 components
from sklearn.svm import SVC
svm1 = SVC()
svm1.fit(x_train1, y_train1)

#library, 0.95 info
svm2=SVC()
svm2.fit(x_train2, y_train2)

#from scratch, 2 components
svm3=SVC()
svm3.fit(x_train3, y_train3)

#No PCA
svm4=SVC()
svm4.fit(x_train4, y_train4)

# Accuracy check for 4 models
from sklearn.metrics import accuracy_score

y_pred1 = svm1.predict(x_test1)
accuracy1 = accuracy_score(y_test1, y_pred1)
print("Accuracy1:", accuracy1)

y_pred2 = svm2.predict(x_test2)
accuracy2 = accuracy_score(y_test2, y_pred2)
print("Accuracy2:", accuracy2)

y_pred3 = svm3.predict(x_test3)
accuracy3 = accuracy_score(y_test3, y_pred3)
print("Accuracy3:", accuracy3)

y_pred4 = svm4.predict(x_test4)
accuracy4 = accuracy_score(y_test4, y_pred4)
print("Accuracy4:", accuracy4)

#prediciton from 4 models
predictions1 = svm1.predict(x_test1)
predictions2 = svm2.predict(x_test2)
predictions3 = svm3.predict(x_test3)
predictions4 = svm4.predict(x_test4)
predictions1.shape

# Lets define a function to plot confusion matrix
from sklearn.metrics import confusion_matrix
import seaborn as sns
def plot_confusion_matrix(y_true, y_pred, model_name,text):
    cm = confusion_matrix(y_true, y_pred)
    sns.heatmap(cm, annot=True, cmap="Blues", fmt="d")
    plt.title(f"Confusion Matrix - {model_name}")
    plt.xlabel("Predicted Labels")
    plt.ylabel("True Labels")
    plt.text(0.2, -0.5, "'THA076BCT026','THA076BCT027','THA076BCT041'", fontsize=8,color='red')
    plt.savefig(f'./plots/{model_name}.png', bbox_inches='tight',dpi=300)
    plt.show()

# Plot confusion matrix for model 1
plot_confusion_matrix(y_test1, predictions1, "UsingLibrary, 2 Components")

# for model 2
plot_confusion_matrix(y_test2, predictions2, "Using Library, 0.95 Info retention")

#for model 3
plot_confusion_matrix(y_test3, predictions3, "From Scratch, 2 Components")

#for model4
plot_confusion_matrix(y_test4, predictions4, "No PCA")
```

C. CANCER DATASET

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import sklearn
import seaborn as sns


# **STEP 1 : Load the dataset into a dataframe and analyze the data**
#specify path of csv file relative to "/content/drive" directory.
file_path = './dataset/Cancer_Data.csv'
df = pd.read_csv(file_path)
df.head(10)
df.shape

# Print the column names
print(df.columns)

# statistics computed for each column in the df
summary_stats = df.describe()
print(summary_stats)

# see the unique values in diagnosis column
```

```
unique_diagnosis = df['diagnosis'].unique()
print(unique_diagnosis)

# Looks like diagnosis is our target column. Lets assign X and y accordingly
# Separate the data and target variables
X = df.drop(columns=['id', 'diagnosis', 'Unnamed: 32'])
y = df['diagnosis']

print("Data:")
print(X.shape)
print(X.head())
print()
print("Target:")
print(y.shape)
print(y.head())
X.columns

# Create a heatmap of the attribute correlations
# High correlation (+ve or -ve) by bright colors, low correlation by dull colors.
plt.figure(figsize=(12, 8))
sns.heatmap(X.corr(), cmap='coolwarm', annot=True, annot_kws={'size': 5})
plt.title('Attribute Correlation Heatmap')
plt.text(8, -1.2, "'THA076BCT026','THA076BCT027','THA076BCT041'", fontsize=8,color='red')
plt.savefig('./plots/Corelation Matrix', bbox_inches='tight')
plt.show()


# **STEP 2: Preprocessing**
# Perform label encoding on the 'diagnosis' target variable
from sklearn.preprocessing import LabelEncoder
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)
print(y_encoded)

# change the dataframe's diagnosis column
df['diagnosis'] = y_encoded
df.head()

# Standardizing everything to have a common ground so that magnitude of some feature's value doesn't affect the PCA.
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaler.fit(X)
# Scaling of the whole dataframe.
df_scaled = pd.DataFrame(scaler.fit_transform(X), columns=X.columns)

# Define new X and y
X_train = df_scaled
y_train = df['diagnosis']
print("shape of new input X",X_train.shape)
print("shape of new output Y",y_train.shape)
df_scaled.head()

# Assign different colors to the data points based on the 'diagnosis' column
# Used to visually distinguish 2 different categories in that column
colors = ['blue' if t == 0 else 'red' for t in df['diagnosis'].map({0: 'B', 1: 'M'})]

X_train.cov()


# **STEP 3:CoVariance Matrix Calculation **
# covariance matrix will be 30*30 since X=(569,30). [each 30 features has covariance with each 30. so 30*30 matrix]
covariance_matrix = np.cov(X_train, rowvar=False)
print("Covariance matrix shape:", covariance_matrix.shape)
attribute_labels = X.columns
covariance_matrix = pd.DataFrame(covariance_matrix, columns=attribute_labels, index=attribute_labels)
print(covariance_matrix.head())

# Visualize covariance matrix
plt.figure(figsize=(12, 8))
sns.heatmap(covariance_matrix, cmap='coolwarm', annot=True, annot_kws={'size': 5})
plt.title('Covariance Matrix on Processed Cancer Data')
plt.text(8, -1.2, "'THA076BCT026','THA076BCT027','THA076BCT041'", fontsize=8,color='red')
plt.savefig('./plots/Covariance Matrix Processed Cancer Data', bbox_inches='tight')
plt.show()


#### **STEP 4: Eigen vectors/values calculation**
eigenvalues, eigenvectors = np.linalg.eig(covariance_matrix)
print("shape of eigenvalue:",eigenvalues.shape)
print("shape of eigen vectors:", eigenvectors.shape)
print("eigen values are",eigenvalues)
print("eigen vectors are",eigenvectors)


# **STEP 5: See the variance explained by each eigen values.**
# Element wise array division to obtain a
```

```
    new array
# proportion_of_variance array provides insights into relative importance of each PC in capturing the overall variance in the dataset
proportion_of_variance = eigenvalues/sum(eigenvalues)
proportion_of_variance

# Plot a scree plot to visualize how much variance is captured by which PC
cumulative_variance = np.cumsum(proportion_of_variance)

# range(1 to 30) on x axis, cummulative pov on y axis
plt.plot(range(1, len(proportion_of_variance) + 1), proportion_of_variance, marker='x', color='blue')
plt.plot(range(1, len(proportion_of_variance) + 1), cumulative_variance, marker='o', color='orange')
plt.xlabel('Principal Components(Eigen Vector)')
plt.ylabel('Proportion of Variance')
plt.title('Scree Plot')
legend_class = ["Explained Variance", "Cummulative Explained Variance"]
plt.legend(labels = legend_class)
plt.text(12, 0.9, "'THA076BCT026','THA076BCT027','THA076BCT041'", fontsize=8,color='red')
plt.savefig('./plots/Scree Plot', bbox_inches='tight')
plt.show()


# **STEP 6: Select the eigen vectors as needed to compute the final output** Formula: New data (Y) = Row feature vector * Row data
# Transpose data into row wise
transposed_X_train = np.transpose(X_train)
print("shape of transposed x train is", transposed_X_train.shape)
# Transpose eigen vectors into row wise for row feature vector
eigenvectors_transposed=np.transpose(eigenvectors)

# Dictionary to store the final Y(because we want to have multiple Y, each obtained by selecting our choice of combination of PC).
Y = {}
selected_components = [[0], [0, 1], [0,3], [2,3], [0,29]]
                            # best PC, 2 best PCs, best+medium, medium+medium, best+worst
count=len(selected_components)

# Iterate over the selected components
for i in range (0,count):
  selected_eigenvectors = eigenvectors_transposed[selected_components[i]]       # select the combination of components from above list
  Y[i] = np.transpose(selected_eigenvectors @ transposed_X_train)               # projection
  print(Y[i].shape)

# checking the nature of Y
print(type(Y))
print(Y.keys())
Y[1].shape
print(type(Y[1]))


# **STEP 7: Computing and visualizing covariance for the new data.**
# Dictionary to store all covariance matrices(obtained by selecting our choice of combination of PC).
new_covariance_matrices_dictionary = {}
for i in range(0, count):
  new_covariance_matrix = np.cov(Y[i], rowvar=False)
  new_covariance_matrices_dictionary[i] = new_covariance_matrix                                 # Combination:
  print("*Covariance matrix shape for combination",i,"is", new_covariance_matrix.shape)
  # 0=best PC, 1=2 best PCs, 2= best+medium PCs
  print("-->Covariance matrix:", new_covariance_matrices_dictionary[i])                         # 3= medium+medium PCs, 4= best+worst PCs

# Heatmap to visualize covariance matrix combination.
plt.imshow(new_covariance_matrices_dictionary[1], cmap='Reds', interpolation='nearest')  #instead of 'nearest', we can use 'bilinear','bicubic','spline16', 'spline36', 'hanning', 'hamming', 'hermite',etc..
plt.colorbar()
plt.title("New Covariance Matrix Combination 1 (2 best PCs)")
plt.show()

#lets try bilinear,pixel coloring is smoother(linear interpolation between 4 nearest data points)
plt.imshow(new_covariance_matrices_dictionary[2], cmap='Reds', interpolation='bilinear')
plt.colorbar()
plt.title("New Covariance Matrix Combination 2 (Best and medium PCs)")
plt.show()
```

```

#lets try bicubic,pixel coloring is again smoother(linear interpolation based on 16 nearest data points)
plt.imshow(new_covariance_matrices_dictionary[3], cmap='Reds', interpolation='bicubic')
plt.colorbar()
plt.title("New Covariance Matrix Combination 3 (Two medium PCs)")
plt.show()

#lets try spline16,pixel coloring is even smoother(16 degree of polynomial used for interpolation, for flexibility)
plt.imshow(new_covariance_matrices_dictionary[4], cmap='Reds', interpolation='spline16')
plt.colorbar()
plt.title("New Covariance Matrix Combination 4 (Best and worst PCs)")
plt.show()


# **STEP 8:Visualize the final output**
#Combination 0 = Y[0] = Best PC
#plt.ylim(-10,10)
#plt.xlim(-10,10)
varlegend= "Roll: 26,27,41"
plt.text(5,0.04,varlegend,color='red')
plt.title("1 best PC")
plt.xlabel('PC1')
plt.scatter(Y[0], np.zeros_like(Y[0]), c=y_train)   # colors = based on y_train

# Combination 1 = Y[1] = 2 best PCs
#plt.ylim(-3,3)
#plt.xlim(-3,3)
varlegend= "Roll: 26,27,41"
plt.text(5,4,varlegend,color='red')
plt.title("2 best PCs")
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.scatter(Y[2].iloc[:, 0], Y[2].iloc[:, 1], c=y_train)    # iloc used to access data

# Combination 3 = Y[3] = 2 medium PCs
#plt.ylim(-3,3)
#plt.xlim(-3,3)
varlegend= "Roll: 26,27,41"
plt.text(2,4.5,varlegend,color='red')
plt.title("2 medium PCs")
plt.xlabel('PC3')
plt.ylabel('PC4')
plt.scatter(Y[3].iloc[:,0],Y[3].iloc[:,1],c=y_train)

# Combination 4 = Y[4] = Best and worst PCs
plt.ylim(-3,3)
#plt.xlim(-3,3)
varlegend= "Roll: 26,27,41"
plt.text(5,2,varlegend,color='red')
plt.title("Best and worst PCs")
plt.xlabel('PC1')
plt.ylabel('PC30')
plt.scatter(Y[4].iloc[:,0],Y[4].iloc[:,1],c=y_train)


# **STEP 9: PCA using Library**
from sklearn.decomposition import PCA
# Create instances of PCA with the desired number of components. (pca1 = 2components. pca2=so as to retain 95% info)
pca1 = PCA(n_components=2)
pca2 = PCA(0.95)

# Fit the PCA model to the data and transform the data
reduced_data1 = pca1.fit_transform(X_train)
reduced_data2 = pca2.fit_transform(X_train)

# Print the shape of the reduced data
print("Shape of reduced data:", reduced_data1.shape)
print("reduced through first", reduced_data1)
print("Shape of reduced data:", reduced_data2.shape)
print("reduced through second", reduced_data2)


# Here it can be seen that to retain 95% info, 10 principle components are required
# Visualize the output for the first PCA done by library
#plt.ylim(-3,3)
#plt.xlim(-3,3)
varlegend= "Roll: 26,27,41"
plt.text(10,10,varlegend,color='red')
plt.title("Two best PCs-Using Library")
plt.xlabel('PC1')
plt.ylabel('PC2')
plt.scatter(reduced_data1.T[0], reduced_data1.T[1], c= y_train) # After applying transform method of PCA,Each row was represented as a sample. So need to transpose.

# For the 2nd PCA done by library(to retain 95% info), 10 components were required! We will try to plot 3d scatter plot to visualize.
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
fig = plt.figure(figsize=(10, 10))
ax = fig.add_subplot(111, projection='3d')
varlegend= "Roll: 26,27,41"
ax.text(-5,12.5,10,varlegend,color='red')


    # 3d coordinates required to specify
```

```
    the legend position
ax.scatter(reduced_data2.T[0],
    reduced_data2.T[1],reduced_data2.T
    [2],c=y_train)
ax.set_xlabel('PC1')
ax.set_ylabel('PC2')
ax.set_zlabel('PC3')
ax.set_title('3D Scatter Plot of 3 PCs')
plt.show()

# For the next step, we can train a model
    and compare the performance to see
    the impact of PCA we did so far.
```