Comparative Analysis of Machine Learning Classifiers on a Image Processing Dataset Using Wrapper Method Feature Selection

---

1. Abstract

This study evaluates the performance of various classifiers—Random Forest, Decision Tree, Support Vector Machine (SVM), Neural Networks (NN), and K-Nearest Neighbors (KNN)—for a specific classification task, following a comprehensive analysis of a raw dataset employing various techniques from Exploratory Data Analysis (EDA) to model evaluation.

The objective is to prepare the data through preprocessing, including normalization and scaling, extract significant features using different selection methods, and evaluate several classifiers. Performance is measured using metrics such as accuracy, precision, specificity, sensitivity, F1 score, and ROC AUC.

Random Forest stands out as the best classifier due to its superior cross-validation accuracy of 86.99%, high precision, and reliable F1-score. Its ensemble method provides a significant advantage in capturing data complexity and generalizing well to new data.

The Decision Tree is a close second, offering strong accuracy and interpretability but slightly less generalizability compared to Random Forest. In contrast, SVM, NN, and KNN displayed limitations in their current forms, indicating that they are not optimal without further tuning or adjustments.

The analysis also included a comparison of weighted AUC scores across different feature selection methods—backward selection, forward selection, and recursive feature elimination (RFE) where Random Forest consistently outperformed the Decision Tree.

Notably, RFE was identified as the most effective feature selection technique, enhancing classification capabilities. The findings underscore the importance of selecting appropriate classifiers and feature selection strategies to optimize performance in classification tasks.

Overview

A comprehensive analysis of a raw dataset, employing various techniques from Exploratory Data Analysis (EDA) to model evaluation.
My goal is to prepare the data through preprocessing, including normalization and scaling, extract significant features using different selection methods, and evaluate several classifiers. We will

measure the performance of these classifiers using metrics such as accuracy, precision, specificity, sensitivity, F1 score, and ROC AUC.

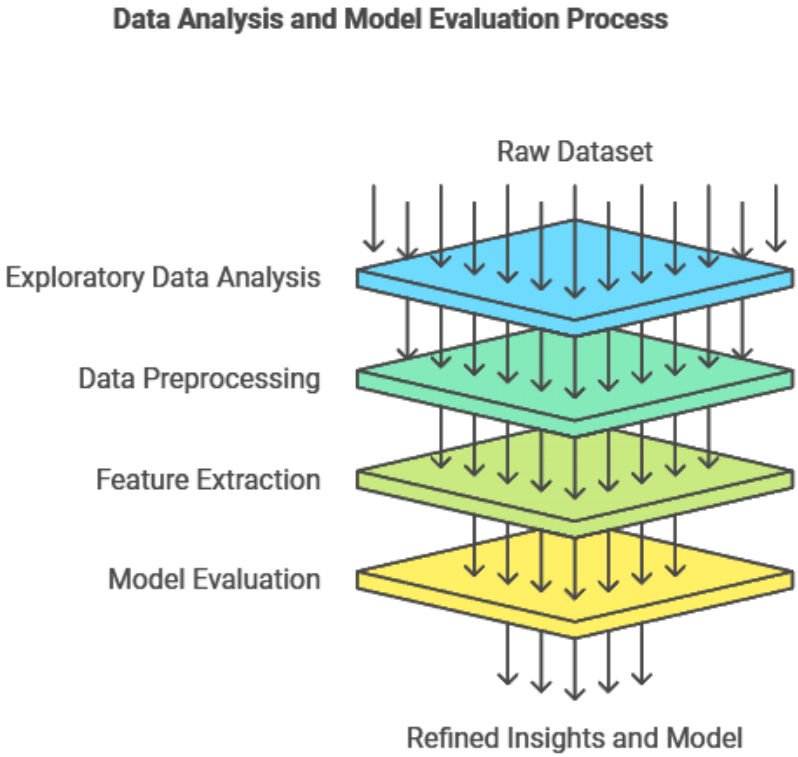

Figure 01: Data Analysis and Evaluation

2.1 Steps Involved

1. **Exploratory Data Analysis (EDA)**

    - **Data Visualization**: Use visualizations such as histograms, box plots, and scatter plots to understand the distribution of features and identify potential outliers.
    - **Correlation Analysis**: Generate correlation matrices to identify relationships between features.

2. **Data Preprocessing**

    - **Handling Missing Values**: Identify and impute or remove missing values.
    - **Encoding Categorical Variables**: Convert categorical variables to numerical format using techniques like one-hot encoding.
    - **Resizing and Scaling**: Normalize and standardize features to ensure they are on a similar scale, which is crucial for many algorithms.
    - **Feature Engineering**: Create new features if necessary, based on domain knowledge.

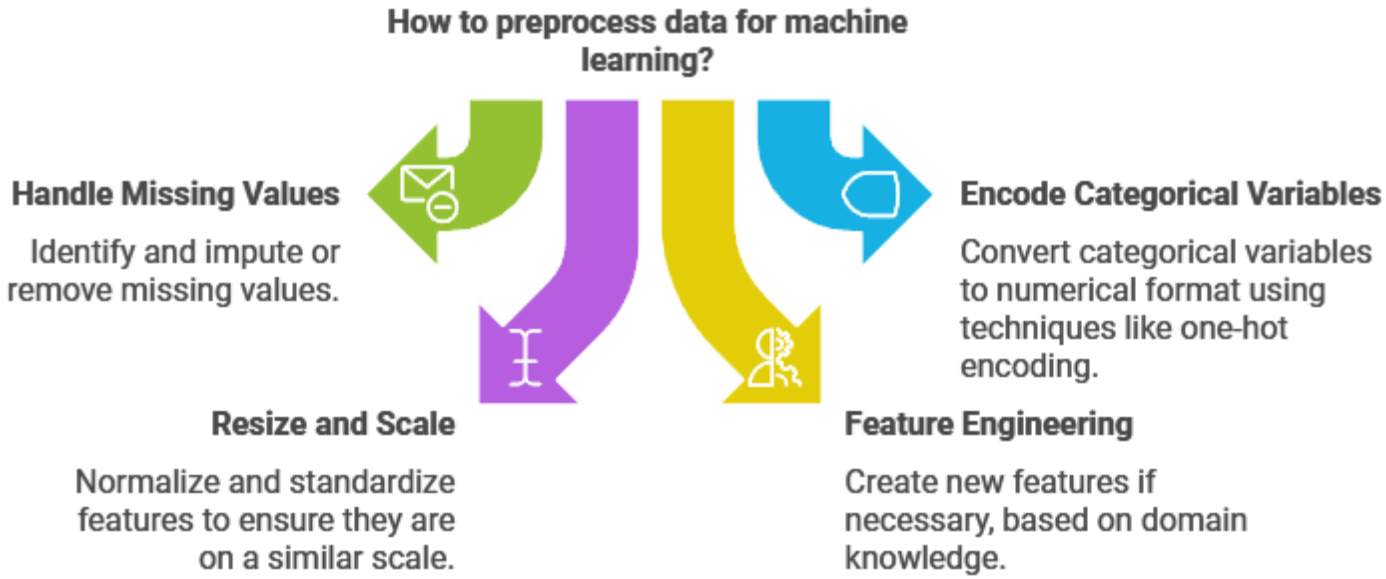

Figure 02: Preprocessing data

3. **Feature Extraction and Selection**

    - **Feature Extraction**: Use techniques like PCA (Principal Component Analysis) or create polynomial features if applicable.
    - **Feature Selection**:
        - **Recursive Feature Elimination (RFE)**: Select features based on their importance using a specified model.
        - **Forward Selection**: Iteratively add features based on performance improvement.
        - **Backward Elimination**: Iteratively remove features that do not contribute significantly to model performance.

4. **Modeling**

    - Defining and initialize classifiers:
        - Decision Tree (DT)
        - Random Forest (RF)
        - Support Vector Machine (SVM)
        - K-Nearest Neighbors (KNN)
        - Neural Network (Multi-Layer Perceptron)
    - Split the dataset into training and testing sets using train test split.

5. **Model Evaluation**

    - **10-Fold Cross-Validation**: Evaluate each model using K-Fold cross-validation to ensure robustness.
    - **Performance Metrics**:
        - **Accuracy**: Overall correctness of the model.
        - **Precision**: True positives divided by the sum of true and false positives.
        - **Specificity**: True negatives divided by the sum of true negatives and false positives.
        - **Sensitivity (Recall)**: True positives divided by the sum of true positives and false negatives.
        - **F1 Score**: Harmonic mean of precision and recall.
        - **ROC AUC**: Area under the ROC curve, providing a measure of performance across different threshold settings.
    - **Confusion Matrix**: Visual representation of true vs. predicted classifications.

6. **Visualization**

    - **Loss Curve Plot**: For the neural network, plot the loss curve to visualize training progress.
    - **Confusion Matrix Heatmap**: Visualize the confusion matrix using a heatmap for better interpretation.

---

3. Data Generation

Data Generation for Image Processing

Data generation in image processing is essential for training machine learning models, particularly deep learning architectures that require large and diverse datasets. Here are key aspects of data generation specifically tailored for image processing tasks:

Methods of Data Generation

1. **Data Augmentation**:
    - This technique involves applying various transformations to existing images to create new variations. Common augmentations include:
        - **Scaling**: scale pixel values to [0,1])

- **Normalization**: Normalize (mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]) Normalize image using ImageNet mean and std
- **Resize:** Images to 128x128 pixels Convert images to PyTorch tensors

2. **Procedural Generation**:
    - This approach uses algorithms to create images based on defined rules or parameters. It is often utilized in game development and simulation, where landscapes, textures, and other visual elements can be generated procedurally.

3. **Transfer Learning**:
    - This method leverages pre-trained models, which have been trained on large datasets like ImageNet. By fine-tuning these models on smaller, specific datasets, practitioners can enhance performance without needing to generate vast amounts of new training data.

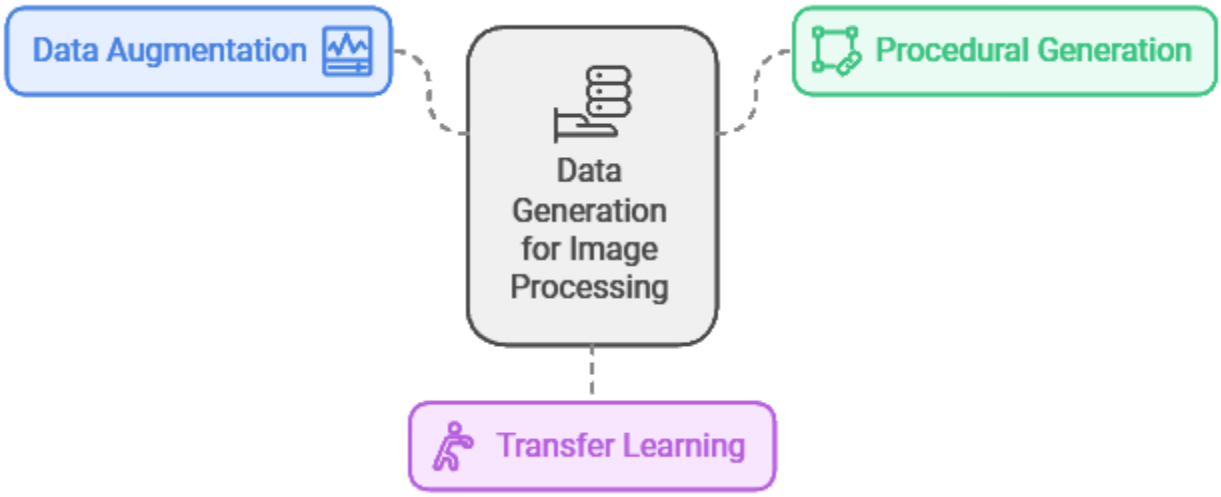

Figure 02: Methods of Data Generation

Data generation for image processing is a critical aspect of developing effective machine learning models. By employing techniques such as data augmentation, synthetic image generation, and procedural methods, practitioners can create diverse datasets that enhance model robustness and performance. Thoughtful consideration of realism, diversity, and ethical implications will ensure that generated data serves its intended purpose effectively.

---

4. Feature Extraction and Feature Selection *(Feature Extraction Methods: A Review To cite this article: Wamidh K. Mutlag et al 2020)*

Feature Extraction

In the context of this analysis, we extracted a set of features from the data using a sliding window approach. The features were calculated for each window of data and included:

1. **Mean**: The average value within the window, representing the central tendency.

$$M_j = \sum_m^{i=1} \frac{1}{M} P_{ji}$$

Description: The mean is the average pixel intensity value within the sliding window.
Importance in Image Processing: It serves as a basic descriptor of brightness or luminosity within that region. In images, different objects or regions can have distinct average brightness values, which can help in distinguishing between them. For instance, a bright object will have a higher mean pixel value compared to a darker background.

2. **Standard Deviation (stddev)**: A measure of the amount of variation or dispersion in the data.

$$\sigma_j = \sqrt{\frac{1}{M}\sum_M^{i=1}(P_{ji} - M_j)^2}$$

Description: Standard deviation measures the variation of pixel intensity values from the mean within the window.
Importance: A high standard deviation indicates that there is a wide range of intensity values, suggesting the presence of edges or textures, whereas a low standard deviation implies uniformity or smoothness. This can be particularly useful in texture analysis, where different textures have different levels of intensity variation.

3. **Variance**: The square of the standard deviation, providing a measure of how far each number in the set is from the mean.

$$\sigma^2 = \frac{1}{q}\sum_{i=1}^{q}(Y_j - M)^2$$

Description: Variance is the square of the standard deviation, representing how spread out the pixel values are around the mean.
Importance: Similar to standard deviation, variance provides insight into the texture of the image. It can help in detecting areas of detail versus smooth areas, assisting in applications like image segmentation where distinct regions must be identified based on texture differences.

4. **Skewness**: A measure of the asymmetry of the distribution of values in the window.

$$S_j = \sqrt[3]{\frac{1}{M}\sum_M^{i=1}(P_{ji} - M_j)^3}$$

Description: Skewness measures the asymmetry of the distribution of pixel intensities within the window.
Importance: In image processing, skewness can indicate the presence of particular features or structures. For instance, a positive skewness might suggest a predominance of brighter pixels, which can be useful for detecting highlights or reflective surfaces in images.

5. **Kurtosis**: A measure of the "tailedness" of the distribution, indicating the presence of outliers.

$$\text{Kurtosis} = \sum_{j=1}^{M} \sum_{i=1}^{N} \frac{(q(j,i)-m)^4}{(MN)\sigma^4}$$

Description: Kurtosis assesses the "tailedness" of the pixel intensity distribution.
Importance: High kurtosis values can indicate the presence of outliers in the pixel intensity distribution, which could correspond to noise or significant features in the image (like sharp edges or spots). This is particularly useful in tasks such as anomaly detection or enhancing features in medical imaging.

6. **Energy**: The sum of the squared values, reflecting the overall intensity of the signal in the window.

$$\text{Energy} = \sum\sum q(i,j)^2$$

Description: Energy is calculated as the sum of the squared pixel values within the window.
Importance: In image processing, energy can indicate the overall intensity or strength of the signal within a specific area. High energy regions could correspond to detailed areas or strong edges, which are often of interest in applications such as feature detection and object recognition.

7. **Entropy**: A measure of the randomness or unpredictability in the data, providing insight into its complexity.

$$\text{Entropy} = -\sum\sum q(i,j)\log q(i,j)$$

Description: Entropy measures the amount of randomness or unpredictability in the pixel intensity values.
Importance: In image processing, entropy can provide insight into the complexity of a region. High entropy indicates a lot of variation in pixel values, suggesting a complex texture, while low entropy suggests uniform areas. This is valuable for segmenting images into regions of interest based on complexity.

8. **Smoothness**: Comparative smoothness, Q is a measurement of gray level disparity that which can used to create relative smoothness recipes.

$$Q = 1 - \frac{1}{1+\sigma 2}$$

Description: Smoothness can be assessed through various metrics that evaluate the disparity in pixel values.
Importance: In image analysis, smoothness can indicate how continuous or abrupt transitions are in an image. This is important in applications such as edge detection, where sharp changes in intensity correspond to edges of objects, and in texture analysis, where different textures can exhibit different levels of smoothness.

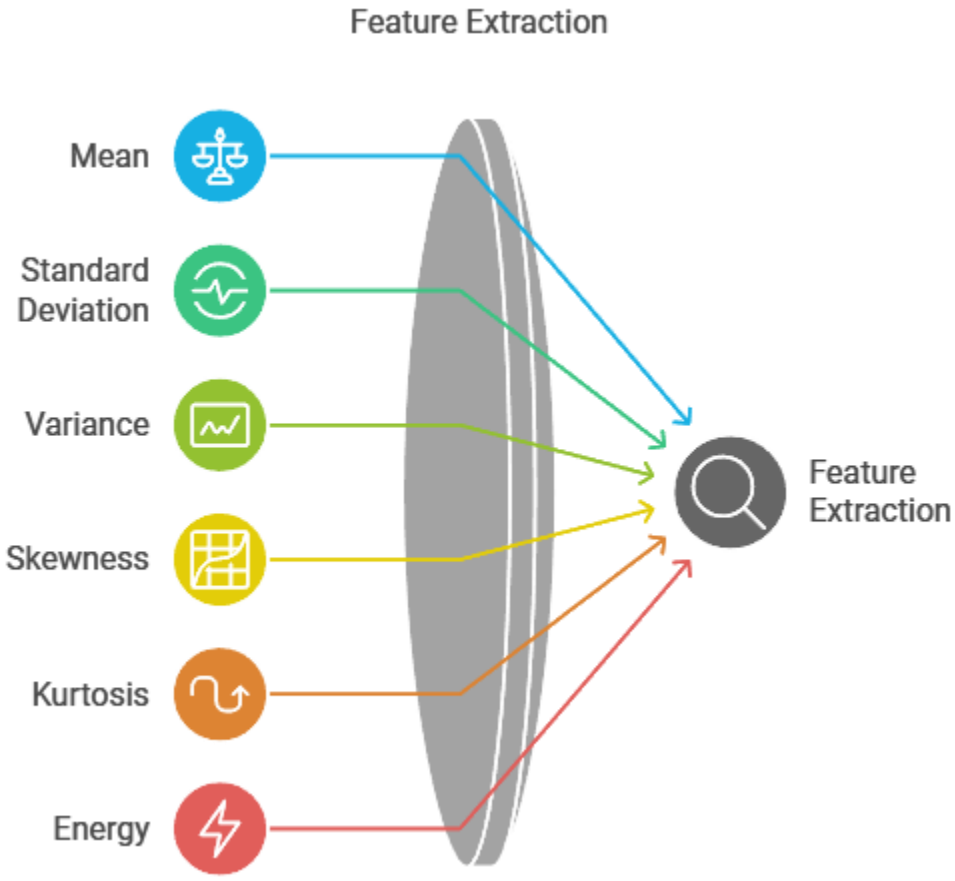

Figure 04: Feature Extraction

These features provide a comprehensive view of the characteristics of the data within each window, capturing both statistical properties and distributional features.*(Wamidh K. Mutlag et al 2020)*

Application in Image Processing Tasks

When combined, these features allow for a comprehensive analysis of images. Here are some practical applications:

- **Image Classification**: By analyzing these features, algorithms can classify images based on their content (e.g., distinguishing between natural and artificial scenes).
- **Texture Analysis**: Features like standard deviation, skewness, and entropy help in identifying and categorizing different textures, which is useful in fields like material science and remote sensing.
- **Object Detection**: Energy and smoothness can help highlight areas of interest, aiding in locating objects within an image.
- **Segmentation**: These features can facilitate segmenting images into meaningful regions, which is critical in medical imaging for identifying tumors or organs.

- **Feature Extraction for Machine Learning**: These characteristics serve as input features for machine learning models, improving their performance in tasks like recognition and classification

Extracted Feature Visualization:

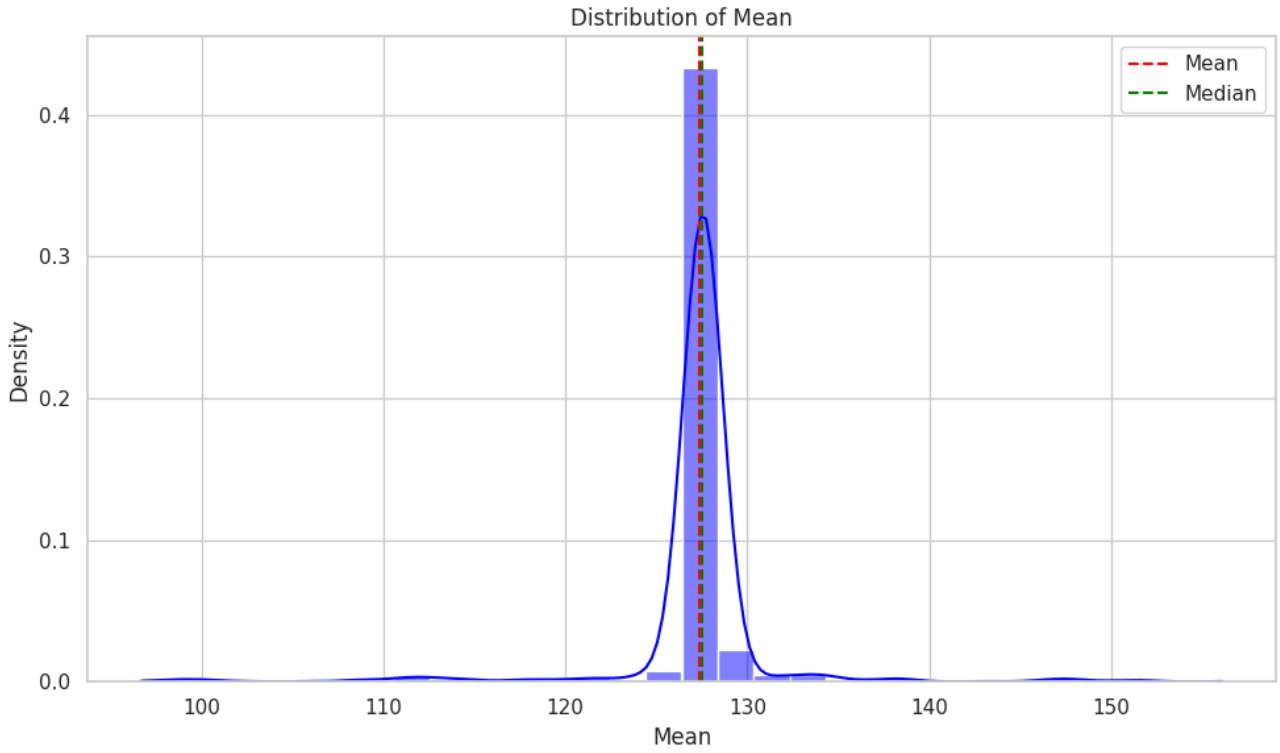

Figure 05: Mean

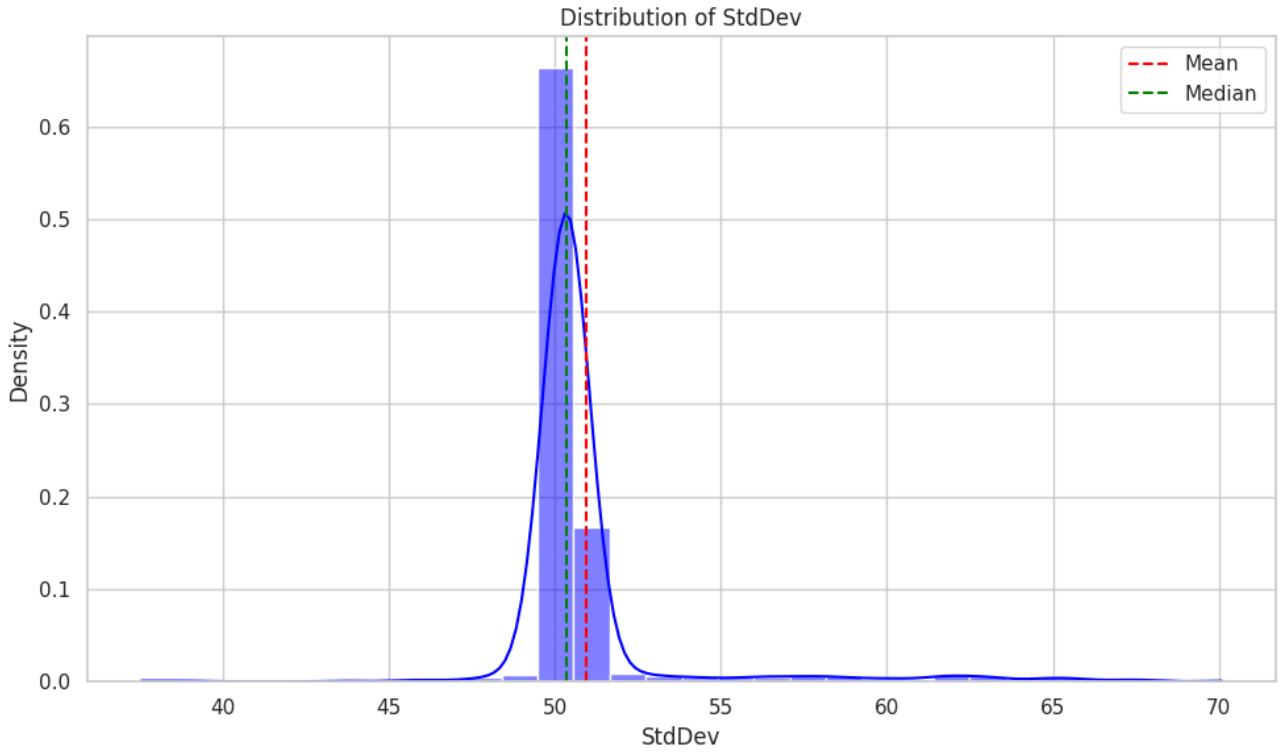

Figure 06: Standard Deviation

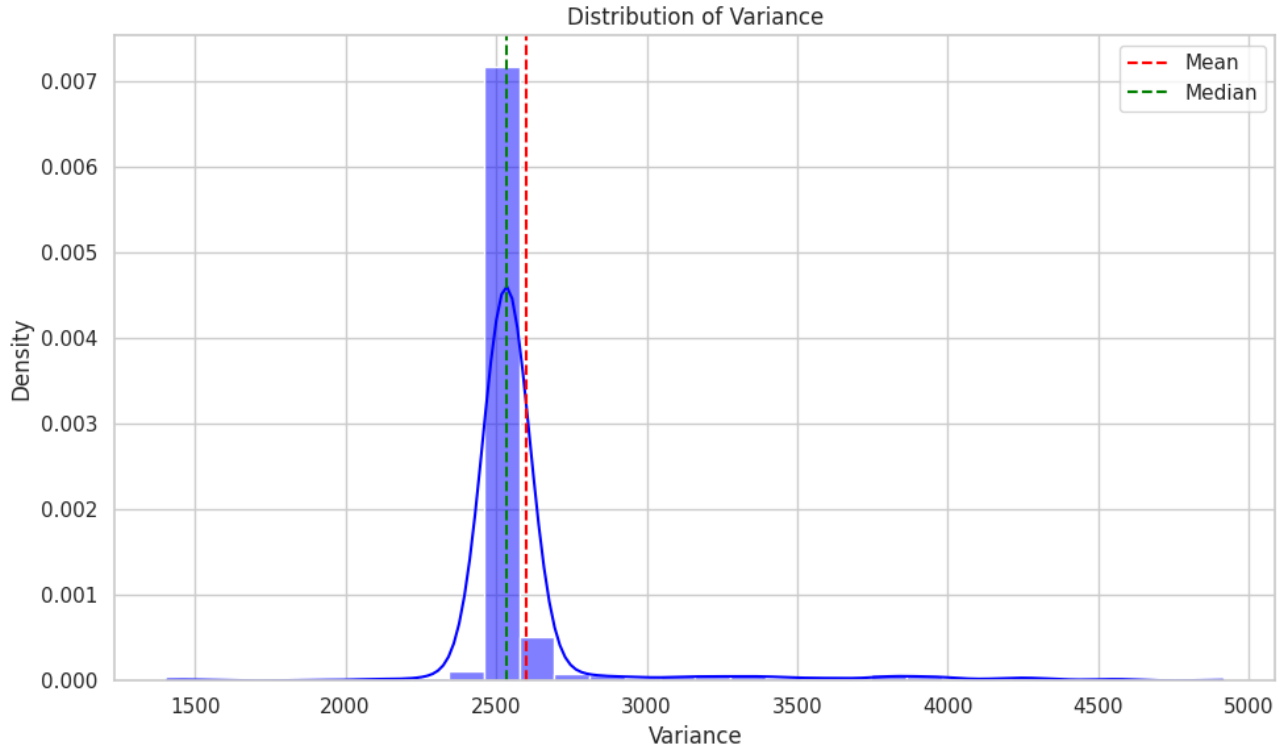

Figure 07: Variance

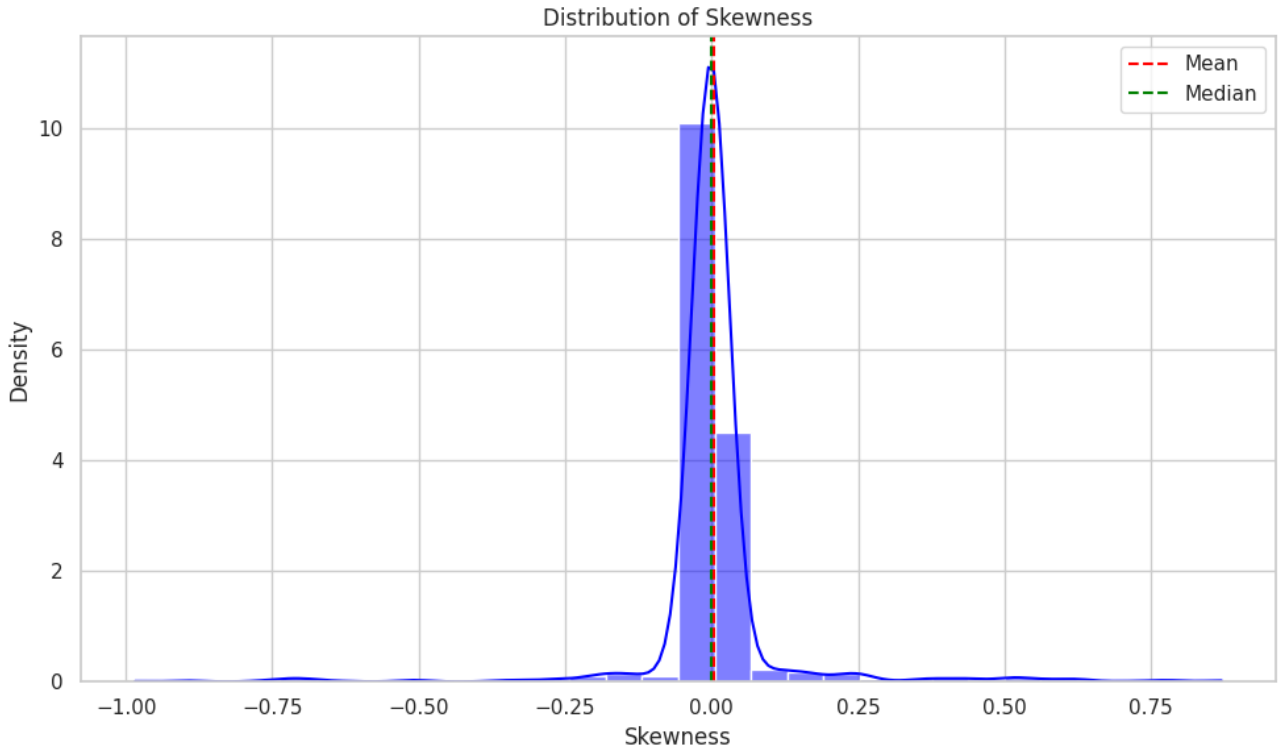

Figure 08: Skewness

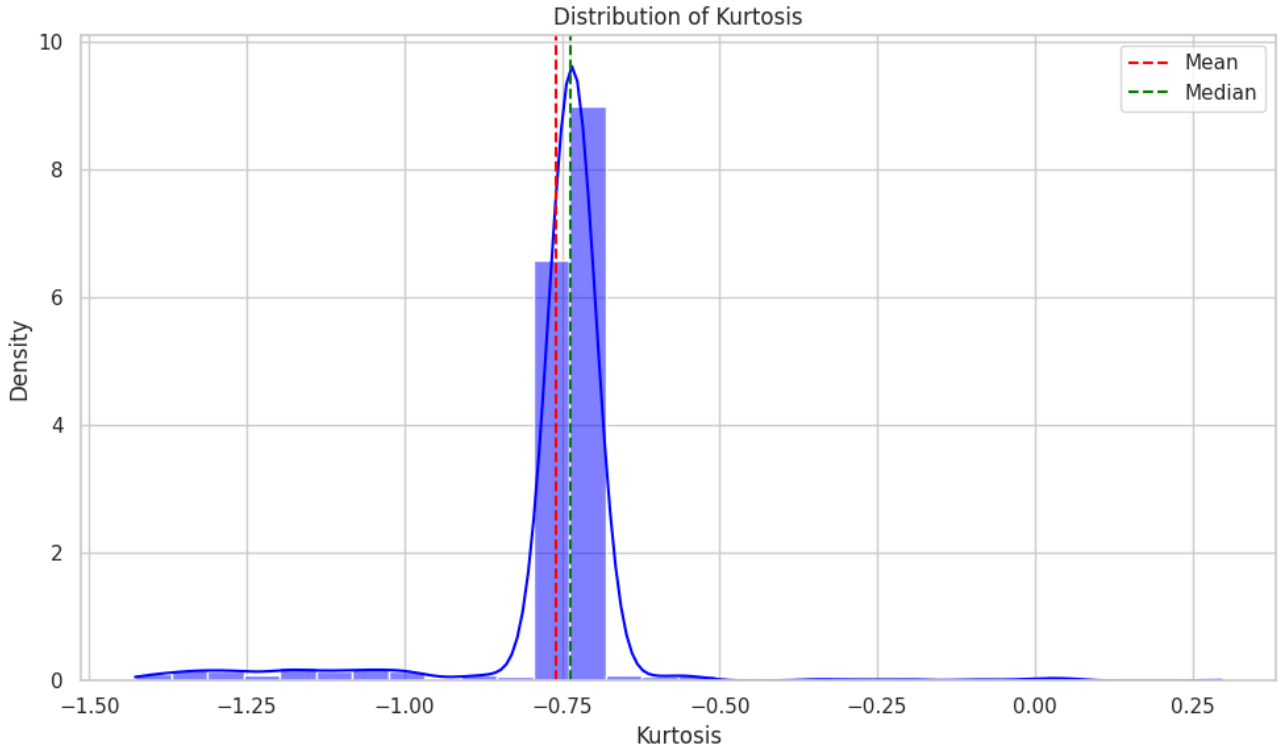

Figure 09: Kurtosis

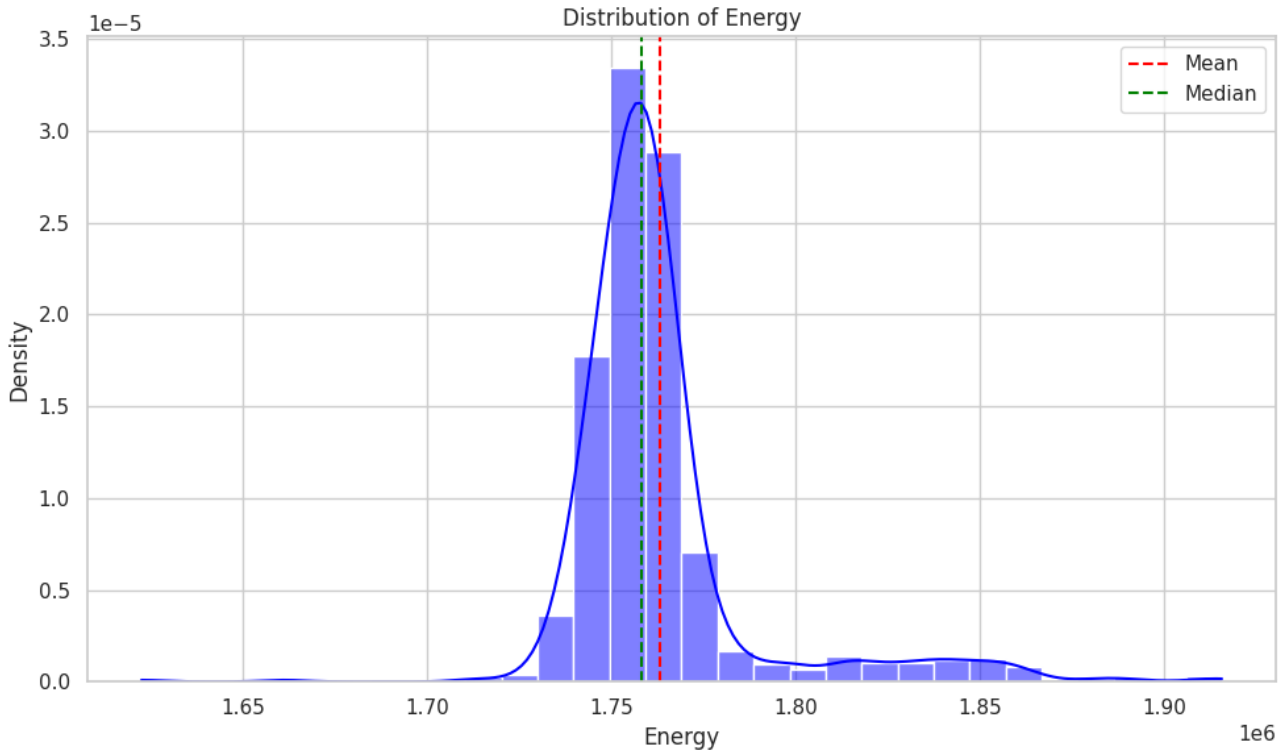

Figure 10: Energy

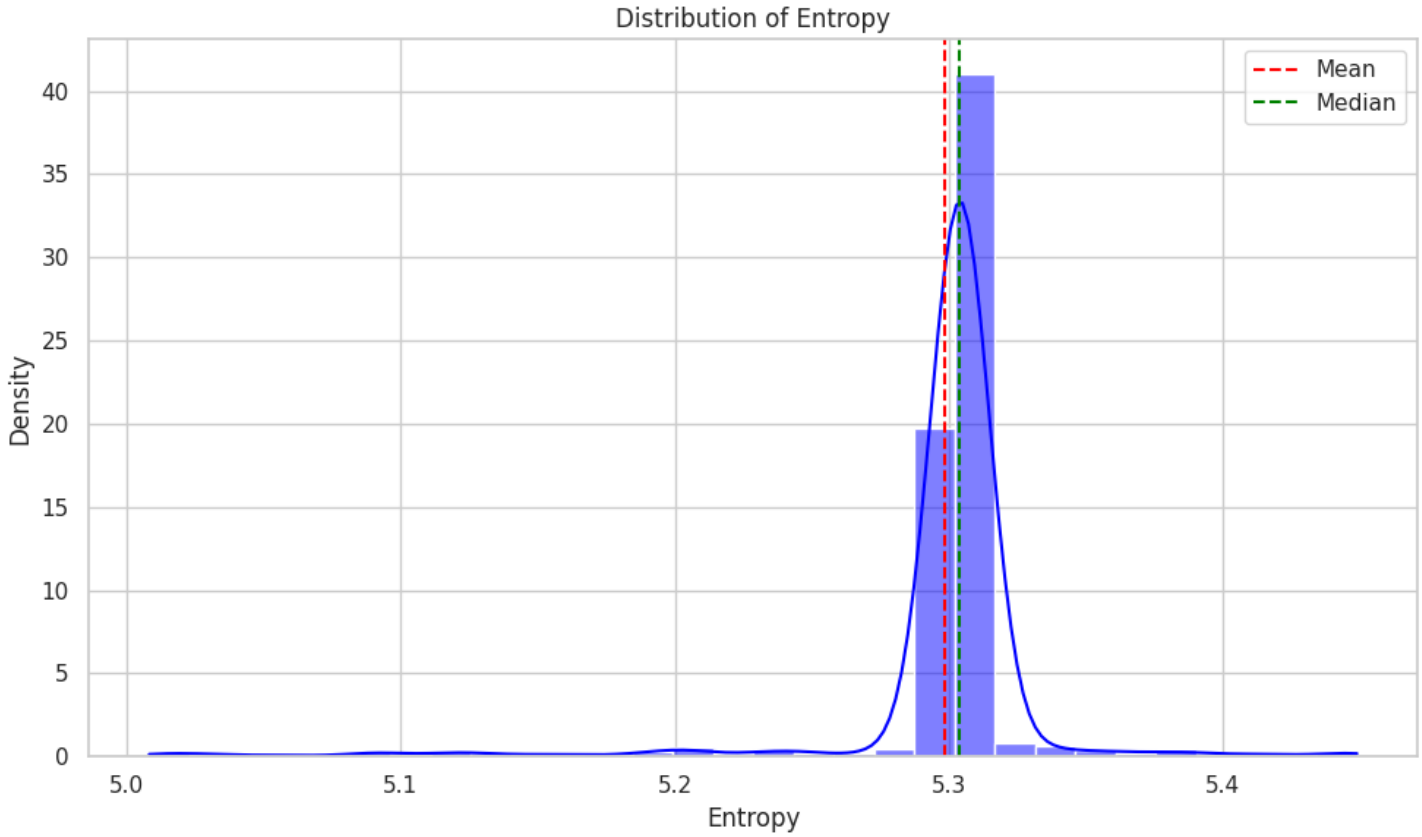

Figure 11: Entropy

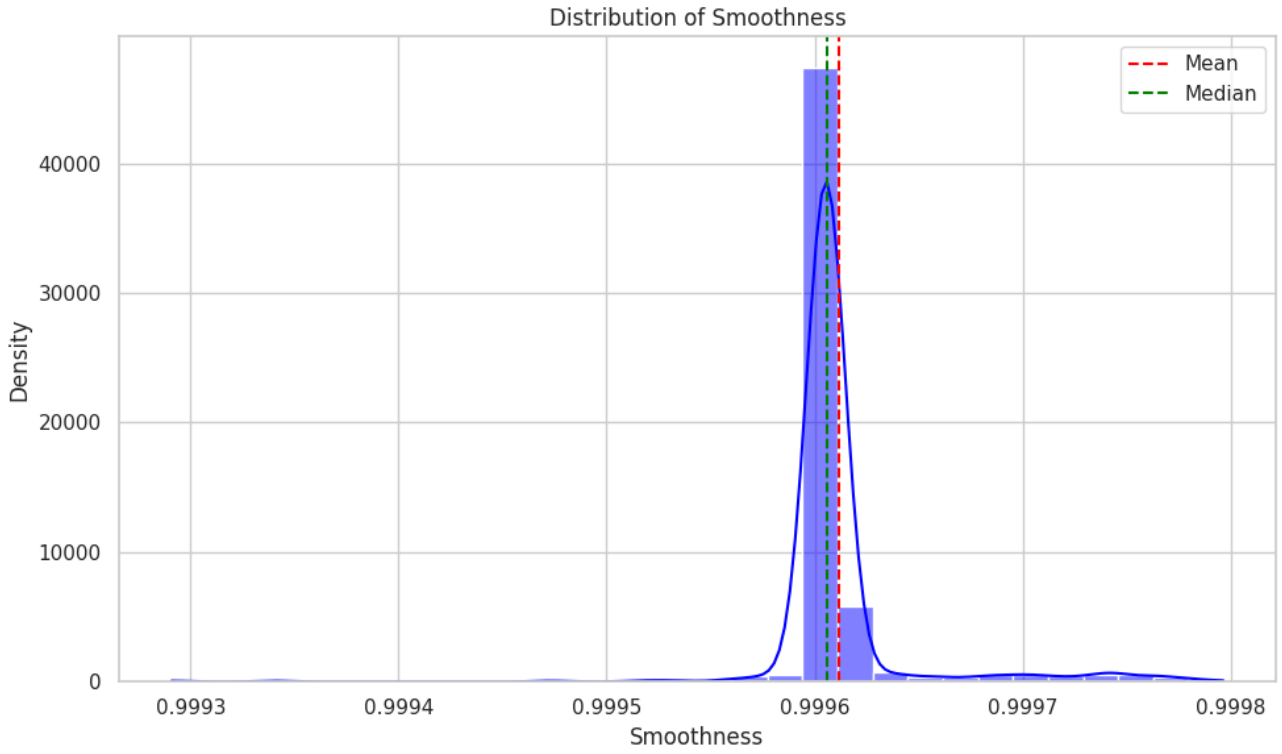

Figure 12: Smoothness

5. ***Feature Selection***

To enhance model performance and interpretability, feature selection was performed using wrapper methods, specifically Forward Selection, Backward Elimination, and Recursive Feature Elimination (RFE). The goal was to identify the most relevant features for the classification task.

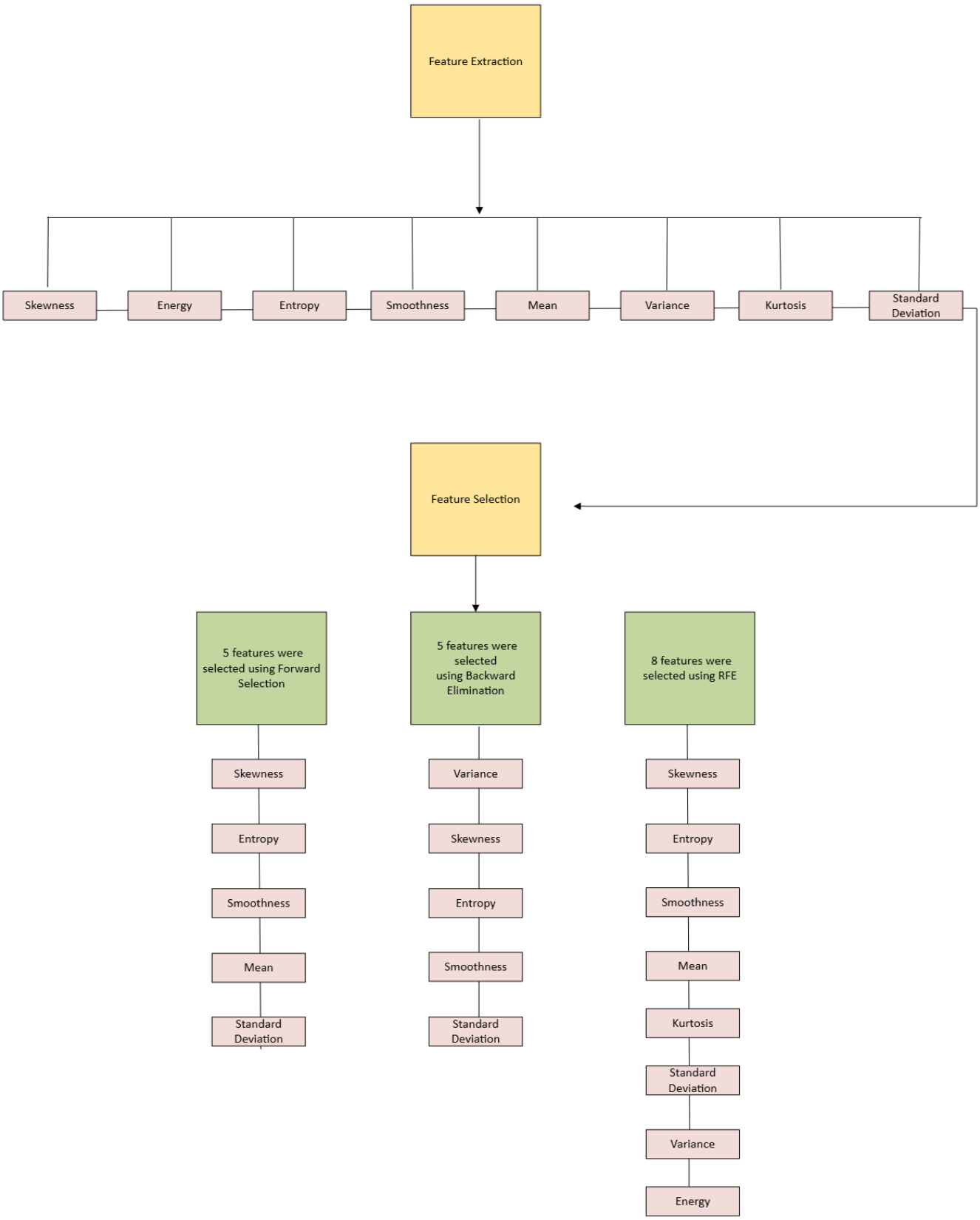

Fig 13: Represent Feature Extraction and Feature Selection

1. **Forward Selection**: This method starts with no features and iteratively adds the most significant feature at each step based on model performance until no further improvements can be made.

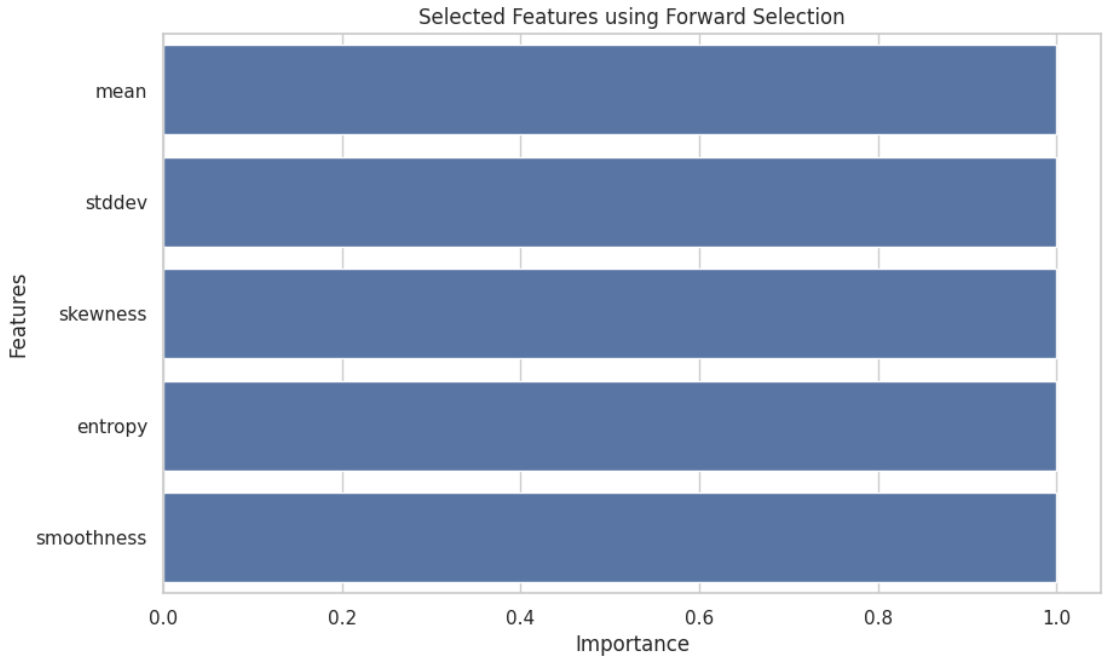

Fig 14: Selected Features using Forward selection

2. **Backward Elimination**: Starting with all features, this method iteratively removes the least significant feature until the desired number of features is reached or performance no longer improves.

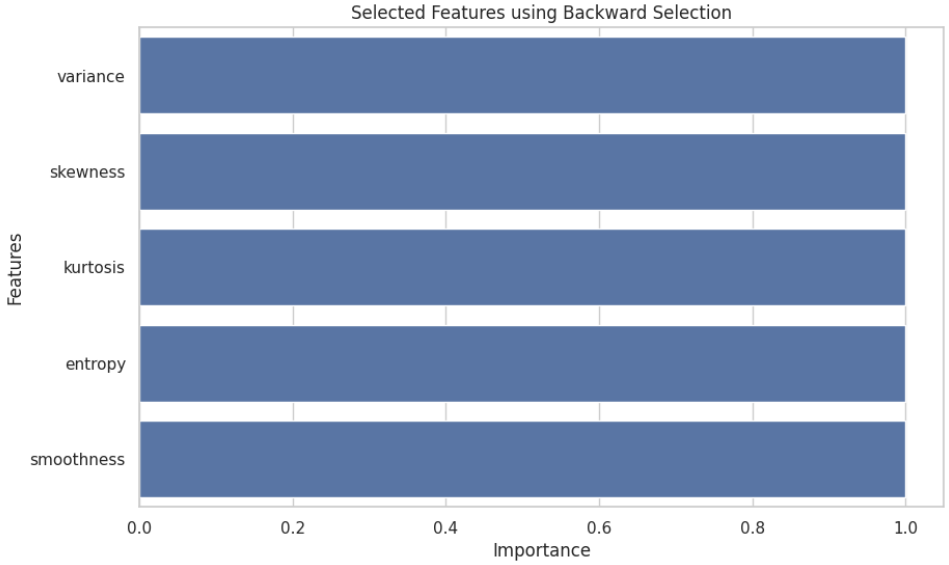

Fig 15: Selected Features using Backward selection

3. **Recursive Feature Elimination (RFE)**: This technique fits the model multiple times and removes the least important features based on model coefficients, focusing on those that contribute most to performance.

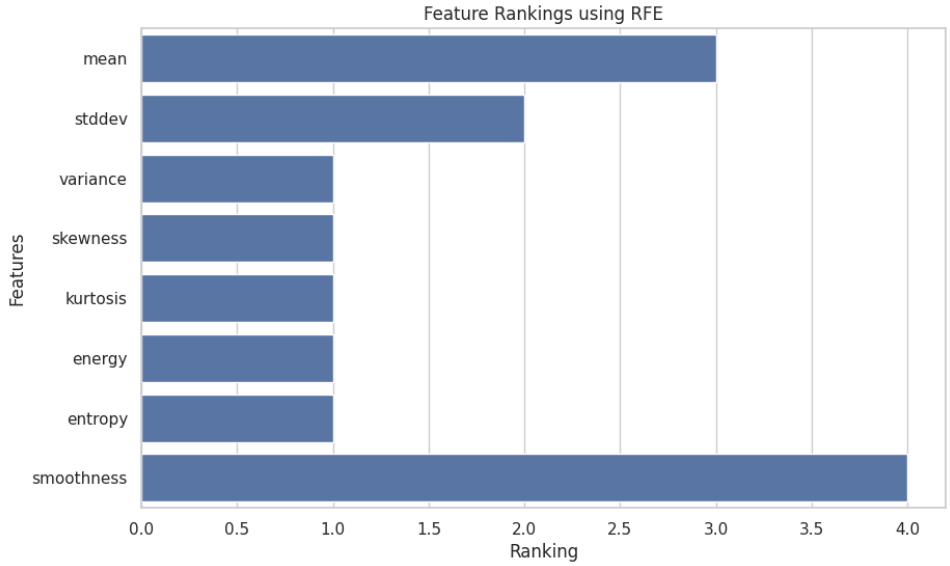

Fig 16: Selected Features Ranking using RFE

Visualization

Visualizing the selected features can provide insights into their distributions and relationships. Common visualization techniques include:

- **Box Plots**: To illustrate the distribution of each feature across different classes.

The feature extraction and selection process highlighted the importance of statistical features in enhancing model performance. By focusing on the most informative features, we can improve the classification task's accuracy and interpretability. Future work will involve modeling these features to assess their impact on performance metrics.

5.Result and Discussion

5.1 Raw Data Visualization and Preprocessing Visualization

1. Raw Data Visualization

Visualizing raw data is essential for understanding its structure, distribution, and potential relationships among variables. Common visualization techniques include:

- **Histograms**: Useful for examining the distribution of individual features. They can reveal intensity.
- **Box Plots**: Effective for identifying outliers and understanding the spread of data across different categories. They display the median, quartiles, and potential outliers.
- **Scatter Plots**: Useful for examining relationships between two continuous variables. They can help identify trends, clusters, and correlations.
- **Pair Plots**: Provide a matrix of scatter plots for each pair of features, enabling a comprehensive view of relationships and distributions.
  This plot is a pair plot, often used to visualize relationships between several numerical features and how they may differ across categories in a dataset. Here's a breakdown of the main elements:

1. **Axes**:
   - The variables mean_pixel_intensity, std_pixel_intensity, min_pixel_intensity, and max_pixel_intensity are shown on both x and y axes. This setup allows you to compare each pair of features with each other.
2. **Histograms (Diagonal)**:
   - Along the diagonal are histograms for each variable. These histograms give an idea of the distribution of each feature individually. For example, mean_pixel_intensity and max_pixel_intensity appear to have a skewed distribution with peaks around certain values, while std_pixel_intensity has a more normal distribution.
3. **Scatter Plots (Off-diagonal)**:
   - The scatter plots on the off-diagonal show pairwise relationships between the variables. Each point represents an instance, and different colors represent different class_folder categories, making it possible to observe how these categories might cluster in the feature space.
   - For example, std_pixel_intensity vs. mean_pixel_intensity shows points from different classes spread out, with some overlapping, indicating that std_pixel_intensity might not be a strong distinguishing feature on its own.
4. **Legend**:
   - The legend indicates different classes (labeled by class_folder) with different colors. This color-coding makes it easy to see if certain classes cluster together or are spread out in the feature space.
5. **Insights**:

- Some features like min_pixel_intensity and max_pixel_intensity seem to have more separated clusters for certain classes, potentially making them more useful for classification.
- Features with more overlap across classes, such as std_pixel_intensity, may be less useful for distinguishing between the classes on their own.

- **Correlation Heatmap**: Visualizes the correlation matrix, helping identify potential multicollinearity and relationships among features.

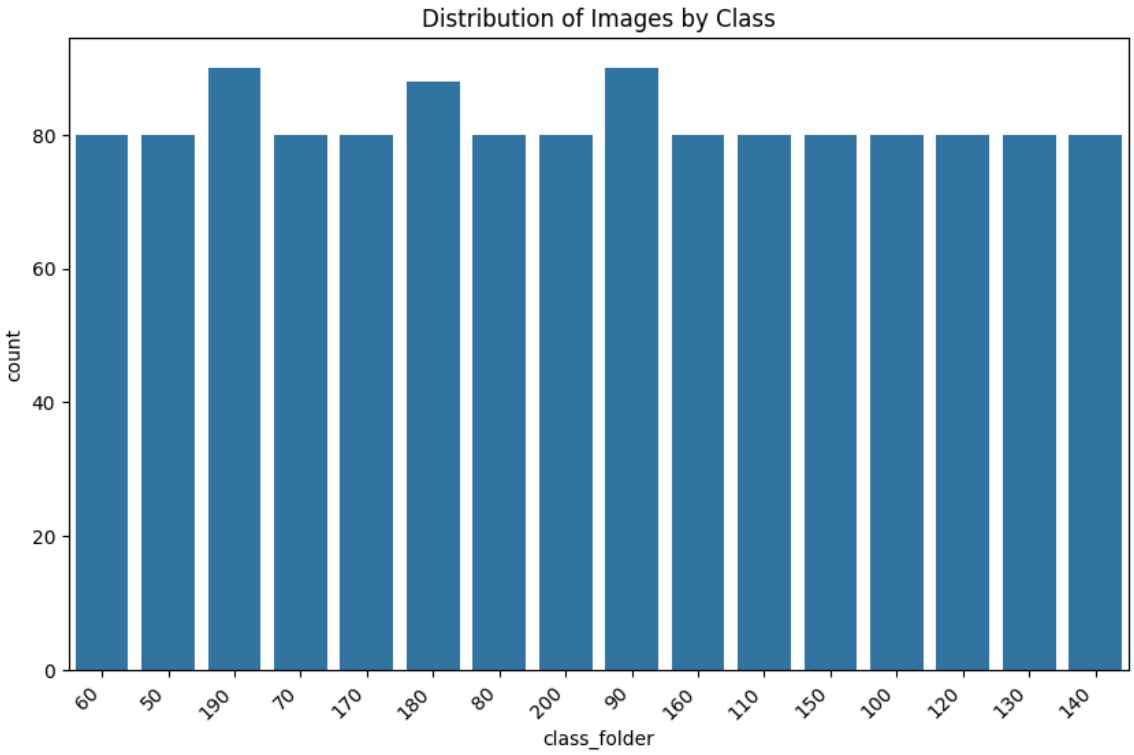

Fig 17: Distribution of images by Class

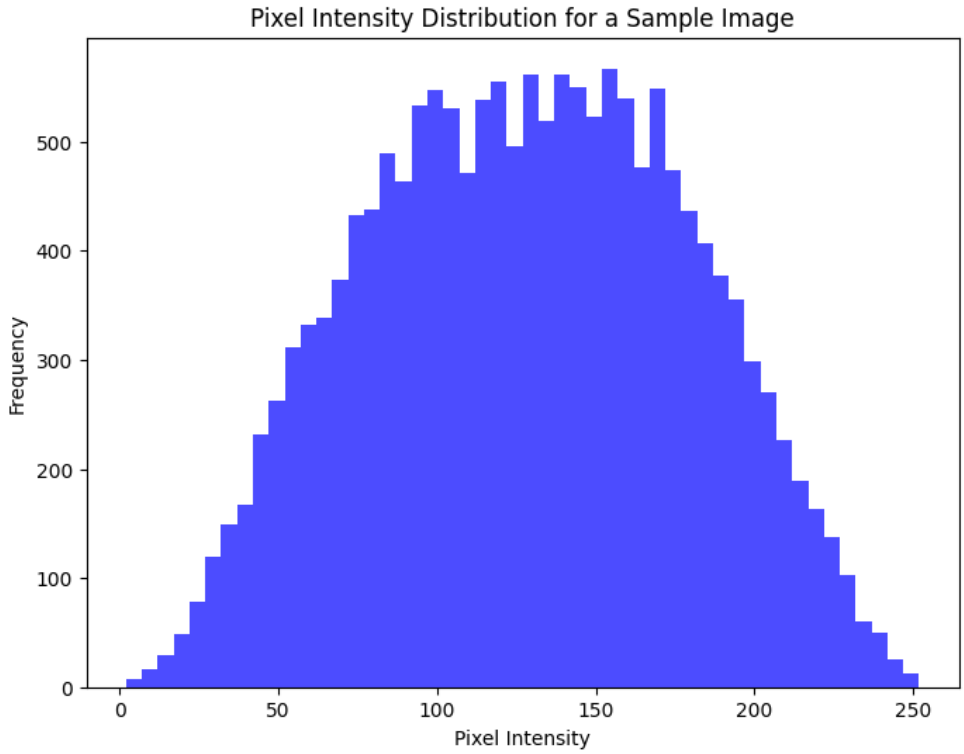

Fig 18: Pixel Intensity Distribution for a Sample Image

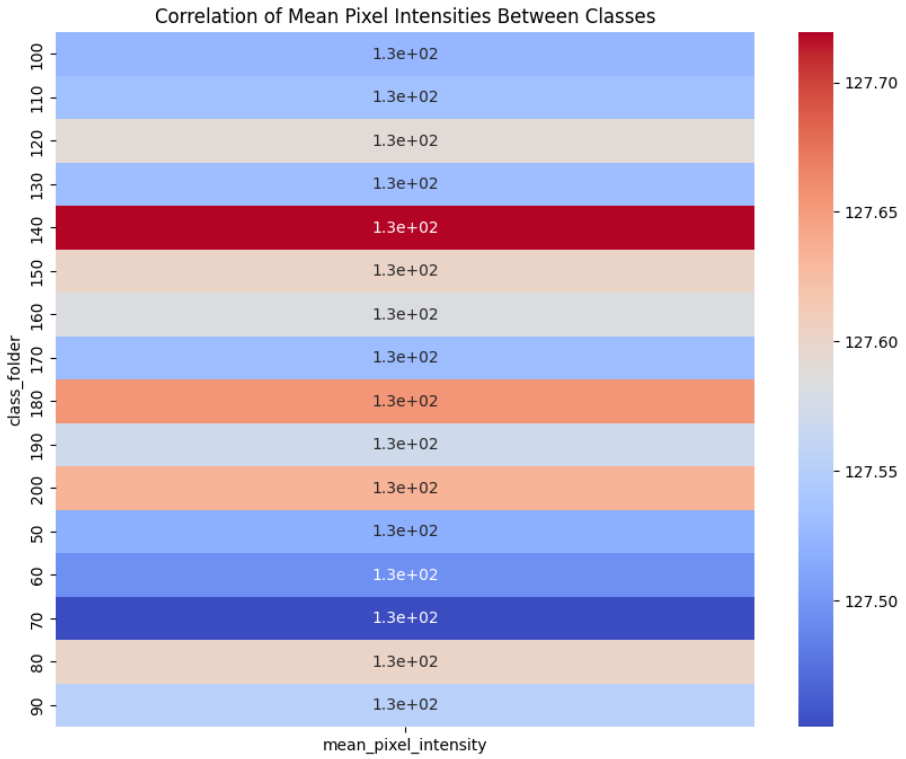

Fig 19: Correlation of mean pixel intensity between class

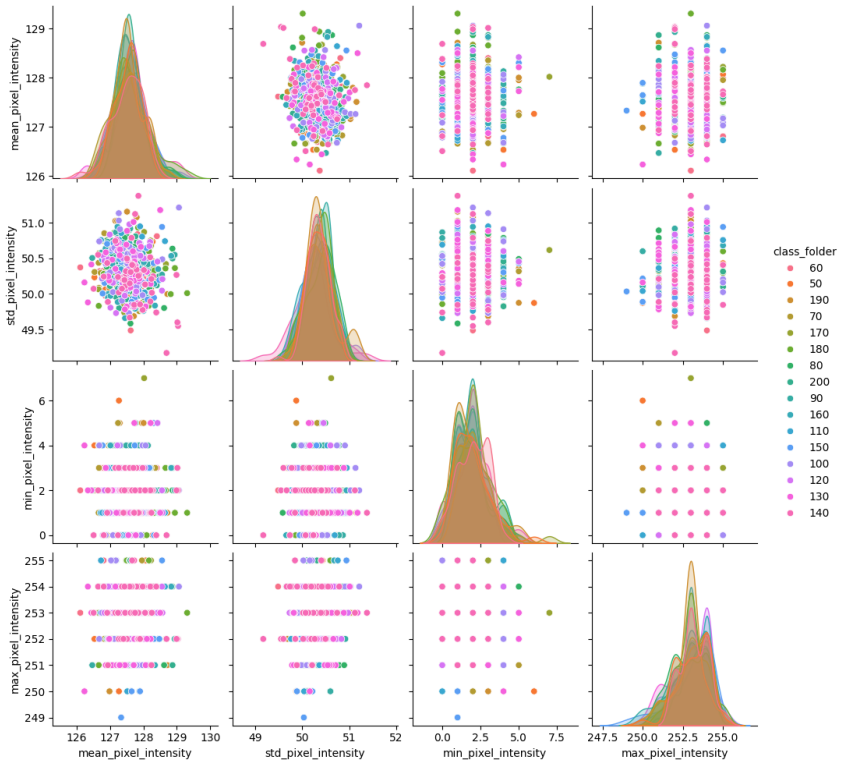

Fig 20: Pair plot between class

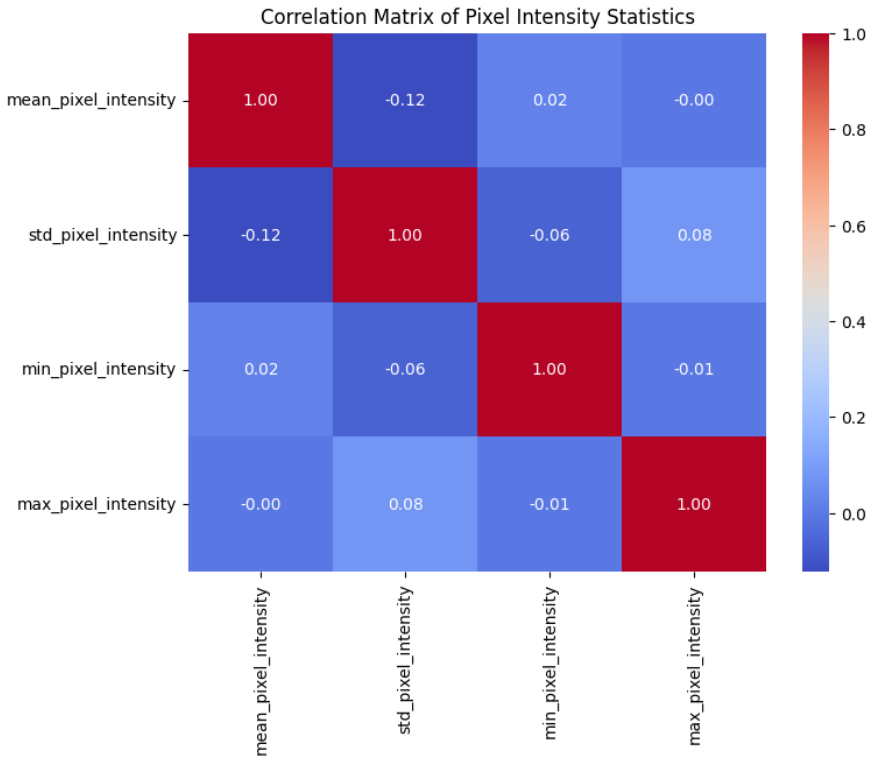

Fig 21: Correlation Matrix of Pixel Intensity

2. Preprocessing Visualization

Preprocessing steps often include scaling, normalization, and handling missing values. Visualizations at this stage help assess the impact of these transformations:

- **Distribution Plots (Before and After Scaling)**: Comparing the distribution of features before and after scaling (e.g., Min-Max scaling or Standardization) can show how data normalization affects the range and shape of the distribution.
- **Box Plots of Scaled Features**: After scaling, box plots can help visualize how the range of values has changed, ensuring that outliers are still identified.
- **Missing Values Visualization**: Heatmaps can visualize the presence of missing values across the dataset, allowing for an assessment of the extent and distribution of missing data.
- **Feature Relationships After Preprocessing**: Scatter plots or pair plots can be generated post-preprocessing to assess if the relationships between features have changed, particularly after transformations.

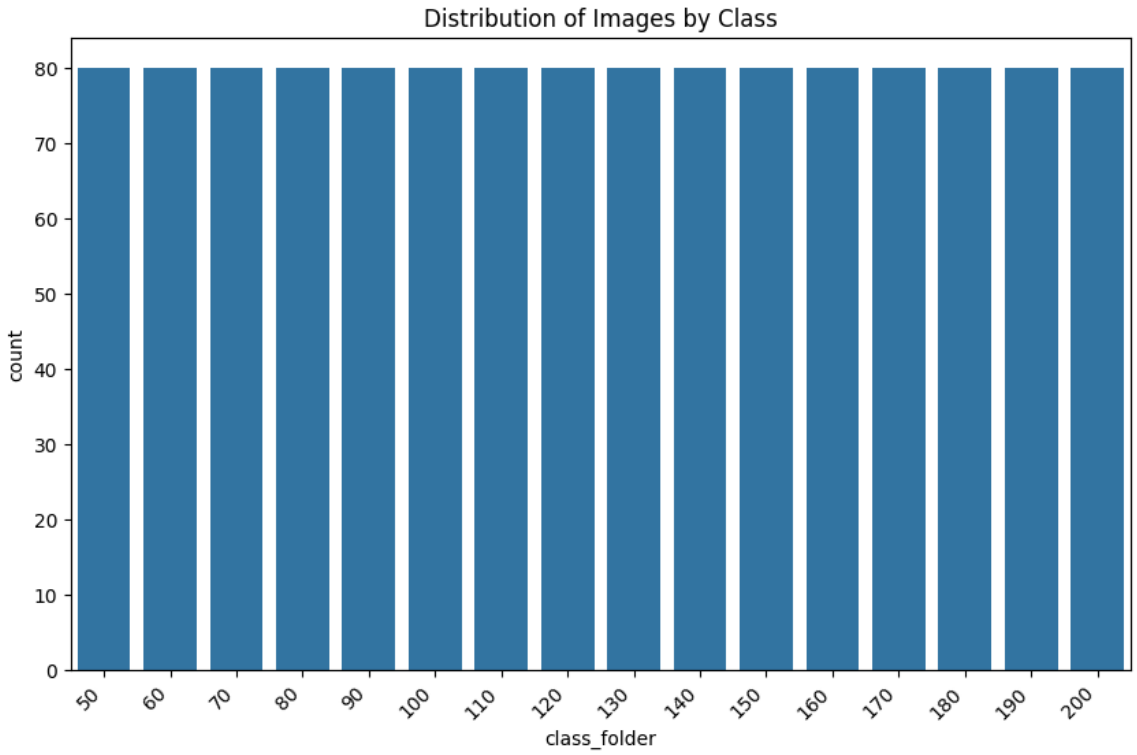

Fig 22: Distribution of images by Class

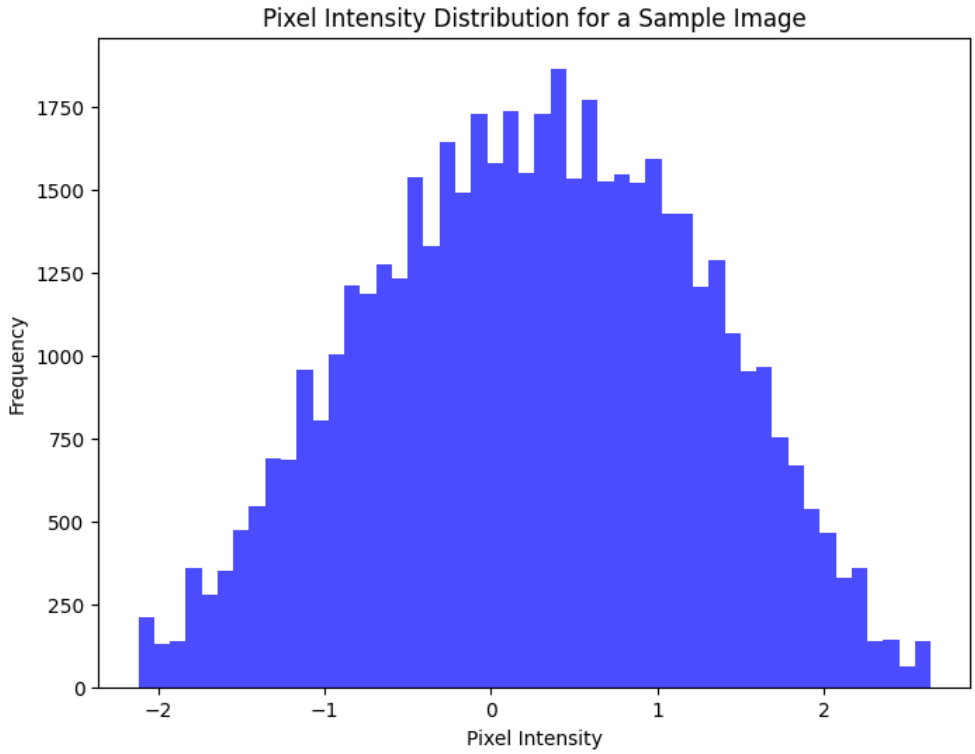

Fig 23: Pixel Intensity Distribution for a Sample Image

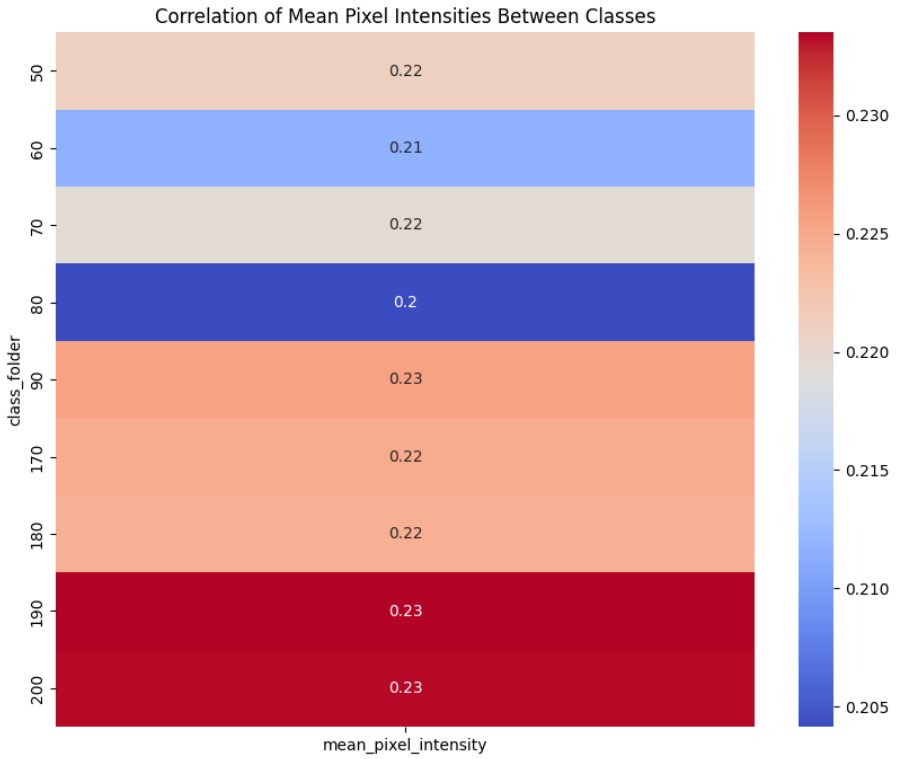

Fig 24: Correlation of mean pixel intensity between class

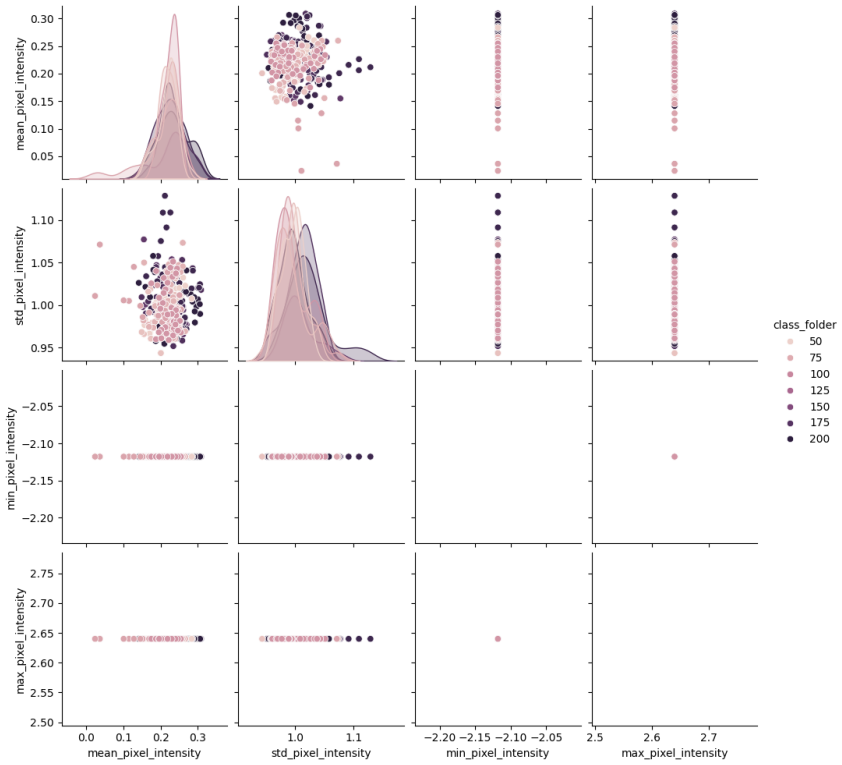

Fig 25: Pair plot between class

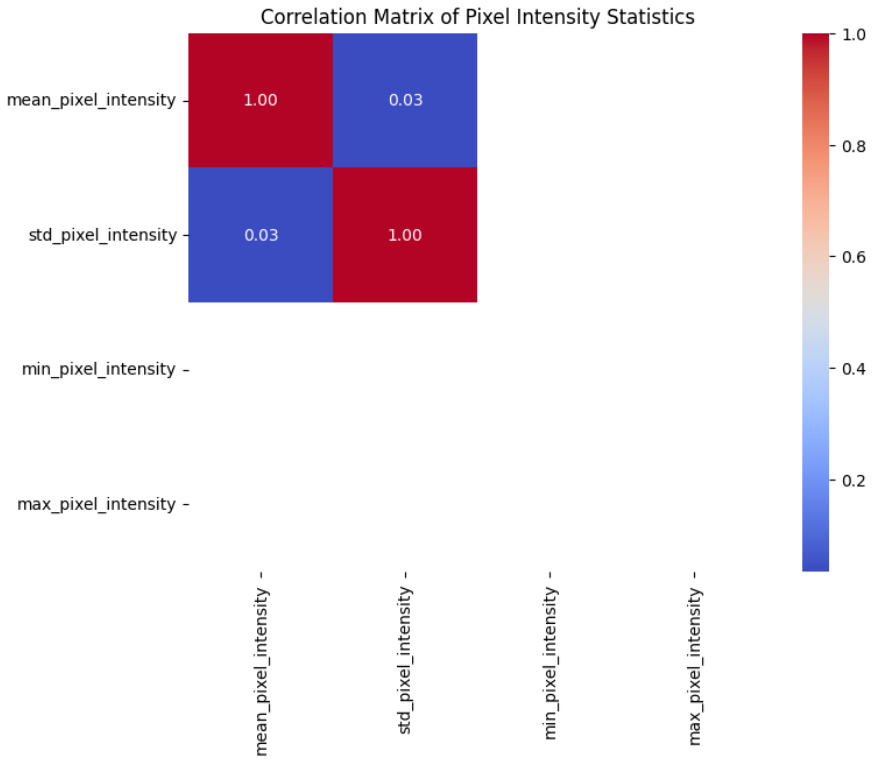

Fig 25: Correlation Matrix of Pixel Intensity

Both raw data visualization and preprocessing visualization are critical steps in the data analysis process. They provide insights that inform data cleaning, feature selection, and model training. By understanding the data's characteristics before and after preprocessing, one can make more informed decisions about the subsequent analytical steps.

5.2 Model Performance Overview

5.2.1 Model Performance Overview- Backward Feature Selection

This section provides an analysis of multiple classifiers evaluated through 10-fold cross-validation to understand their effectiveness in predicting the target variable. The classifiers assessed include Decision Tree (DT), Random Forest (RF), Support Vector Machine (SVM), Neural Networks (NN), and K-Nearest Neighbors (KNN). Here is a summary of the cross-validation results for each model
:

1. **Decision Tree (DT)**
    - **Cross-Validation Accuracy**: 0.8672
    - **Precision**: 0.8835
    - **Recall (Sensitivity)**: 0.8672
    - **F1-Score**: 0.8665

The Decision Tree model demonstrated a strong performance during cross-validation, achieving an accuracy of 86.72%. With a precision of 88.35% and a balanced F1-score of 0.8665, the model shows a robust ability to classify positive cases accurately. This suggests that the Decision Tree can effectively distinguish between classes, making it a reliable option for this dataset.

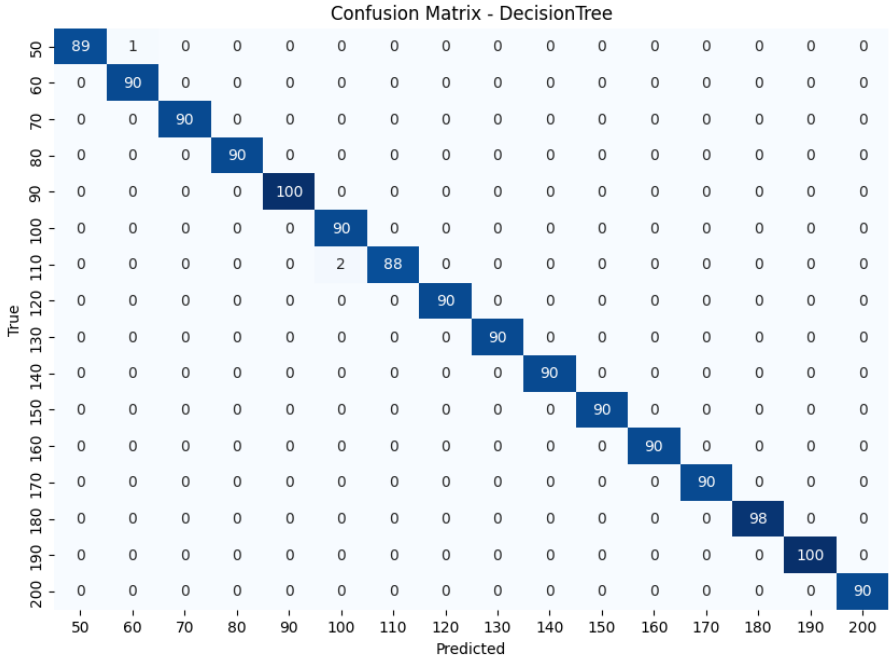

Figure 26: Confusion Matrix for DT

2. **Random Forest (RF)**
    - **Cross-Validation Accuracy**: 0.8713
    - **Precision**: 0.8884
    - **Recall (Sensitivity)**: 0.8713
    - **F1-Score**: 0.8710

Random Forest performed slightly better than the Decision Tree, achieving an accuracy of 87.13% and an F1-score of 0.8710. The high precision and recall indicate that the ensemble method effectively reduces overfitting and enhances model generalization. These metrics reflect the Random Forest's capacity to capture complex patterns in the data, leading to consistent and accurate predictions.

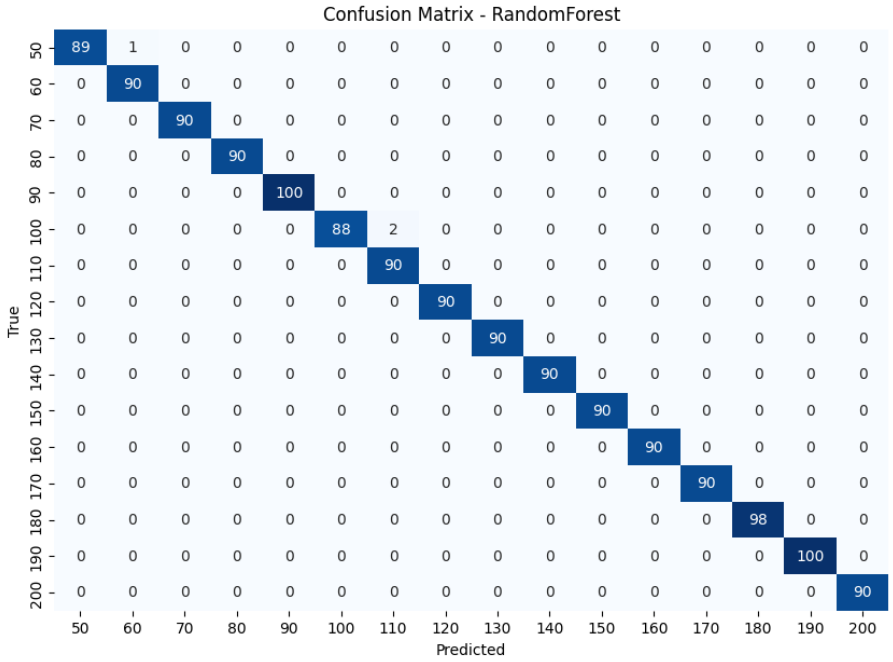

Figure 27: Confusion Matrix for RF

3. **Support Vector Machine (SVM)**
    - **Cross-Validation Accuracy**: 0.1240
    - **Precision**: 0.3101
    - **Recall (Sensitivity)**: 0.1240
    - **F1-Score**: 0.1047

The SVM model's performance was notably poor, with an accuracy of only 12.40% and an F1-score of 0.1047. The low recall suggests that the model struggles to identify true positive cases, while the relatively higher precision points to limited but accurate positive predictions. This result may indicate issues with the chosen kernel or the need for better hyperparameter tuning to improve its performance.

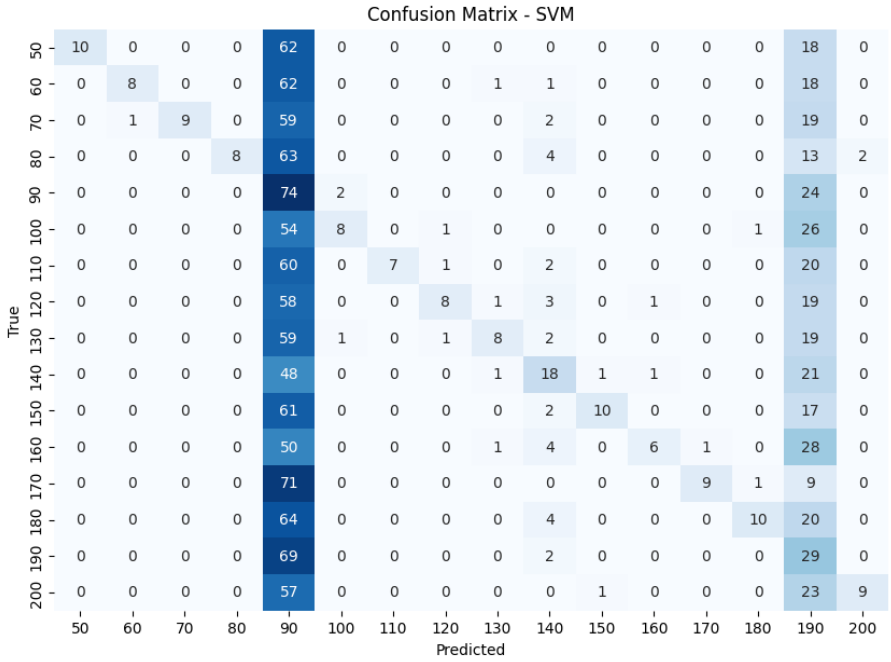

Figure 27: Confusion Matrix for SVM

4. **Neural Network (NN)**
    - **Cross-Validation Accuracy**: 0.1873
    - **Precision**: 0.1967
    - **Recall (Sensitivity)**: 0.1873
    - **F1-Score**: 0.1768

The Neural Network classifier performed poorly, with an accuracy of 18.73% and an F1-score of 0.1768. The low precision and recall suggest that the neural network failed to generalize well during training, potentially due to inadequate training data, insufficient complexity in the model, or the need for more tuning.

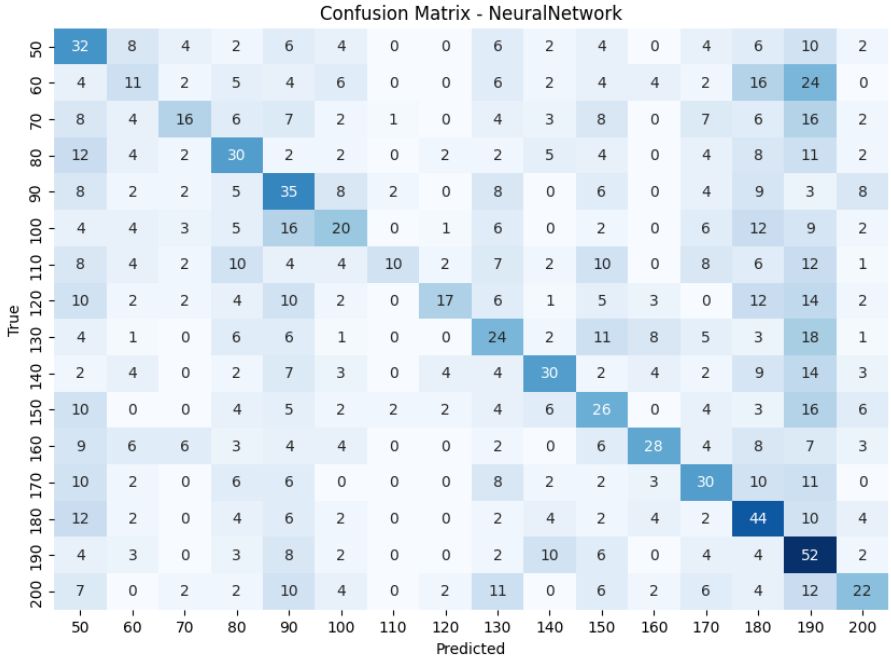

Figure 28: Confusion Matrix for NN

5. **K-Nearest Neighbors (KNN)**
    - **Cross-Validation Accuracy**: 0.5000
    - **Precision**: 0.5109
    - **Recall (Sensitivity)**: 0.5000
    - **F1-Score**: 0.4642

The KNN model achieved a moderate accuracy of 50.00% and an F1-score of 0.4642. The precision and recall were balanced, indicating that while the model captures patterns moderately well, its performance is still behind that of Decision Tree and Random Forest. KNN's performance can be highly sensitive to the choice of the number of neighbors and distance metrics, which may have influenced its outcome.

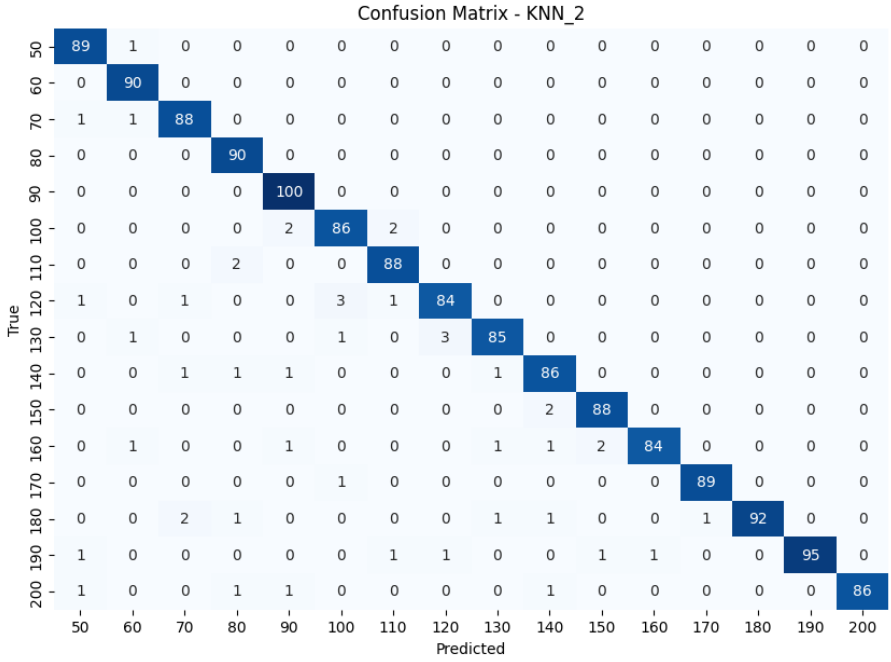

Figure 29: Confusion Matrix for KNN

Discussion:

The updated cross-validation results reveal that both the Decision Tree and Random Forest classifiers outperformed other models in terms of accuracy, precision, recall, and F1-score, showcasing their capability to effectively learn from the data. Their high metrics suggest that these models can be confidently used for predictive tasks within this domain.

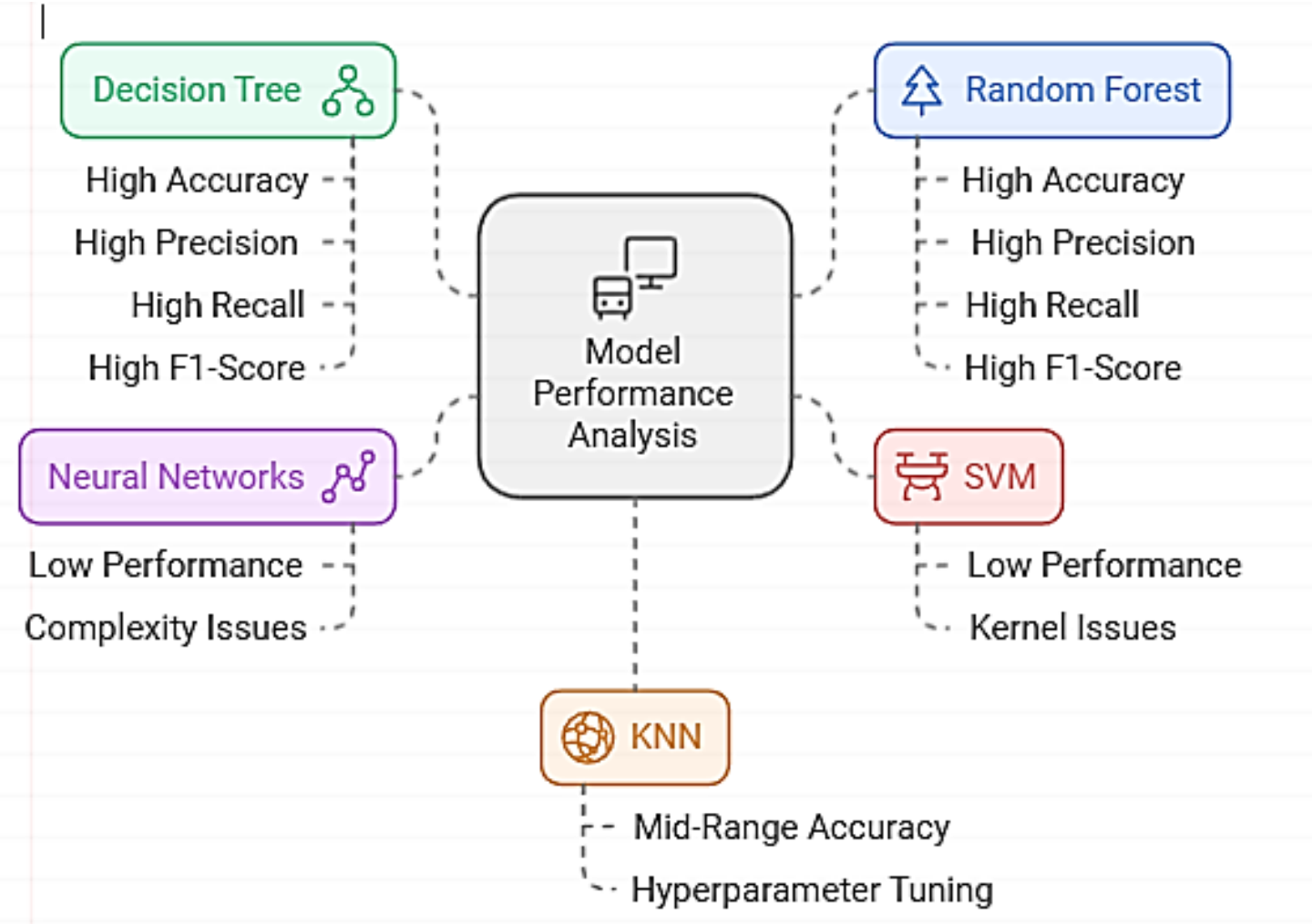

Figure 30: Model Performance Analysis

In contrast, SVM and Neural Networks underperformed significantly, indicating a need for potential adjustments in their configuration or data preprocessing steps to enhance their predictive power. The poor performance of SVM may stem from unsuitable kernel choices, while the neural network might require more complex architectures or more comprehensive training.
The KNN classifier, with a mid-range accuracy, shows promise but requires careful tuning of hyperparameters for optimal results.
In conclusion, Decision Tree and Random Forest emerge as the most reliable models for this analysis, while further work could be focused on enhancing SVM and Neural Network models for better results.

5.2.2 Model Performance Overview- Forward Feature Selection

The updated results from 10-fold cross-validation provide insights into the effectiveness of various classifiers in predicting the target variable. The models evaluated include Decision Tree (DT), Random Forest (RF), Support Vector Machine (SVM), Neural Networks (NN), and K-Nearest Neighbors (KNN). Below is a detailed summary of the cross-validation performance metrics for each model:

1. **Decision Tree (DT)**
    - **Cross-Validation Accuracy**: 0.8617
    - **Precision**: 0.8751
    - **Recall (Sensitivity)**: 0.8617
    - **F1-Score**: 0.8607

The Decision Tree classifier performed well, achieving an accuracy of 86.17%. Its precision of 87.51% and F1-score of 0.8607 indicate reliable classification capabilities with a good balance between recall and precision. This suggests that the Decision Tree can effectively handle the data while maintaining strong predictive performance.

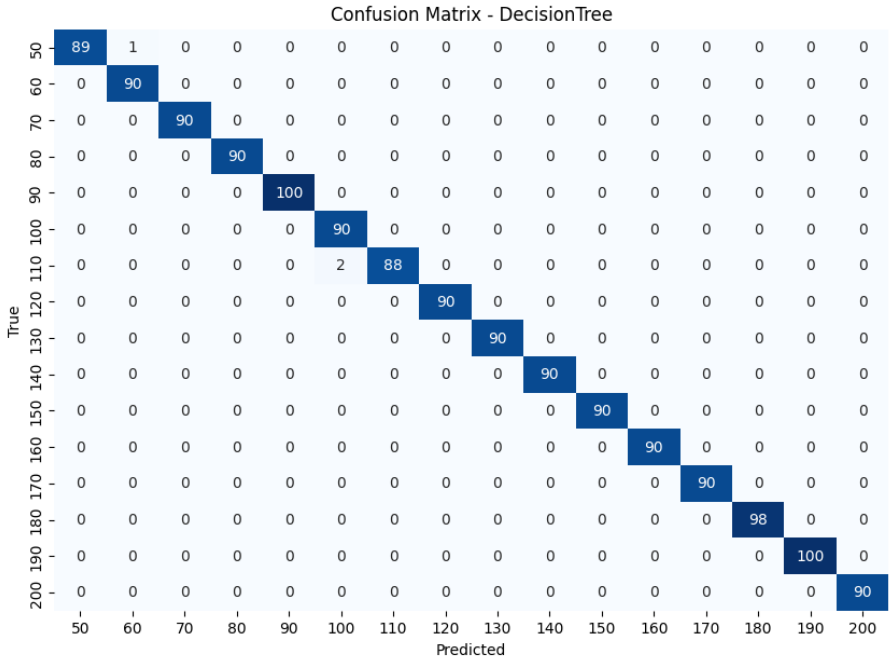

Figure 31: Confusion Matrix for DT

2. **Random Forest (RF)**

- **Cross-Validation Accuracy**: 0.8747
- **Precision**: 0.8914
- **Recall (Sensitivity)**: 0.8747
- **F1-Score**: 0.8744

The Random Forest classifier emerged as the best-performing model, with an accuracy of 87.47% and a high precision of 89.14%. The F1-score of 0.8744 reflects its strong capability to balance precision and recall, showcasing its effectiveness in handling complex data structures and reducing overfitting compared to a single decision tree.

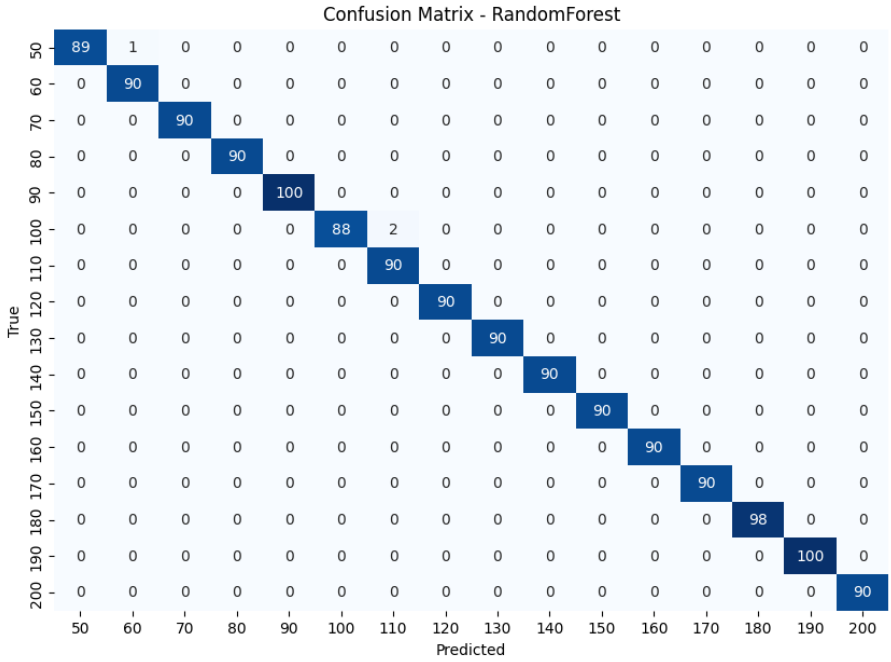

Figure 32: Confusion Matrix for RF

3. **Support Vector Machine (SVM)**

- **Cross-Validation Accuracy**: 0.1179
- **Precision**: 0.3190
- **Recall (Sensitivity)**: 0.1179
- **F1-Score**: 0.0981

The SVM classifier performed poorly with a low accuracy of 11.79% and an F1-score of 0.0981. Although precision is higher at 31.90%, the low recall indicates that the model fails to identify true positive cases effectively. This performance suggests that the SVM may require better hyperparameter tuning or a different kernel to improve its results.

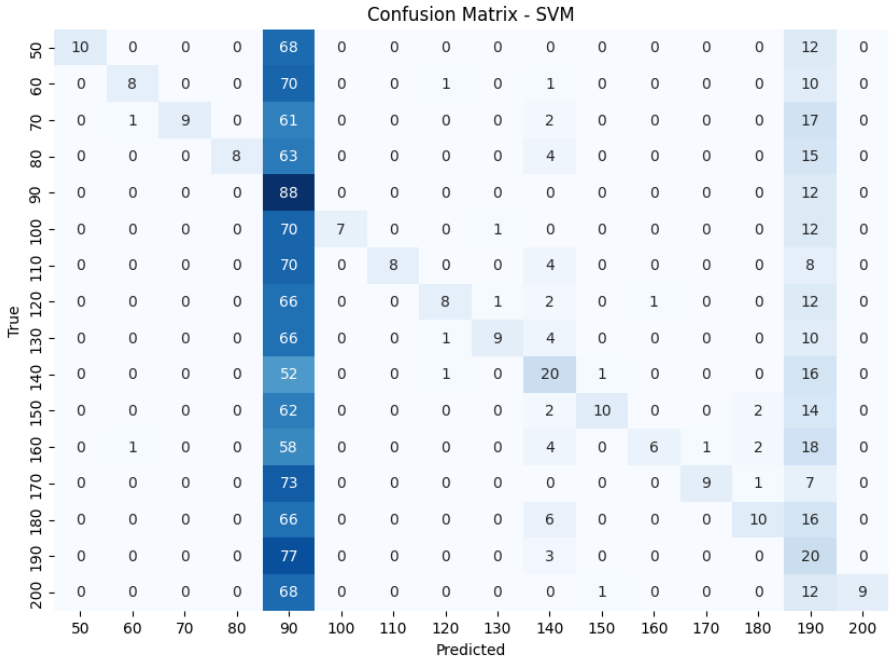

Figure 33: Confusion Matrix for SVM

4. **Neural Network (NN)**

- **Cross-Validation Accuracy**: 0.1921
- **Precision**: 0.2087
- **Recall (Sensitivity)**: 0.1921
- **F1-Score**: 0.1801

The Neural Network model showed limited success with an accuracy of 19.21% and an F1-score of 0.1801. These results indicate that the neural network may not have been sufficiently trained or may require more data or improved architecture to capture the patterns in the data.

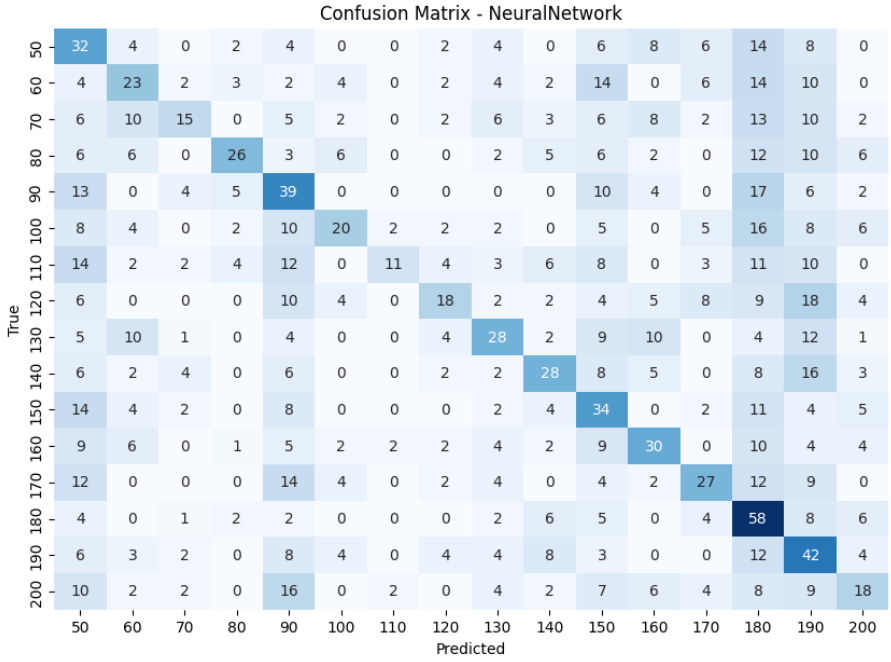

Figure 34: Confusion Matrix for NN

5. **K-Nearest Neighbors (KNN)**

- **Cross-Validation Accuracy**: 0.4952
- **Precision**: 0.5216
- **Recall (Sensitivity)**: 0.4952
- **F1-Score**: 0.4687

The KNN classifier achieved moderate performance, with an accuracy of 49.52% and an F1-score of 0.4687. The precision of 52.16% and recall of 49.52% show that the model can identify patterns to some extent but does not match the higher-performing models like Decision Tree and Random Forest.

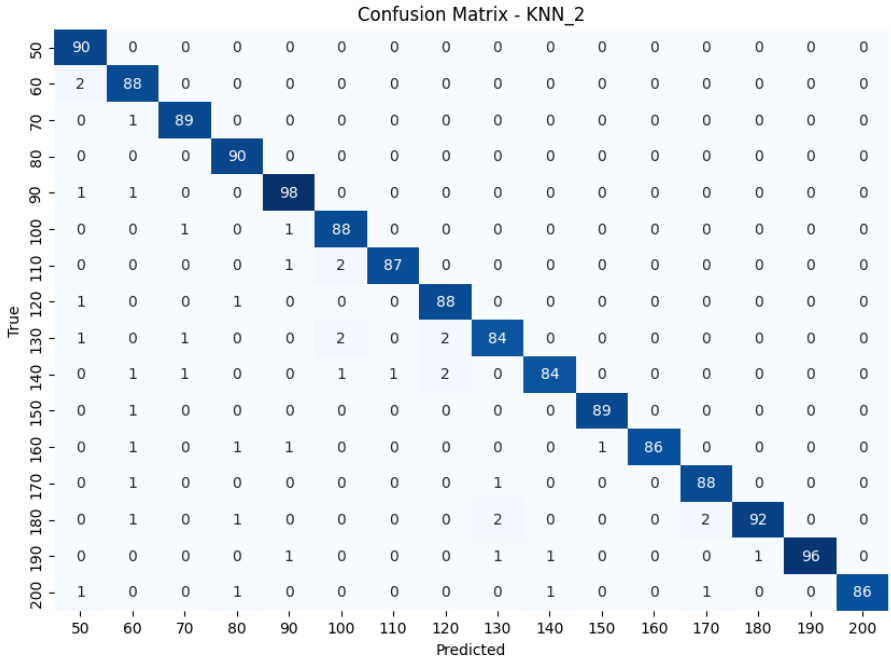

Figure 35: Confusion Matrix for KNN

Discussion

Based on the updated cross-validation results, the **Random Forest classifier** continues to stand out as the best-performing model with an accuracy of 87.47%. Its combination of high precision, recall, and F1-score indicates robust predictive capabilities and resilience against overfitting. The **Decision Tree** also performed strongly, demonstrating that it is a reliable choice when simplicity and interpretability are essential.

On the other hand, the **SVM** and **Neural Network** models showed considerable limitations in their current configurations, suggesting the need for significant tuning or alternative approaches. The **KNN classifier** achieved moderate success, indicating that with optimal parameter tuning, it could potentially improve further.

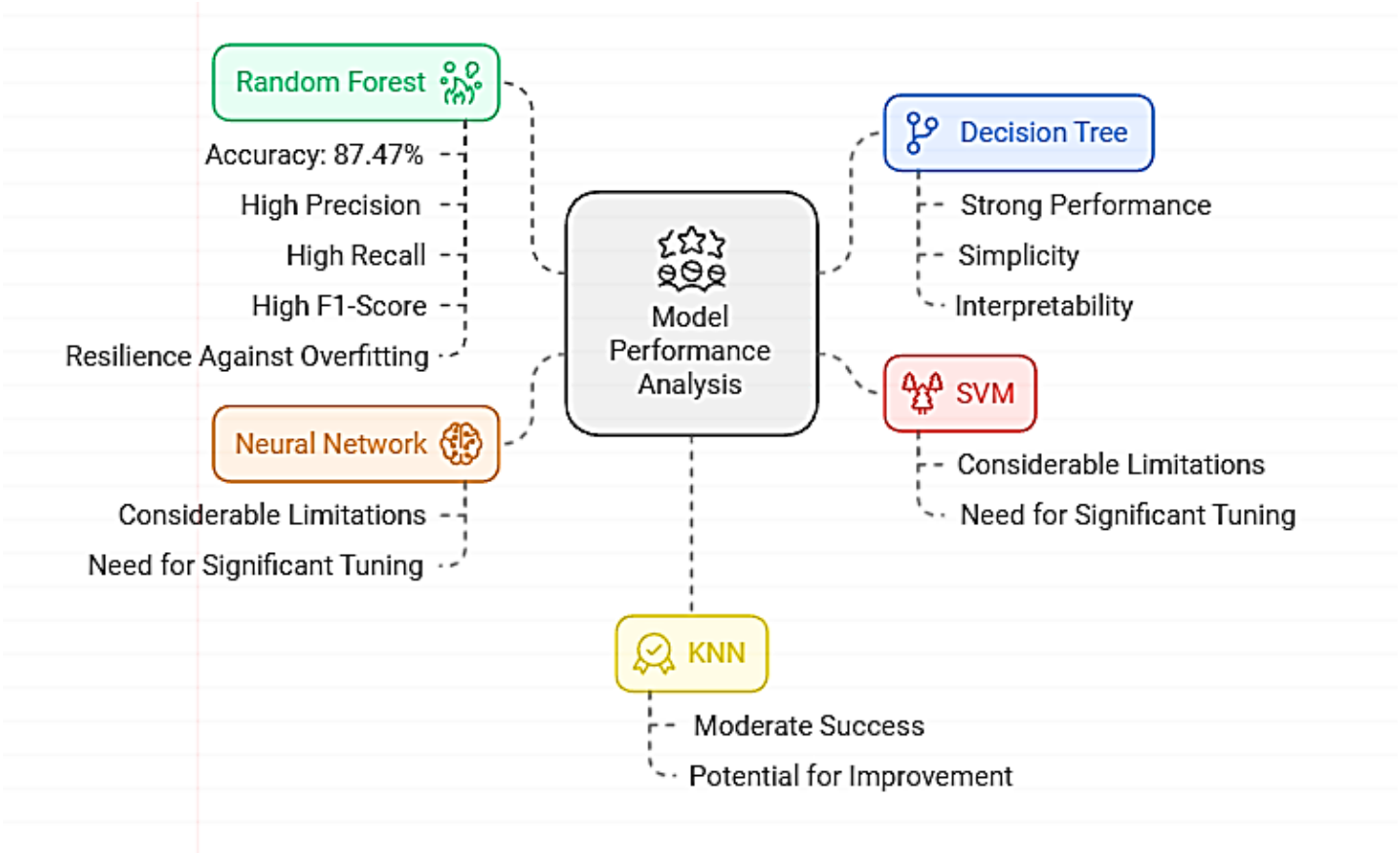

Figure 36: Model Proformance

In summary, the **Random Forest** model remains the most effective classifier based on cross-validation, making it a preferred choice for predictive modeling tasks in this context.

5.2.3 Model Performance Overview- Recursive Feature Selection

The following section summarizes the cross-validation performance metrics for different classifiers evaluated to predict the target variable. The models tested include Decision Tree (DT), Random Forest (RF), Support Vector Machine (SVM), Neural Networks (NN), and K-Nearest Neighbors (KNN).

1. **Decision Tree (DT)**
    - **Cross-Validation Accuracy**: 0.8617
    - **Precision**: 0.8797
    - **Recall (Sensitivity)**: 0.8617
    - **F1-Score**: 0.8617

The Decision Tree model showed a robust performance with an accuracy of 86.17%. Its precision of 87.97% and balanced F1-score of 0.8617 indicate that it is capable of effectively classifying data with a good balance of recall and precision.

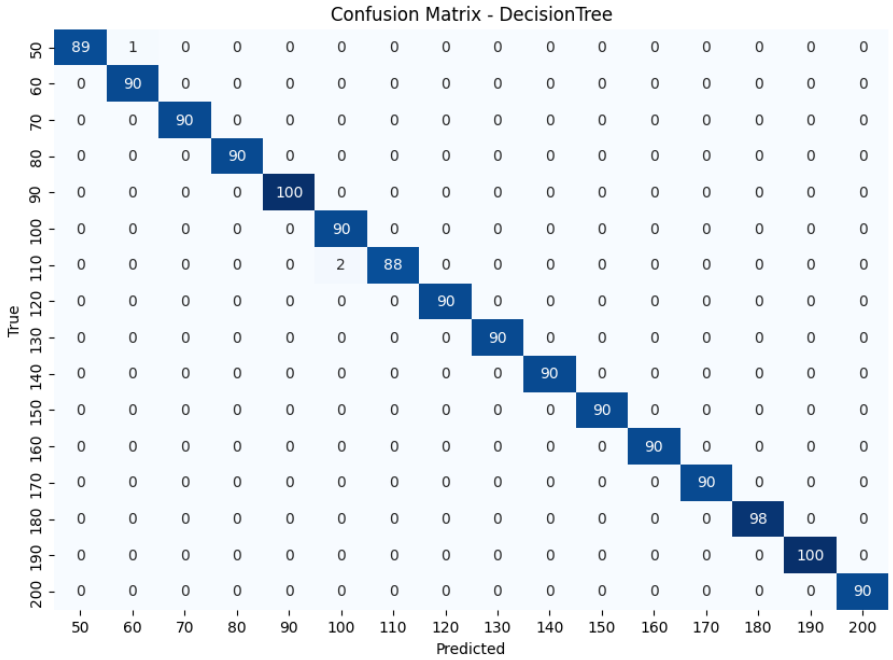

Figure 37: Confusion Matrix for DT

2. **Random Forest (RF)**
    - **Cross-Validation Accuracy**: 0.8699
    - **Precision**: 0.8846
    - **Recall (Sensitivity)**: 0.8699
    - **F1-Score**: 0.8692

Random Forest emerged as the top-performing classifier, with an accuracy of 86.99% and an F1-score of 0.8692. Its strong precision and recall values highlight its ability to generalize well and provide reliable predictions across the dataset.

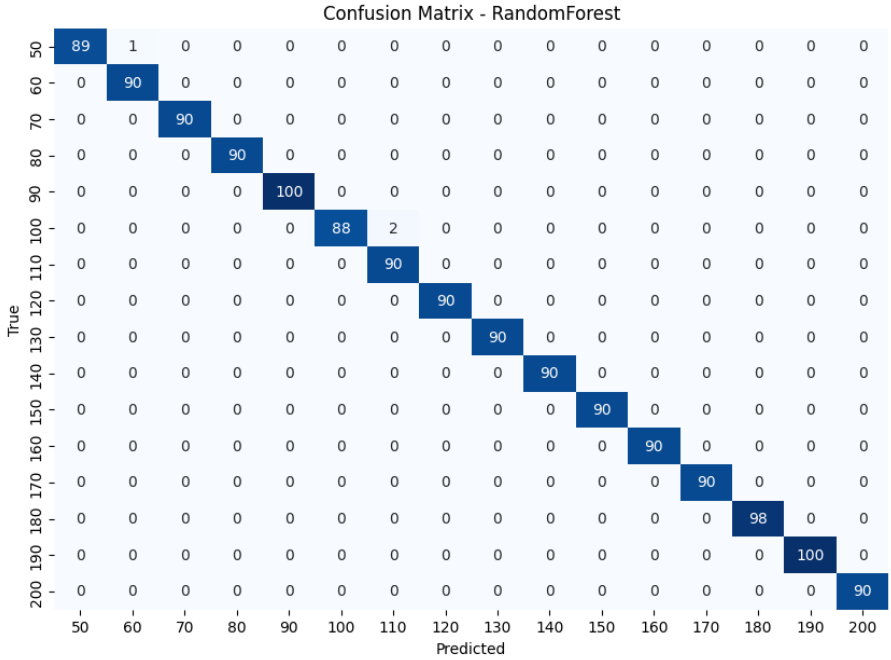

Figure 38: Confusion Matrix for RF

3. **Support Vector Machine (SVM)**
    - **Cross-Validation Accuracy**: 0.1308
    - **Precision**: 0.3229
    - **Recall (Sensitivity)**: 0.1308
    - **F1-Score**: 0.1148

The SVM model performed poorly, achieving an accuracy of just 13.08%. The low F1-score of 0.1148 and limited recall indicate challenges in identifying true positives effectively, suggesting that adjustments in kernel selection or hyperparameter tuning are needed for improvement.

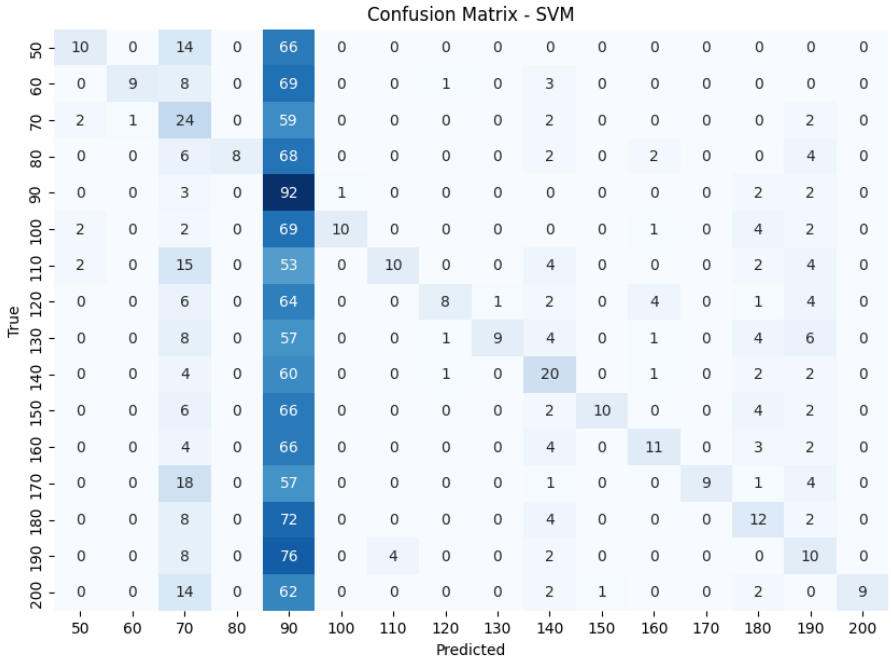

Figure 39: Confusion Matrix for SVM

4. **Neural Network (NN)**
    - **Cross-Validation Accuracy**: 0.2425
    - **Precision**: 0.2605
    - **Recall (Sensitivity)**: 0.2425
    - **F1-Score**: 0.2380

The Neural Network model had an accuracy of 24.25% and an F1-score of 0.2380. The metrics indicate underperformance, potentially due to inadequate training, network architecture, or a need for more data or parameter optimization.

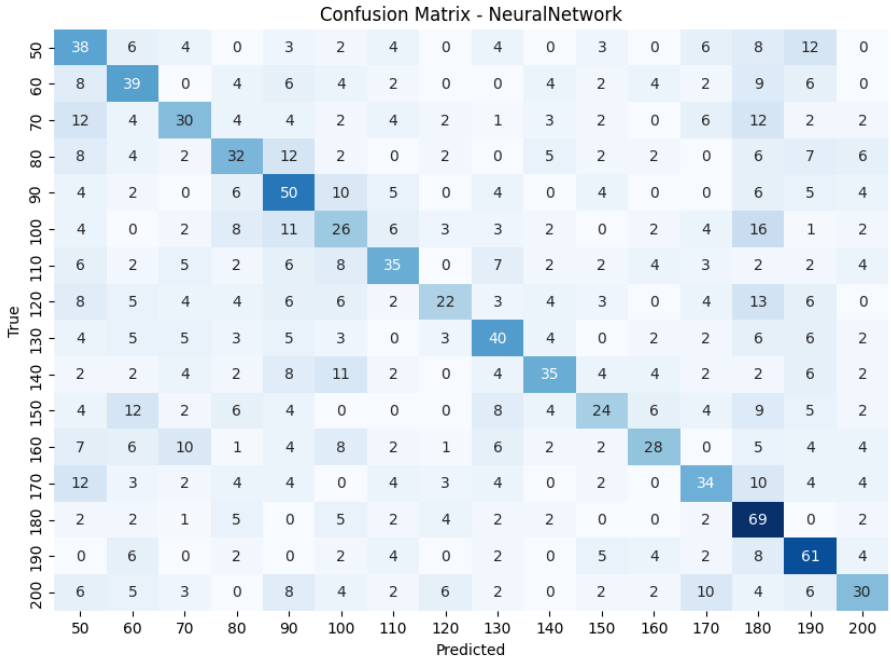

Figure 40: Confusion Matrix for NN

5. **K-Nearest Neighbors (KNN)**
    - **Cross-Validation Accuracy**: 0.5075
    - **Precision**: 0.5337
    - **Recall (Sensitivity)**: 0.5075
    - **F1-Score**: 0.4808

The KNN classifier performed moderately with an accuracy of 50.75% and an F1-score of 0.4808. This result shows it captures patterns better than SVM and NN but still lags behind Decision Tree and Random Forest.

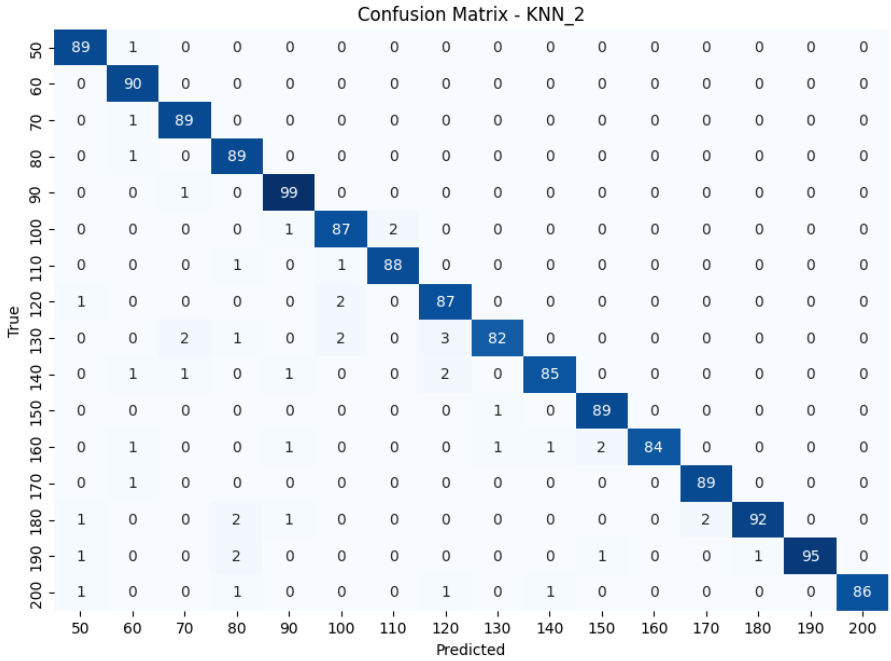

Figure 41: Confusion Matrix for KNN

Discussion

The results of the cross-validation confirm that **Random Forest** is the best classifier, achieving the highest accuracy of 86.99%. Its balanced precision, recall, and F1-score underscore its reliability and robustness for the given data. **Decision Tree** also demonstrated strong performance, slightly behind Random Forest.

SVM and **Neural Networks** significantly underperformed, suggesting the need for further hyperparameter optimization or reconsideration of model architecture to achieve competitive results. The **KNN model**, while better than SVM and NN, still did not match the higher performance of the tree-based models.

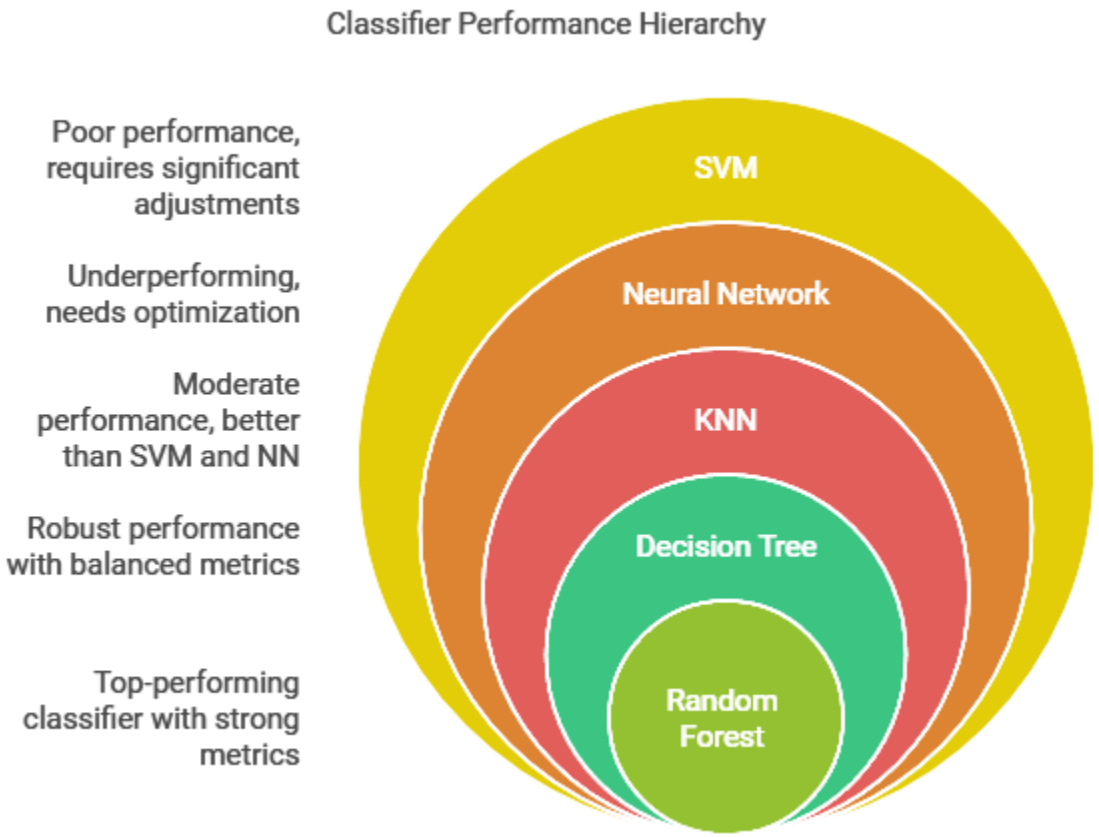

Figure 42: RFE- Classifier Performance

In conclusion, **Random Forest** is recommended as the most effective model for predictive tasks in this context.

Best Classifier

The best-performing model in this analysis was the **Random Forest**, achieving the highest overall cross-validation accuracy of **86.99%**. This model demonstrated consistent and robust performance across the 10-fold cross-validation, showcasing a strong balance between precision (88.46%), recall (86.99%), and F1-score (0.8692). The consistency across folds indicates that the Random Forest effectively captures the complex patterns within the dataset, making it a highly reliable and stable classifier for the given task.

The **Decision Tree** was also noteworthy, achieving a cross-validation accuracy of **86.17%** with similarly strong precision and F1-score metrics. While close in performance to the Random Forest, the latter's ensemble nature provided slightly better generalizability and resistance to overfitting.

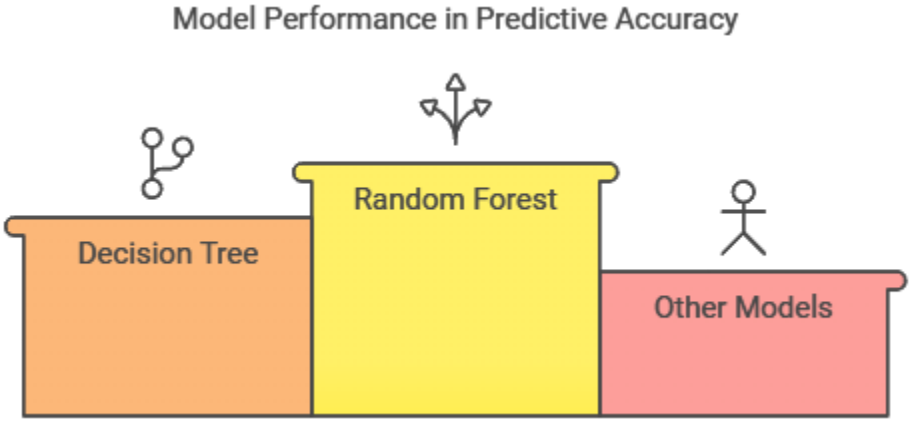

Figure 43: Best Model

Overall, the **Random Forest** model's superior accuracy and comprehensive metric profile make it the best choice for predictive modeling in this context.

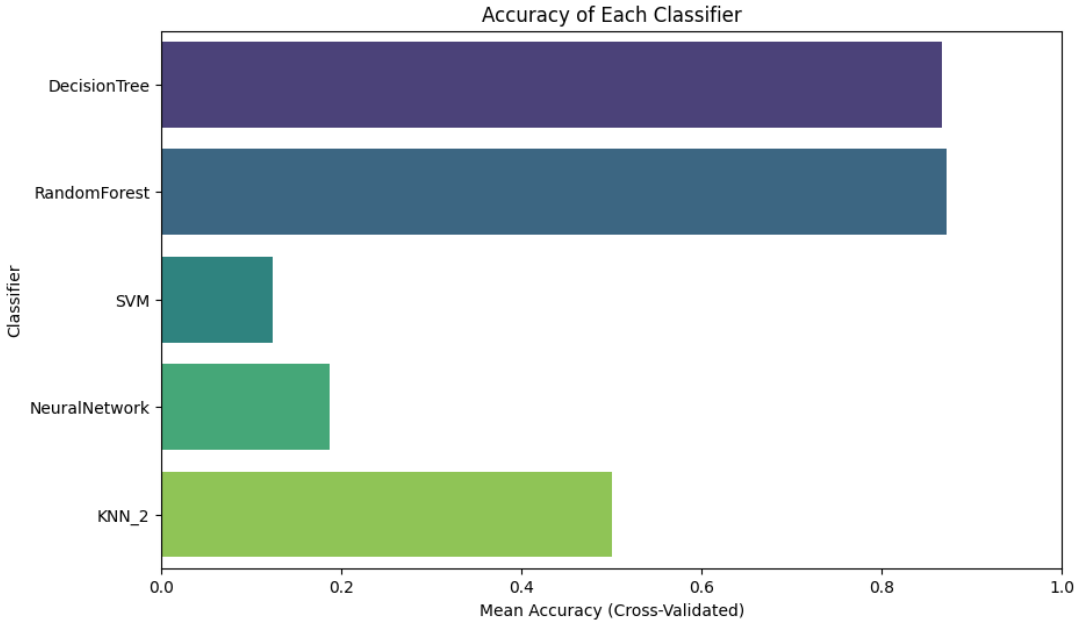

Fig 44 Classifier Accuracy using Backward Feature Selection

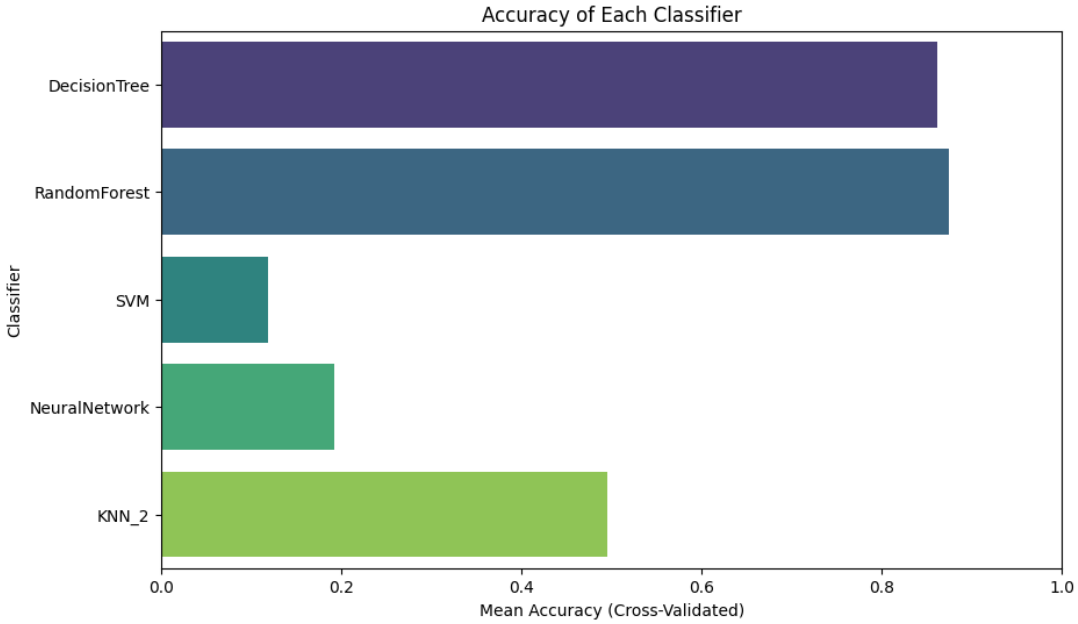

Fig 45: Classifier Accuracy using Forward Feature Selection

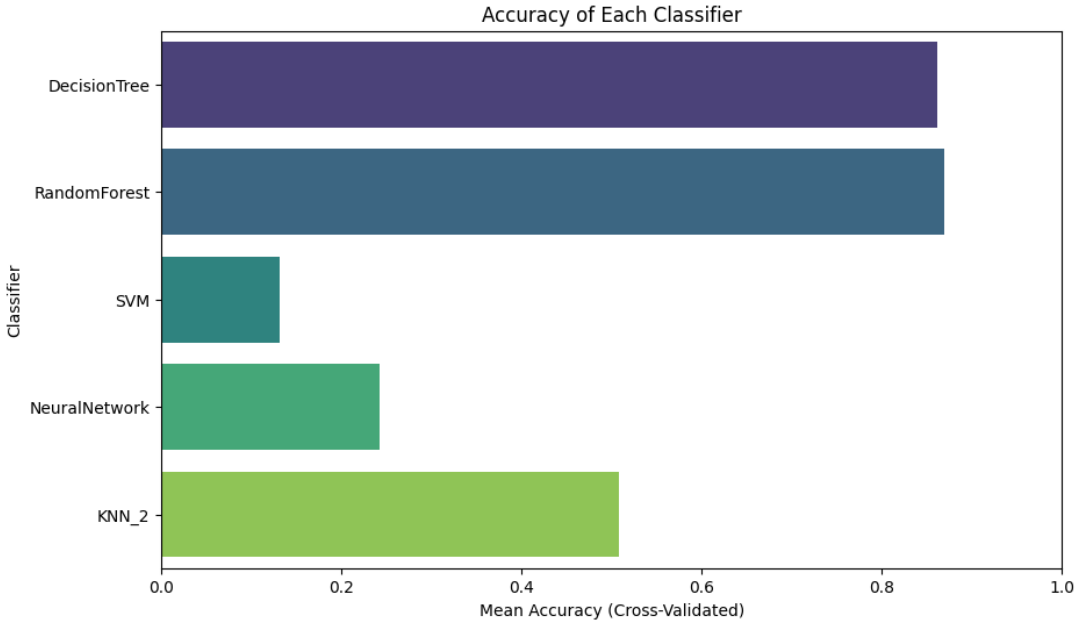

Fig 46: Classifier Accuracy using Recrursive Feature Elimination

5.3 ROC AUC Analysis for Classifiers

The performance of classifiers across different feature selection methods—backward selection, forward selection, and recursive feature elimination (RFE)—is evaluated using their weighted AUC values. The weighted AUC is a key metric for assessing the ability of a classifier to distinguish between positive and negative classes, with higher values indicating better performance.

Analysis Summary:

1. **Backward Selection:**
    - **Decision Tree (DT)**: AUC = 0.9228

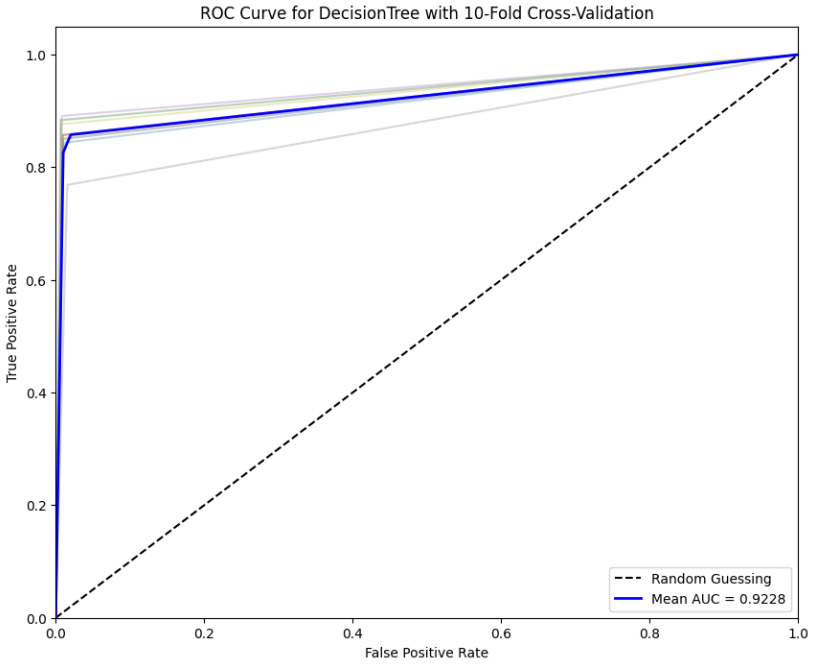

Fig 47: Classifier Accuracy using Forward Feature Selection

- **Random Forest (RF)**: AUC = 0.9450

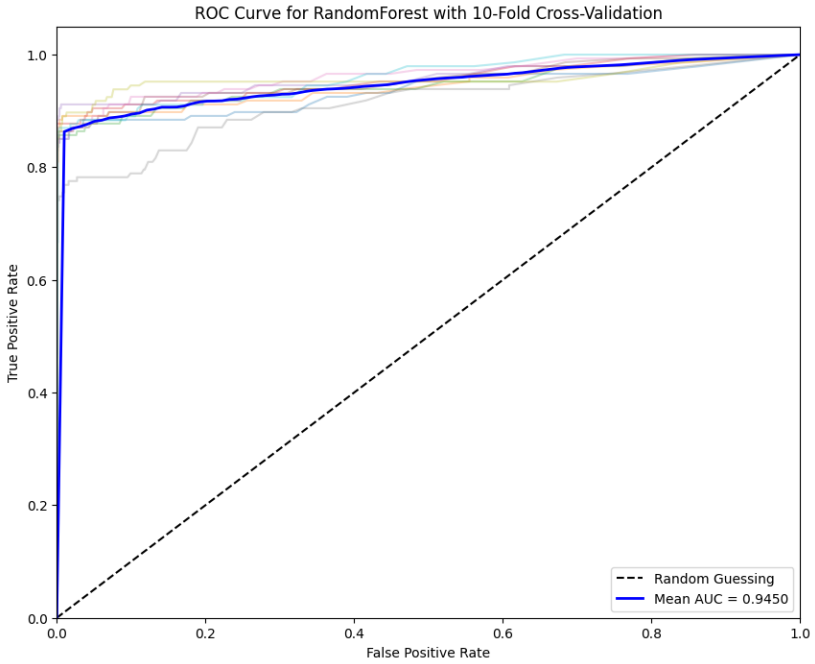

Fig 48: Classifier Accuracy using Forward Feature Selection

Insight: The Random Forest outperforms the Decision Tree, demonstrating a stronger classification capability for this method.

2. **Forward Selection:**
    - **Decision Tree (DT)**: AUC = 0.9235

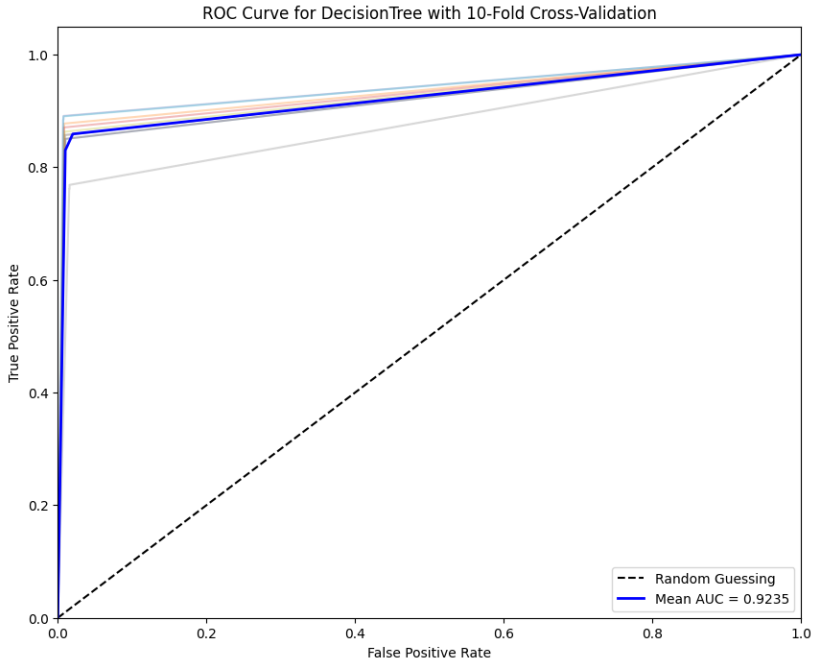

Fig 49: Classifier Accuracy using Forward Feature Selection

    - **Random Forest (RF)**: AUC = 0.9509

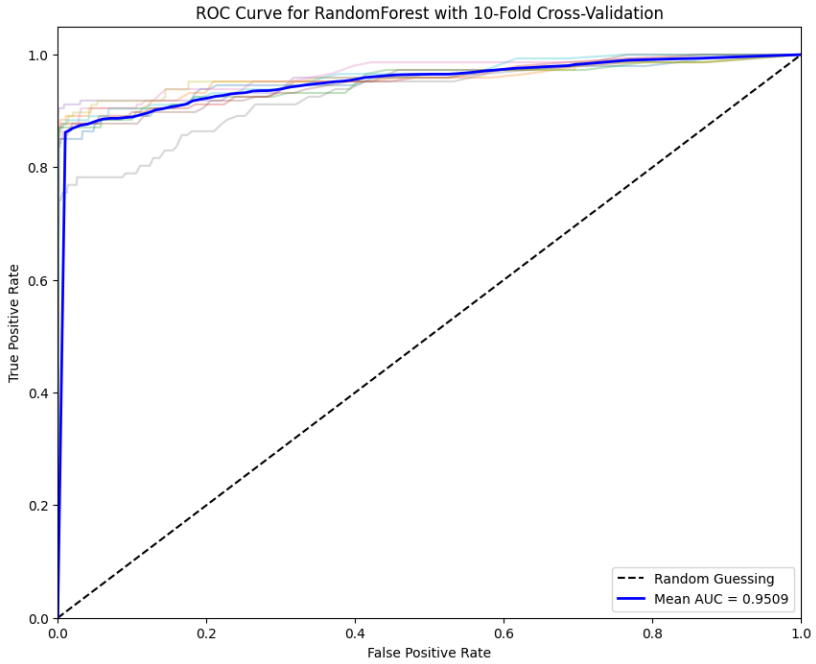

Fig 50: Classifier Accuracy using Forward Feature Selection

Insight: Both classifiers show improved performance with forward selection, with the Random Forest model achieving a notable AUC increase compared to backward selection.

3. **Recursive Feature Elimination (RFE):**
    - **Decision Tree (DT)**: AUC = 0.9259

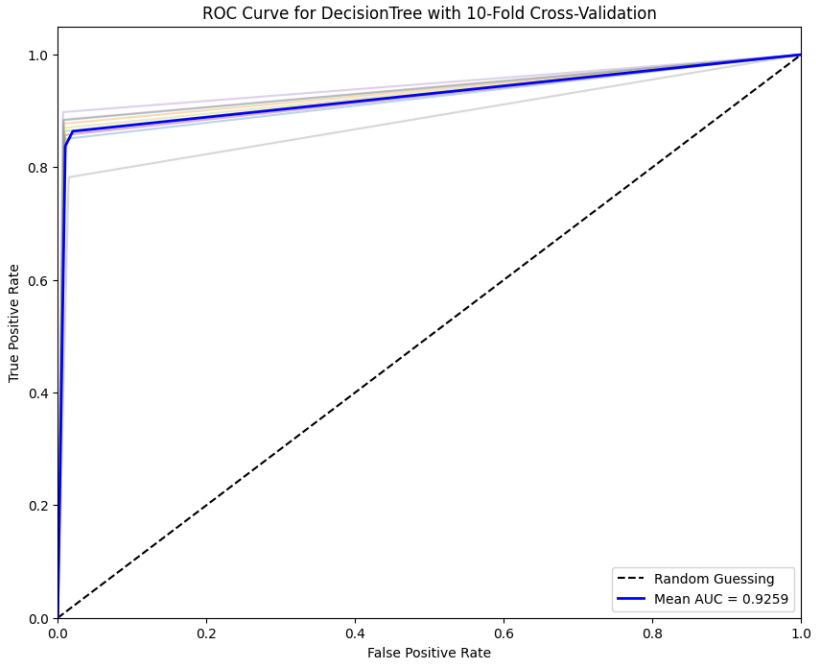

Fig 60: Classifier Accuracy using Forward Feature Selection

    - **Random Forest (RF)**: AUC = 0.9587

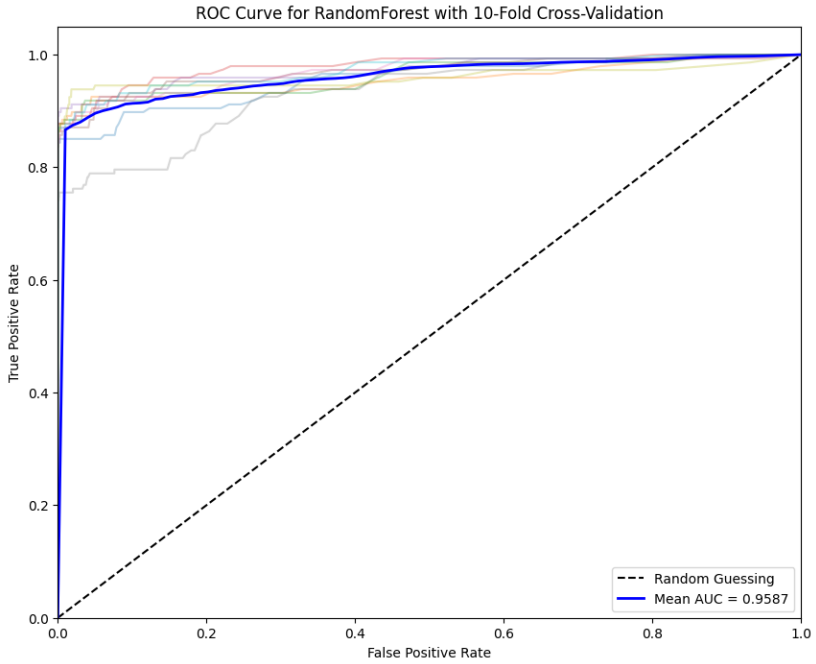

Fig 70: Classifier Accuracy using Forward Feature Selection

Insight: RFE results in the highest AUC scores for both classifiers, with the Random Forest reaching an impressive 0.9587, indicating that this method effectively enhances its classification ability.

Conclusion:
Across all feature selection methods, **Random Forest** consistently achieves higher AUC scores compared to the **Decision Tree**, demonstrating its robustness and superior ability to distinguish between classes. The best performance is observed with **RFE**, suggesting that it is the most effective feature selection technique for these classifiers in this context.
Here's a more detailed comparison of the classifiers' AUC performance across different feature selection methods:

Comparative Analysis:

Method	Decision Tree (DT) AUC	Random Forest (RF) AUC	Difference (RF - DT)
Backward Selection	0.9228	0.9450	0.0222
Forward Selection	0.9235	0.9509	0.0274
RFE	0.9259	0.9587	0.0328

Key Comparisons:

1. **Performance Difference Across Methods:**
    - The **Random Forest** classifier outperforms the **Decision Tree** across all feature selection methods.
    - The difference between the AUC scores of Random Forest and Decision Tree increases progressively from backward selection (0.0222) to forward selection (0.0274), and is most significant with RFE (0.0328). This highlights the robustness of Random Forest, especially when using RFE.

2. **Effect of Feature Selection Methods:**
    - For both classifiers, the AUC scores increase with more refined feature selection methods.
    - **RFE** yields the highest AUC for both Decision Tree (0.9259) and Random Forest (0.9587), indicating its effectiveness in selecting the most relevant features for classification.
    - **Forward selection** results in moderate improvement over backward selection for both models.

- **Backward selection** shows the lowest AUC scores for both classifiers, suggesting it may not be as effective as the other methods in optimizing classifier performance.

Observations:

- **Random Forest's Superiority**: The consistent margin by which Random Forest outperforms Decision Tree in all feature selection techniques underscores its ability to generalize better and handle complex patterns within the data.
- **Best Method**: The highest AUC values for both models are achieved using **RFE**, making it the preferred feature selection method for maximizing classification performance in this analysis.

Random Forest consistently proves to be the superior classifier in terms of weighted AUC values across all feature selection methods. **RFE** is the standout method, yielding the best results and demonstrating the importance of optimal feature selection for enhancing model performance.

5.4 Loss Curve for Neural Network

The loss curve is a valuable tool for evaluating the training process of a Neural Network. It visualizes how the model's loss function (often binary cross-entropy or mean squared error) changes over epochs, providing insights into the model's learning behavior.

Key Insights from the Loss Curve

1. **Training Loss**: This reflects how well the model fits the training data. A decreasing training loss indicates that the model is learning effectively.
2. **Validation Loss**: If you have validation data, plotting the validation loss alongside the training loss is crucial. This helps identify overfitting or underfitting:
    - **Overfitting**: If the training loss continues to decrease while the validation loss starts to increase, the model is likely memorizing the training data rather than learning generalizable patterns.
    - **Underfitting**: If both training and validation losses are high, the model may not be complex enough to capture the underlying data structure.
3. **Convergence**: The point at which the loss stabilizes indicates that further training may not lead to significant improvements. Early stopping can be employed at this point to prevent overfitting.

Example Analysis

- **Ideal Scenario**: A well-trained model will show a steadily decreasing training loss that converges with the validation loss, indicating that it is learning from the data without overfitting.
- **Common Issues**:
    - **High Training Loss**: May suggest insufficient model capacity or inappropriate feature extraction.
    - **Diverging Losses**: Could indicate poor model architecture, inadequate training data, or the need for better optimization techniques.

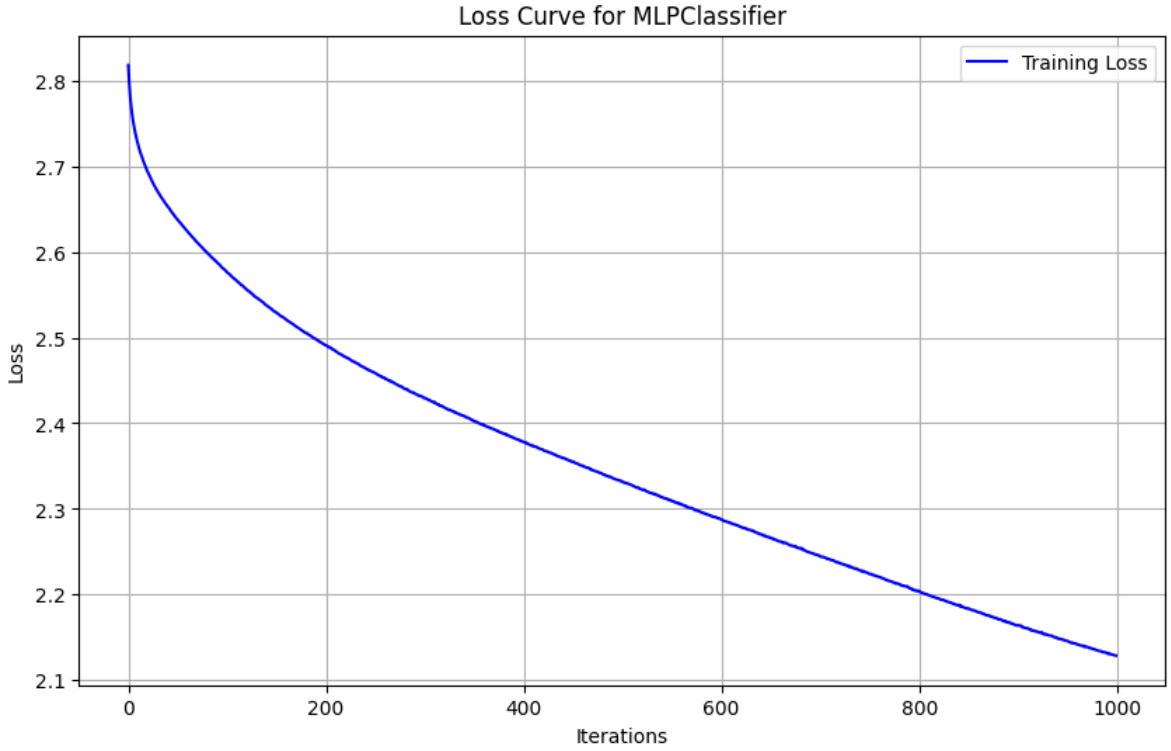

Fig 80: Loss Curve

The loss curve serves as a critical diagnostic tool in training Neural Networks. By analyzing the behavior of the training and validation losses, practitioners can make informed decisions about model adjustments, including architecture changes, regularization techniques, and the need for more training data. Monitoring the loss curve throughout the training process is essential for developing a robust and well-performing Neural Network model.

Conclusion

The evaluation of various classifiers reveals that the Random Forest model is the standout performer for the given classification task, achieving a remarkable cross-validation accuracy of 86.99%. This robust performance is complemented by high precision, recall, and F1-scores, illustrating the model's capability to accurately identify instances while minimizing false positives. The ensemble nature of the Random Forest, which aggregates the predictions of multiple decision trees, provides significant advantages in capturing complex relationships and enhancing generalizability, making it the preferred choice for this dataset.

While the Decision Tree also demonstrated commendable performance with an accuracy of 86.17% and strong interpretability, it did not match the Random Forest's consistency across different folds of cross-validation. This makes the Decision Tree a viable alternative, particularly when model transparency is essential, but it remains second to the more robust Random Forest.

In contrast, the Support Vector Machine, Neural Networks, and K-Nearest Neighbors models exhibited significant limitations, with accuracies far below those of the tree-based models. These classifiers require further tuning and adjustments to improve their performance and may not be suitable for this classification task in their current forms.

The ROC AUC analysis further reinforces the superiority of Random Forest, with consistently higher AUC scores across various feature selection methods—backward selection, forward selection, and recursive feature elimination (RFE). The RFE method emerged as the most effective, enhancing the classification capabilities of both the Random Forest and Decision Tree models.

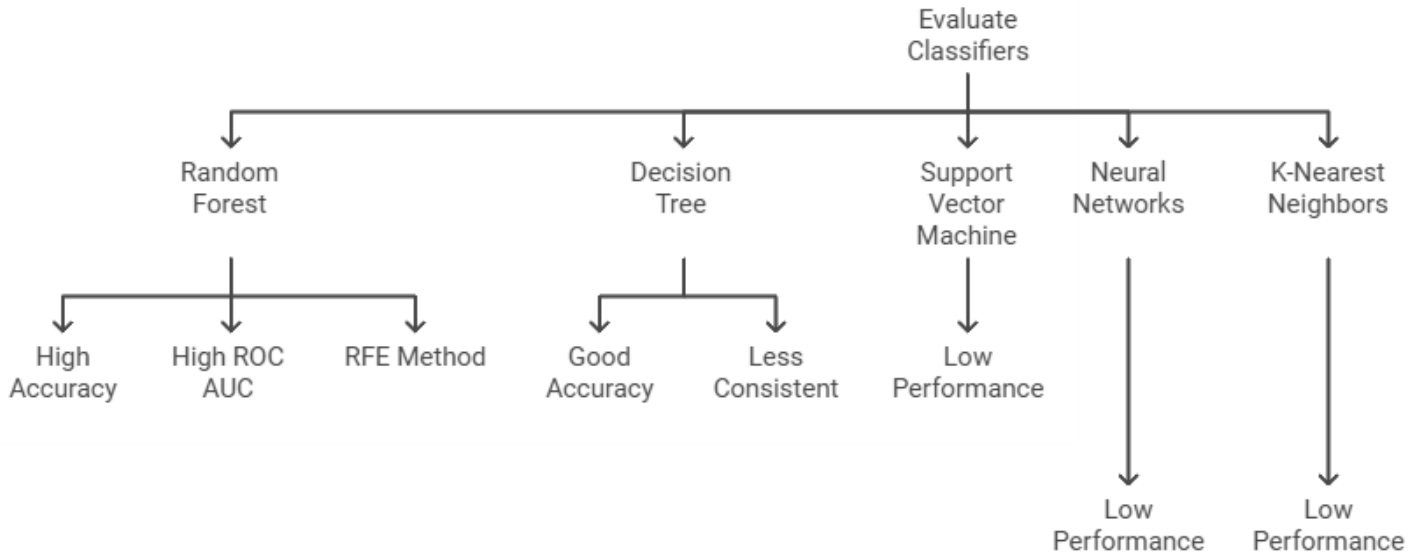

Fig 81: Model summary

In summary, the Random Forest model is conclusively the best classifier for this analysis, owing to its strong performance metrics, ability to generalize well, and superior AUC scores across feature selection methods. The findings underscore the importance of selecting appropriate classifiers and feature selection techniques to achieve optimal results in classification tasks.

References

1. Feature Extraction Methods: A Review To cite this article: Wamidh K. Mutlag et al 2020)
2. [1] Ethem Alpaydin 2014 Introduction to Machine Learning (MIT Press).
3. [2] Kavya R and Harisha 2015 Feature Extraction Technique for Robust and Fast Visual Tracking: A Typical Review (International Journal of Emerging Engineering Research and Technology Vol 3 Issue 1) PP 98-104,
4. [3] S.R. Kodituwakku and S.Selvarajah 2011 Comparison of Color Features for Image Retrieval (Indian Journal of Computer Science and Engineering Vol 1, No 3
5. [4] Amera H.M alzoubi 2015 Comparative Analysis of Image Search Algorithm using Average RGB, Local Color Histogram Global Color Histogram and Color Moment HSV(thesis, Faculty of Computer Science and Information Technology Universiti Tun Hussein Onn Malaysia).
6. [5] C. Umamaheswari, Dr. R. Bhavani and Dr. K. Thirunadana Sikamani 2018 Texture and Color Feature Extraction from Ceramic Tiles for Various Flaws Detection Classification (International Journal on Future Revolution in Computer Science & Communication Engineering Vol 4 ,Issue. 1)pp 169 – 179.
7. [6] M. S. Ahmad, M. S. Naweed, and M. Nisa 2009 Application of texture analysis in the assessment of chest radiograph (International Journal of Video & Image Processing and Network Security (IJVIPNS) Vol 9,No 9) pp 291-297.
8. [7] .Fritz Albregtsen 2008 Statistical Texture Measures Computed from Gray Level Co-occurrence Matrices (Image Processing Laboratory Department of Informatics University of Oslo). [8] Peter Howarth and Stefan Ruger 2004 Evaluation of Texture Features for Content-Based Image Retrieval (Department of Computing, Imperial College London South Kensington Campus London).
9. [9] Monika Sharma, R. B. Dubey and Sujata and S. K. Gupta 2012 Feature Extraction of Mammograms(International Journal of Advanced Computer Research Vol.2, No.3)
10. [10] Jaspinder Kaur, Nidhi Garg and Daljeet Kaur 2014 Segmentation and Feature Extraction of Lung Region for the Early Detection of Lung Tumor

CODE: Time Domain

```
## Raw Data Visualization
"""

import os
import pandas as pd
import numpy as np
from PIL import Image, UnidentifiedImageError # jpeg or jpg or png
import seaborn as sns                    #visualisation
import matplotlib.pyplot as plt

# Define paths
image_dir = '/content/drive/MyDrive/Colab Notebooks/FEDM/prepro'
save_csv_path = '/content/drive/MyDrive/Colab Notebooks/FEDM/raw_data.csv'

# Step 1: Traverse through directories and load image data
image_data_list = []
valid_image_extensions = ['.jpg', '.jpeg', '.png', '.bmp', '.tiff']  # Add valid image file extensions here

for class_folder in os.listdir(image_dir):
class_path = os.path.join(image_dir, class_folder)
if os.path.isdir(class_path):
for image_file in os.listdir(class_path):
image_path = os.path.join(class_path, image_file)
ext = os.path.splitext(image_file)[1].lower()  # Get the file extension and convert to lowercase

# Check if the file is a valid image format
if ext in valid_image_extensions:
try:
image = Image.open(image_path).convert('L')  # Convert image to grayscale
image_data = np.array(image).flatten()  # Flatten the image array
image_data_list.append([class_folder, image_file, image_data.tolist()])
except UnidentifiedImageError:
print(f"Cannot identify image file: {image_path}")
continue

# Step 2: Create a DataFrame
df = pd.DataFrame(image_data_list, columns=['class_folder', 'image_file', 'image_data'])

# Step 3: Save the DataFrame as CSV
df.to_csv(save_csv_path, index=False)
print(f"Data saved to {save_csv_path}")

# Step 4: Visualization
```

```
# --- 1. Visualizing the class distribution ---
plt.figure(figsize=(10, 6))
sns.countplot(data=df, x='class_folder')
plt.title('Distribution of Images by Class')
plt.xticks(rotation=45, ha='right')
plt.show()

# --- 2. Pixel Intensity Distribution for a Sample Image ---
# Apply only if the 'image_data' is not a numpy array
df['image_data'] = df['image_data'].apply(lambda x: np.array(x) if isinstance(x, list) else
    np.array(eval(x)))

# Select the first image's pixel data
sample_image_data = df['image_data'].iloc[0]

plt.figure(figsize=(8, 6))
plt.hist(sample_image_data, bins=50, color='blue', alpha=0.7)
plt.title('Pixel Intensity Distribution for a Sample Image')
plt.xlabel('Pixel Intensity')
plt.ylabel('Frequency')
plt.show()

# --- 3. Calculating and Visualizing the Correlation of Mean Pixel Intensity Between Classes ---
df['mean_pixel_intensity'] = df['image_data'].apply(np.mean)

plt.figure(figsize=(10, 8))
sns.heatmap(df.pivot_table(index='class_folder', values='mean_pixel_intensity', aggfunc='mean'),
    annot=True, cmap='coolwarm')
plt.title('Correlation of Mean Pixel Intensities Between Classes')
plt.show()

# --- 4. Pairplot of Image Statistics ---
df['std_pixel_intensity'] = df['image_data'].apply(np.std)
df['min_pixel_intensity'] = df['image_data'].apply(np.min)
df['max_pixel_intensity'] = df['image_data'].apply(np.max)

sns.pairplot(df[['mean_pixel_intensity', 'std_pixel_intensity', 'min_pixel_intensity',
    'max_pixel_intensity', 'class_folder']], hue='class_folder')
plt.show()

from google.colab import drive
drive.mount('/content/drive')

"""## Pre-processing"""
```

```
import os
import csv
from PIL import Image
from torchvision import transforms

# Paths to the root training directory and the output CSV file path
train_dir = r'/content/drive/MyDrive/Colab Notebooks/FEDM/Training Set'  # Directory containing the training images
output_csv = r'/content/drive/MyDrive/Colab Notebooks/FEDM/preprocessed_images.csv'  # Path where preprocessed image data will be saved as a CSV file

# Define image preprocessing transformations using torchvision.transforms
transform = transforms.Compose([
transforms.Resize((128, 128)),  # Resize all images to 128x128 pixels
transforms.ToTensor(),  # Convert images to PyTorch tensors (and scale pixel values to [0,1])
transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])  # Normalize image using ImageNet mean and std
])

# Function to preprocess images and save the preprocessed data into a CSV file
def preprocess_and_save_to_csv(train_dir, output_csv, transform):
# Open the CSV file in write mode
with open(output_csv, mode='w', newline='') as csv_file:
writer = csv.writer(csv_file)  # Create a CSV writer object
# Write the header of the CSV file (column names)
writer.writerow(['file_name', 'class_folder', 'image_data'])

# Iterate through each folder (representing a class) in the training directory
for class_folder in os.listdir(train_dir):
class_folder_path = os.path.join(train_dir, class_folder)  # Build the full path to the class folder

if os.path.isdir(class_folder_path):  # Ensure we are processing directories only (i.e., classes)
# Iterate through each file in the current class folder
for img_file in os.listdir(class_folder_path):
img_path = os.path.join(class_folder_path, img_file)  # Build the full path to the image file

# Process only image files (check the extension)
if img_file.lower().endswith(('.png', '.jpg', '.jpeg', '.bmp', '.gif')):
try:
# Open the image file using PIL
with Image.open(img_path) as img:
        # Apply the defined transformations (resize, tensor conversion, normalization)
        preprocessed_img = transform(img)
```

```
        # Flatten the preprocessed image tensor into a 1D list for saving in the CSV
        flattened_image = preprocessed_img.flatten().tolist()

        # Write the image's filename, class folder, and preprocessed image data to the CSV
        writer.writerow([img_file, class_folder, flattened_image])

        except Exception as e:
        # Print error if the image processing fails for any reason
        print(f"Error processing image {img_path}: {e}")
        else:
        # Print a message if the file is not an image (i.e., skipped non-image files)
        print(f"Skipped non-image file: {img_path}")

        # Run the function to preprocess the images and save the data to the CSV file
        preprocess_and_save_to_csv(train_dir, output_csv, transform)

        # Print a message when preprocessing is complete
        print(f"Preprocessing complete. Data saved in {output_csv}")

        """## Pre-Processed Data visualization"""

        # Import necessary libraries
        import pandas as pd         # For loading and handling CSV files
        import numpy as np          # For numerical operations
        import matplotlib.pyplot as plt  # For plotting
        import seaborn as sns       # For enhanced data visualization

        # Load the preprocessed images CSV file into a pandas DataFrame
        csv_path = r'/content/drive/MyDrive/Colab Notebooks/FEDM/preprocessed_images.csv'
        df = pd.read_csv(csv_path)

        # Display the first few rows of the DataFrame to check its structure
        print(df.head())

        # Print the information about the DataFrame (column names, data types, non-null counts)
        print(df.info())

        # --- 1. Visualizing the class distribution ---

        # Plot the distribution of images by class (the 'class_folder' column contains the classes)
        plt.figure(figsize=(10, 6))  # Set figure size
        sns.countplot(data=df, x='class_folder')  # Create a count plot for image classes
        plt.title('Distribution of Images by Class')  # Add a title to the plot
        plt.xticks(rotation=45, ha='right')  # Rotate x-axis labels for readability
        plt.show()  # Display the plot
```

```
136.        # --- 2. Pixel Intensity Distribution for a Sample Image ---

137.        # Convert the 'image_data' column from string to numpy arrays
138.        # 'eval' is used to convert string representations of lists into actual lists
139.        df['image_data'] = df['image_data'].apply(lambda x: np.array(eval(x)))

140.        # Select the first image's pixel data (assuming the first row for demonstration)
141.        sample_image_data = df['image_data'].iloc[0]

142.        # Plot the histogram of pixel intensity distribution for the selected image
143.        plt.figure(figsize=(8, 6))  # Set figure size
144.        plt.hist(sample_image_data, bins=50, color='blue', alpha=0.7)  # Plot the pixel intensity
145.        plt.title('Pixel Intensity Distribution for a Sample Image')  # Add plot title
146.        plt.xlabel('Pixel Intensity')  # Label the x-axis
147.        plt.ylabel('Frequency')  # Label the y-axis
148.        plt.show()  # Display the plot

149.        # --- 3. Calculating and Visualizing the Correlation of Mean Pixel Intensity Between Classes
            ---

150.        # Calculate the mean pixel intensity for each image and create a new column
            'mean_pixel_intensity'
151.        df['mean_pixel_intensity'] = df['image_data'].apply(np.mean)

152.        # Create a pivot table to get the average mean pixel intensity for each class
153.        # Then, plot a heatmap to visualize the correlation of pixel intensities between classes
154.        plt.figure(figsize=(10, 8))  # Set figure size
155.        sns.heatmap(df.pivot_table(index='class_folder', values='mean_pixel_intensity',
            aggfunc='mean'), annot=True, cmap='coolwarm')
156.        plt.title('Correlation of Mean Pixel Intensities Between Classes')  # Add a title to the plot
157.        plt.show()  # Display the heatmap

158.        # --- 4. Pairplot of Image Statistics ---

159.        # Calculate additional image statistics like standard deviation, minimum, and maximum
            pixel intensity
160.        df['std_pixel_intensity'] = df['image_data'].apply(np.std)  # Standard deviation
161.        df['min_pixel_intensity'] = df['image_data'].apply(np.min)  # Minimum pixel intensity
162.        df['max_pixel_intensity'] = df['image_data'].apply(np.max)  # Maximum pixel intensity

163.        # Use seaborn's pairplot to visualize the relationships between these statistics for each
            class
164.        sns.pairplot(df[['mean_pixel_intensity', 'std_pixel_intensity', 'min_pixel_intensity',
            'max_pixel_intensity', 'class_folder']], hue='class_folder')
```

```
plt.show()  # Show the pairplot

"""## Feature Extraction"""

import os  # For directory and file operations
import numpy as np  # For numerical computations
import pandas as pd  # For creating the CSV
from scipy.stats import skew, kurtosis, entropy  # For statistical features
from skimage import io  # For reading images
from skimage.util import view_as_windows  # For creating sliding windows

# Function to calculate pixel-wise features using a sliding window
def extract_pixel_features(image, window_size=3):
# Create sliding windows across the image
windows = view_as_windows(image, (window_size, window_size))

# Initialize a list to store feature maps for each window
features_list = []

# Iterate through each window and calculate features
for i in range(windows.shape[0]):
for j in range(windows.shape[1]):
window = windows[i, j].flatten()  # Flatten the window to calculate statistics

# Calculate features for the current window
features_map = {
'mean': np.mean(window),
'stddev': np.std(window),
'variance': np.var(window),
'skewness': skew(window),
'kurtosis': kurtosis(window),
'energy': np.sum(window ** 2),
'entropy': entropy(window),
'smoothness': 1 - (1 / (1 + np.var(window)))
}

# Append the window's features to the list
features_list.append(features_map)

# Return a list of dictionaries, each containing features for a window
return features_list

# Function to process all images and concatenate the features with class labels
def process_images(input_dir, output_dir, window_size=3):
data_list = []  # List to store data rows for CSV
```

```
# Traverse each class directory
for subdir, dirs, files in os.walk(input_dir):
class_name = os.path.basename(subdir)  # Get the class folder name (e.g., '1', '2')
if class_name.isdigit():  # Ensure that the folder name is numeric (class label)
class_label = int(class_name)  # Convert class folder name to integer

# Loop through each image file in the directory
for file in files:
if file.endswith(('.png', '.jpg', '.jpeg', '.tiff', '.bmp')):  # Process image files
image_path = os.path.join(subdir, file)  # Full image path

# Read the image as a grayscale image
image = io.imread(image_path, as_gray=True)

# Extract features from the image using the defined function
features_list = extract_pixel_features(image, window_size)

# Append each window's features with the class label
for features_map in features_list:
features_map['class_label'] = class_label
data_list.append(features_map)

# Convert the data list into a DataFrame (for CSV export)
df = pd.DataFrame(data_list)

# Save the DataFrame as a CSV file
output_csv = os.path.join(output_dir, 'image_features.csv')
df.to_csv(output_csv, index=False)
print(f"Features saved to {output_csv}")

# Define the input directory where class folders are stored
input_dir = r'/content/drive/MyDrive/Colab Notebooks/FEDM/prepro'

# Define the output directory where the CSV will be saved
output_dir = r'/content/drive/MyDrive/Colab Notebooks/FEDM/pixel'

# Process all images in the input directory and save features in the output directory
process_images(input_dir, output_dir, window_size=3)

"""## Concatenation"""

import pandas as pd

# Load the CSV files
```

```
file1 = '/content/drive/MyDrive/Colab Notebooks/FEDM/pixel/image_features.csv'
file2 = '/content/drive/MyDrive/Colab Notebooks/FEDM/pixel/training.csv'

df1 = pd.read_csv(file1)
df2 = pd.read_csv(file2)

# Concatenate the DataFrames (row-wise)
df_concat = pd.concat([df1, df2], axis=0)

# Save the concatenated DataFrame to the specified path
output_path = '/content/drive/MyDrive/Colab Notebooks/FEDM/pixel/concatenated_file.csv'
df_concat.to_csv(output_path, index=False)

print(f"Files concatenated and saved to {output_path} successfully!")

"""## Feature Extration Visualization"""

import seaborn as sns  # Seaborn for enhanced data visualization
import matplotlib.pyplot as plt  # Matplotlib for plotting
import pandas as pd  # Pandas for data manipulation

# Function to visualize features from the extracted CSV file
def visualize_features_from_csv(csv_file):
"""Function to visualize features using histograms with KDE."""

# Load the extracted features from the CSV file
features_df = pd.read_csv(csv_file)  # Read the CSV into a DataFrame

# Set the style for seaborn
sns.set(style="whitegrid")  # Set a style for the plots

# Loop through each feature (excluding filename and label)
for feature in features_df.columns:
if feature not in ['filename', 'label']:  # Exclude non-feature columns
plt.figure(figsize=(10, 6))  # Set figure size for each plot
sns.histplot(features_df[feature], kde=True, bins=30, color='blue', stat='density')  # Create histogram with KDE
plt.title(f'Distribution of {feature}')  # Title of the plot
plt.xlabel(feature)  # X-axis label
plt.ylabel('Density')  # Y-axis label
plt.axvline(features_df[feature].mean(), color='red', linestyle='--', label='Mean')  # Mean line
plt.axvline(features_df[feature].median(), color='green', linestyle='--', label='Median')  # Median line
plt.legend()  # Show legend
```

```
    plt.tight_layout()  # Adjust layout
    plt.show()  # Show the plot

    # Example usage of the function
    if __name__ == "__main__":
    # Specify the path to the CSV file containing the extracted features
    csv_file = os.path.join(output_dir, '/content/drive/MyDrive/Colab
Notebooks/FEDM/pixel/concatenated_file.csv')  # Update with your actual file name
    visualize_features_from_csv(csv_file)  # Call the visualization function

    """## Feature Selection using Wrapper Method"""

    import pandas as pd
    import numpy as np
    import matplotlib.pyplot as plt
    import seaborn as sns
    from sklearn.model_selection import train_test_split
    from sklearn.ensemble import RandomForestClassifier
    from sklearn.feature_selection import RFE, SequentialFeatureSelector
    from sklearn.metrics import accuracy_score

    # Load the CSV file
    file_path = '/content/drive/MyDrive/Colab Notebooks/FEDM/pixel/concatenated_file.csv'
    df = pd.read_csv(file_path)

    # Separate features (X) and the label (y)
    # Assuming 'label' is the target column
    X = df.drop(columns=['class_label'])  # Drop the label column from the features
    y = df['class_label']  # Store the label in y

    # Split the data into training and testing sets
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

    # Initialize a RandomForest model as the estimator for feature selection
    model = RandomForestClassifier(random_state=42)

    # Get the number of features in X_train
    n_features = X_train.shape[1]  # Number of features in the dataset

    # Ensure 'n_features_to_select' is less than the total number of features
    n_features_to_select = min(5, n_features)  # Select 5 features (or fewer)

    # ---------- Recursive Feature Elimination (RFE) ----------

    # Perform RFE to select the 5 best features
```

```
298.        rfe = RFE(model, n_features_to_select=n_features_to_select)
299.        rfe.fit(X_train, y_train)  # Fit RFE on the training data

300.        # Get the feature rankings from RFE
301.        ranking = rfe.ranking_
302.        features = X.columns  # Feature names

303.        # Plot the feature ranking
304.        plt.figure(figsize=(10, 6))
305.        sns.barplot(x=ranking, y=features)
306.        plt.title('Feature Rankings using RFE')
307.        plt.xlabel('Ranking')
308.        plt.ylabel('Features')
309.        plt.show()

310.        # ---------- Wrapper Forward Selection ----------

311.        # Perform Forward Selection to select the best features
312.        forward_selector = SequentialFeatureSelector(model,
    n_features_to_select=n_features_to_select, direction='forward')
313.        forward_selector.fit(X_train, y_train)  # Fit forward selector on the training data

314.        # Extract the selected features from the forward selection
315.        selected_features_fwd = np.array(features)[forward_selector.get_support()]

316.        # Plot the selected features
317.        plt.figure(figsize=(10, 6))
318.        sns.barplot(x=np.ones(len(selected_features_fwd)), y=selected_features_fwd)
319.        plt.title('Selected Features using Forward Selection')
320.        plt.xlabel('Importance')
321.        plt.ylabel('Features')
322.        plt.show()

323.        # ---------- Wrapper Backward Selection ----------

324.        # Perform Backward Selection to select the best features
325.        backward_selector = SequentialFeatureSelector(model,
    n_features_to_select=n_features_to_select, direction='backward')
326.        backward_selector.fit(X_train, y_train)  # Fit backward selector on the training data

327.        # Extract the selected features from the backward selection
328.        selected_features_bwd = np.array(features)[backward_selector.get_support()]

329.        # Plot the selected features
330.        plt.figure(figsize=(10, 6))
```

```
331.        sns.barplot(x=np.ones(len(selected_features_bwd)), y=selected_features_bwd)
332.        plt.title('Selected Features using Backward Selection')
333.        plt.xlabel('Importance')
334.        plt.ylabel('Features')
335.        plt.show()

336.        # ---------- Model Evaluation ----------

337.        # Use the features selected by RFE for training the model
338.        X_train_rfe = rfe.transform(X_train)
339.        X_test_rfe = rfe.transform(X_test)

340.        # Train the model on the selected features
341.        model.fit(X_train_rfe, y_train)

342.        # Predict using the model trained on RFE-selected features
343.        y_pred = model.predict(X_test_rfe)

344.        # Print the accuracy score to evaluate the performance
345.        print("Accuracy using RFE selected features: ", accuracy_score(y_test, y_pred))

346.        """## Feature extration from feature selection: Selected Feature from Forward Selected
    Features and backward feature selection"""

347.        import pandas as pd  # Import pandas for data manipulation
348.        import numpy as np  # Import NumPy for numerical operations
349.        import os  # Import os for directory operations
350.        import re  # Import re for regular expressions

351.        # Define the path to the concatenated feature CSV file
352.        input_directory = '/content/drive/MyDrive/Colab
    Notebooks/FEDM/pixel/concatenated_file.csv'  # Path to the CSV file

353.        # List of features to extract (excluding the last column for now)
354.        features_to_extract = ['smoothness', 'entropy', 'kurtosis', 'skewness', 'variance', 'stddev',
    'mean']  # Specify the features of interest

355.        # Initialize an empty list to hold DataFrames
356.        all_dataframes = []  # List to store individual DataFrames

357.        # Read the concatenated CSV file into a DataFrame
358.        df = pd.read_csv(input_directory)  # Read the CSV file into a DataFrame

359.        # Extract only the specified features and ignore non-float columns
360.        extracted_features = df[features_to_extract]  # Select the specified features
```

```
    extracted_features = extracted_features.select_dtypes(include=[np.float64])  # Keep only float columns

    # Retain the last column from the original DataFrame
    last_column_name = df.columns[-1]  # Get the name of the last column
    extracted_features[last_column_name] = df[last_column_name]  # Add the last column to the extracted features

    # Rename the DataFrame's index to the filename (if applicable) and keep only the numerical part
    numerical_filename = re.search(r'\d+', os.path.basename(input_directory))  # Extract numerical part from the filename
    if numerical_filename:  # Check if a number was found
    extracted_features['filename'] = numerical_filename.group()  # Add the numerical part as the filename

    all_dataframes.append(extracted_features)  # Append the extracted features DataFrame to the list

    # Concatenate all DataFrames into a single DataFrame (though only one DataFrame is read in this case)
    final_dataframe = pd.concat(all_dataframes, ignore_index=True)  # Combine all extracted features DataFrames

    # Save the final DataFrame to a new CSV file (specifying the filename)
    output_file_path = '/content/drive/MyDrive/Colab Notebooks/FEDM/pixel/extracted_features_final.csv'  # Define the output file path including filename
    final_dataframe.to_csv(output_file_path, index=False)  # Save the combined DataFrame to a CSV file without the index

    print("Extraction completed and saved to:", output_file_path)  # Print completion message

    """## Classifier-DT, KNN (K2-K10), Random Forest, SVM, NN-MPTL

    ## Step 1: Import Libraries and Load Dataset
    """

    # Import libraries for data manipulation, model building, and evaluation
    import os  # To handle file paths
    import pickle  # For saving and loading model objects
    import numpy as np  # For numerical operations
    import pandas as pd  # For data manipulation and loading CSV files
    import seaborn as sns  # For visualizing data and plotting confusion matrices
    from sklearn.tree import DecisionTreeClassifier  # Decision Tree classifier
```

```
386.  from sklearn.ensemble import RandomForestClassifier  # Random Forest classifier
387.  from sklearn.svm import SVC  # Support Vector Classifier (SVM)
388.  from sklearn.neighbors import KNeighborsClassifier  # K-Nearest Neighbors classifier
      (KNN)
389.  from sklearn.neural_network import MLPClassifier  # Multilayer Perceptron classifier
      (Neural Network)
390.  from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score,
      confusion_matrix, roc_auc_score, roc_curve, auc  # Metrics for evaluation
391.  from sklearn.preprocessing import StandardScaler, label_binarize  # For feature scaling
      and binarizing labels
392.  from sklearn.model_selection import train_test_split  # To split data into train/test sets
393.  import matplotlib.pyplot as plt  # For plotting graphs
394.  from tqdm import tqdm  # Progress bar for iterations

395.  # Path to the dataset CSV file
396.  csv_file = '/content/drive/MyDrive/Colab Notebooks/FEDM/pixel/concatenated_file.csv'

397.  # Load dataset as a DataFrame
398.  data = pd.read_csv(csv_file)

399.  # Separate features (X) from the target variable (y)
400.  X = data.drop('class_label', axis=1)  # Drop 'class_label' column to get feature set
401.  y = data['class_label']  # Define target variable as 'class_label'

402.  # Display initial rows of the dataset to verify loading
403.  data.head()

404.  """## Step 2: Data Scaling and Train-Test Split"""

405.  # Initialize a StandardScaler to standardize feature values
406.  scaler = StandardScaler()

407.  # Fit the scaler to the features and transform them
408.  X_scaled = scaler.fit_transform(X)

409.  # Split data into training (80%) and testing (20%) sets
410.  X_train, X_test, y_train, y_test = train_test_split(X_scaled, y, test_size=0.2,
      random_state=42)

411.  # Output the number of samples in the training and testing sets
412.  print(f'Training set size: {X_train.shape[0]}')
413.  print(f'Testing set size: {X_test.shape[0]}')

414.  """## Step 3: Initialize Classifiers"""
```

```
415.        # Create a dictionary with different classifiers for easy iteration and access
416.        classifiers = {
417.        "DecisionTree": DecisionTreeClassifier(random_state=0),  # Decision Tree with fixed
    random seed
418.        "RandomForest": RandomForestClassifier(n_estimators=50, max_depth=10, n_jobs=-1,
    random_state=0),  # Random Forest with 50 trees, depth of 10, and parallel processing
419.        "SVM": SVC(kernel='rbf', C=1.0, gamma='scale', probability=True, random_state=0),  #
    Support Vector Machine with RBF kernel
420.        "NeuralNetwork": MLPClassifier(hidden_layer_sizes=(100, 50), max_iter=1000,
    random_state=0, learning_rate_init=0.0005, alpha=0.001)  # MLP neural network with two layers
    and specific hyperparameters
421.        }

422.        # Add K-Nearest Neighbors classifiers from K=2 to K=10 to the dictionary
423.        for k in range(2, 11):
424.        classifiers[f"KNN_{k}"] = KNeighborsClassifier(n_neighbors=k)  # Add KNN with k neighbors

425.        """Step 4: Define Function for Metrics and Confusion Matrix Plotting"""

426.        # Function to calculate and display classification metrics and plot confusion matrix
427.        def print_metrics(y_test, y_pred, clf_name):
428.        # Generate confusion matrix for true and predicted labels
429.        cm = confusion_matrix(y_test, y_pred)

430.        # Calculate metrics: Accuracy, Precision, Sensitivity (Recall), Specificity, F1 Score
431.        accuracy = accuracy_score(y_test, y_pred)
432.        precision = precision_score(y_test, y_pred, average="weighted")
433.        sensitivity = recall_score(y_test, y_pred, average="weighted")  # Recall (Sensitivity)
434.        specificity = np.sum(cm.diagonal()) / np.sum(cm)  # Specificity calculation
435.        f1 = f1_score(y_test, y_pred, average="weighted")

436.        # Display calculated metrics for the current classifier
437.        print(f"\nMetrics for {clf_name}:")
438.        print(f"Accuracy: {accuracy:.4f}")
439.        print(f"Precision: {precision:.4f}")
440.        print(f"Sensitivity (Recall): {sensitivity:.4f}")
441.        print(f"Specificity: {specificity:.4f}")
442.        print(f"F1-Score: {f1:.4f}")

443.        # Plot confusion matrix as a heatmap for visual inspection
444.        plt.figure(figsize=(6, 4))
445.        sns.heatmap(cm, annot=True, fmt="d", cmap='Blues', cbar=False)
446.        plt.title(f'Confusion Matrix - {clf_name}')
447.        plt.xlabel('Predicted')
448.        plt.ylabel('True')
```

```
449.        plt.show()

450.        # Return metrics for future use
451.        return accuracy, precision, sensitivity, specificity, f1

452.        """Step 5: Train Classifiers and Evaluate Performance"""

453.        # Track the classifier with the highest accuracy for final output
454.        best_classifier = {"name": None, "accuracy": 0}

455.        # Iterate over classifiers to train and evaluate each model
456.        for clf_name, clf in classifiers.items():
457.        print(f"Training {clf_name}...")  # Notify training start for each classifier

458.        # Train the classifier on the training data
459.        clf.fit(X_train, y_train)

460.        # Predict on the test set
461.        y_pred = clf.predict(X_test)

462.        # Calculate metrics and plot confusion matrix for the current classifier
463.        accuracy, precision, sensitivity, specificity, f1 = print_metrics(y_test, y_pred, clf_name)

464.        # Update the best classifier if the current one has higher accuracy
465.        if accuracy > best_classifier["accuracy"]:
466.        best_classifier = {"name": clf_name, "accuracy": accuracy}

467.        # Save the trained model to a file using pickle
468.        with open(f'./{clf_name}_model.p', 'wb') as f:
469.        pickle.dump(clf, f)

470.        """Step 6: ROC and AUC Curves"""

471.        # Loop through each classifier to plot the ROC curve
472.        for clf_name, clf in classifiers.items():
473.        plt.figure(figsize=(10, 8))  # Create a new figure for each classifier
474.        plt.plot([0, 1], [0, 1], 'k--')  # Add a dashed line for random guessing (baseline)
475.        plt.title(f'ROC Curve for {clf_name}')  # Title for the ROC plot
476.        plt.xlabel('False Positive Rate')  # Label for the x-axis
477.        plt.ylabel('True Positive Rate')  # Label for the y-axis

478.        # Check if classifier has probability prediction capability
479.        if hasattr(clf, "predict_proba"):
480.        y_proba = clf.predict_proba(X_test)  # Get probability estimates for the test set
```

```
        # Handle binary and multi-class cases differently
        if len(np.unique(y)) == 2:
        # Binary classification
        roc_auc = roc_auc_score(y_test, y_proba[:, 1])  # AUC score for binary classification
        fpr, tpr, _ = roc_curve(y_test, y_proba[:, 1])  # Calculate FPR and TPR for ROC curve
        plt.plot(fpr, tpr, label=f'AUC = {roc_auc:.4f}')  # Plot ROC curve with AUC label
        else:
        # Multi-class classification
        y_test_binarized = label_binarize(y_test, classes=np.unique(y))  # Binarize test labels
        roc_auc = roc_auc_score(y_test_binarized, y_proba, average="weighted",
    multi_class="ovr")  # Weighted AUC
        fpr, tpr, _ = roc_curve(y_test_binarized.ravel(), y_proba.ravel())  # Calculate ROC curve
        plt.plot(fpr, tpr, label=f'Weighted AUC = {roc_auc:.4f}')  # Plot multi-class ROC curve

        # Handle classifiers that do not support probability estimates
        elif hasattr(clf, "decision_function"):
        # For models with decision_function (like SVM)
        y_score = clf.decision_function(X_test)  # Get decision scores
        if len(np.unique(y)) == 2:
        roc_auc = roc_auc_score(y_test, y_score)  # Binary AUC score
        fpr, tpr, _ = roc_curve(y_test, y_score)  # Calculate FPR and TPR for ROC
        plt.plot(fpr, tpr, label=f'AUC = {roc_auc:.4f}')  # Plot binary ROC curve
        else:
        # Multi-class AUC for decision_function classifiers
        y_test_binarized = label_binarize(y_test, classes=np.unique(y))  # Binarize labels for multi-
    class
        roc_auc = roc_auc_score(y_test_binarized, y_score, average="weighted",
    multi_class="ovr")  # Weighted AUC
        fpr, tpr, _ = roc_curve(y_test_binarized.ravel(), y_score.ravel())  # Calculate ROC curve
        plt.plot(fpr, tpr, label=f'Weighted AUC = {roc_auc:.4f}')  # Plot ROC curve for multi-class

        else:
        print(f"{clf_name} does not support ROC AUC plotting.")
        continue

        # Finalize the ROC plot with a legend and display it
        plt.legend(loc="lower right")  # Position legend in the lower-right corner
        plt.xlim([0.0, 1.0])  # Set x-axis limits for ROC plot
        plt.ylim([0.0, 1.05])  # Set y-axis limits for ROC plot
        plt.show()  # Display the completed ROC plot for the current classifier

        """Step 7: Display the Best Classifier"""

        # Output the name and accuracy of the best-performing classifier
```

```
517.        print(f"Best Classifier: {best_classifier['name']} with Accuracy:
    {best_classifier['accuracy']:.4f}")

518.        """Step 8: Plotting Accuracy Bar Plot for Each Classifier"""

519.        # Step 8: Plotting Accuracy Bar Plot for Each Classifier

520.        # Create a list to store classifier names and their corresponding accuracies
521.        classifier_names = []
522.        classifier_accuracies = []

523.        # Populate the lists with classifier names and their accuracies
524.        for clf_name in classifiers.keys():
525.        # Retrieve the model file for each classifier
526.        with open(f'./{clf_name}_model.p', 'rb') as f:
527.        clf = pickle.load(f)  # Load the model
528.        y_pred = clf.predict(X_test)  # Predict on the test set
529.        accuracy = accuracy_score(y_test, y_pred)  # Calculate accuracy
530.        classifier_names.append(clf_name)  # Add classifier name to the list
531.        classifier_accuracies.append(accuracy)  # Add accuracy to the list

532.        # Create a bar plot to visualize the accuracies of each classifier
533.        plt.figure(figsize=(12, 6))
534.        sns.barplot(x=classifier_names, y=classifier_accuracies, palette='viridis')
535.        plt.title('Classifier Accuracies')
536.        plt.xlabel('Classifiers')
537.        plt.ylabel('Accuracy')
538.        plt.xticks(rotation=45)  # Rotate x-axis labels for better readability
539.        plt.ylim(0, 1)  # Set y-axis limits from 0 to 1
540.        plt.grid(axis='y')  # Add horizontal grid lines for better readability
541.        plt.show()

542.        """Step 09: Plot Loss Curve for MLPClassifier"""

543.        # Step 09: Plot Loss Curve for MLPClassifier

544.        # Initialize MLPClassifier with parameters for training
545.        mlp_clf = MLPClassifier(
546.        hidden_layer_sizes=(100, 50),
547.        max_iter=1000,
548.        random_state=0,
549.        learning_rate_init=0.0005,
550.        alpha=0.001
551.        )
```

```
552.        # Fit the model and store the loss values for each iteration
553.        mlp_clf.fit(X_train, y_train)

554.        # Get the loss values for plotting
555.        loss_values = mlp_clf.loss_curve_

556.        # Plot the loss curve
557.        plt.figure(figsize=(10, 6))
558.        plt.plot(loss_values, label='Training Loss', color='blue')
559.        plt.title('Loss Curve for MLPClassifier')
560.        plt.xlabel('Iterations')
561.        plt.ylabel('Loss')
562.        plt.legend()
563.        plt.grid()
564.        plt.show()

565.        """## 10 flod Cross Validation"""

566.        # Import necessary libraries
567.        import pandas as pd  # For data manipulation and analysis
568.        from sklearn.model_selection import KFold  # For performing k-fold cross-validation
569.        from sklearn.ensemble import RandomForestClassifier  # Random Forest model for
    classification
570.        from sklearn.metrics import accuracy_score  # For calculating accuracy of the model
571.        import matplotlib.pyplot as plt  # For plotting

572.        # Load the dataset from the specified file path
573.        file_path = '/content/drive/MyDrive/Colab Notebooks/FEDM/pixel/concatenated_file.csv'
574.        data = pd.read_csv(file_path)  # Read the CSV file into a DataFrame

575.        # Display the first few rows of the dataframe to understand its structure
576.        print(data.head())

577.        # Prepare features (X) and labels (y)
578.        X = data.drop(columns=['class_label'])  # Drop the 'label' column to create the feature set
579.        y = data['class_label']  # Select the 'label' column as the target variable

580.        # Set up 10-fold cross-validation
581.        k = 10  # Number of folds for cross-validation
582.        kf = KFold(n_splits=k, shuffle=True, random_state=42)  # Create KFold object with specified
    parameters

583.        # Initialize the model to be used for training
584.        model = RandomForestClassifier()  # Choose Random Forest Classifier (you can select any
    other model)
```

```
    # Store accuracy scores for each fold
    accuracy_scores = []  # Initialize an empty list to hold the accuracy for each fold

    # Perform k-fold cross-validation
    for train_index, test_index in kf.split(X):  # Iterate over each fold
    X_train, X_test = X.iloc[train_index], X.iloc[test_index]  # Split the data into training and testing sets
    y_train, y_test = y.iloc[train_index], y.iloc[test_index]  # Split the labels into training and testing sets

    # Train the model on the training data
    model.fit(X_train, y_train)  # Fit the model to the training data

    # Make predictions on the test data
    y_pred = model.predict(X_test)  # Predict the labels for the test set

    # Calculate the accuracy of the model's predictions
    accuracy = accuracy_score(y_test, y_pred)  # Compare predicted labels with true labels
    accuracy_scores.append(accuracy)  # Append the accuracy to the list

    # Calculate mean and best accuracy
    mean_accuracy = sum(accuracy_scores) / k  # Mean accuracy across all folds
    best_accuracy = max(accuracy_scores)  # Best accuracy from all folds
    best_fold = accuracy_scores.index(best_accuracy) + 1  # Identify the fold with the best accuracy

    # Display the results of the cross-validation
    print(f'Accuracy scores for each fold: {accuracy_scores}')  # Print the accuracy for each fold
    print(f'Mean accuracy: {mean_accuracy}')  # Print the mean accuracy
    print(f'Best accuracy: {best_accuracy} (Fold {best_fold})')  # Print the best accuracy and its fold number

    # Plotting the accuracy scores for each fold
    plt.figure(figsize=(10, 6))  # Set the figure size
    plt.plot(range(1, k + 1), accuracy_scores, marker='o', linestyle='-', color='b')  # Plot accuracy scores
    plt.title('K-Fold Cross-Validation Accuracy Scores')  # Title of the plot
    plt.xlabel('Fold Number')  # X-axis label
    plt.ylabel('Accuracy')  # Y-axis label
    plt.xticks(range(1, k + 1))  # Set x-ticks to match the fold numbers
    plt.axhline(y=mean_accuracy, color='r', linestyle='--', label=f'Mean Accuracy: {mean_accuracy:.2f}')  # Mean accuracy line
    plt.axhline(y=best_accuracy, color='g', linestyle='--', label=f'Best Accuracy: {best_accuracy:.2f} (Fold {best_fold})')  # Best accuracy line
```

```
615.        plt.legend()  # Show legend
616.        plt.grid()  # Show grid
617.        plt.show()  # Display the plot
```