****

Faculty of Technology and Engineering

Computer Engineering

**Machine Learning Project**

**Hedieh Moshtaghi Mohamadzade**

**40214163119**

**Dr. Fatemeh Bagheri**

Fall of 2024

Table of Contents

[1 Dataset 4](#_Toc182743130)

[1.1 Specifications of the Data Set 4](#_Toc182743131)

[1.2 Examples of Data Sets 4](#_Toc182743132)

[1.3 The Basic Characteristics of The Data Set 5](#_Toc182743133)

[1.4 Data Description Charts 5](#_Toc182743134)

[1.4.1 Nominal Feature Distribution Chart 5](#_Toc182743135)

[1.4.2 Heatmap of Feature Correlation 6](#_Toc182743136)

[2 Preprocessing Code 7](#_Toc182743137)

[2.1 Data Loading 7](#_Toc182743138)

[2.2 Handling Missing Values 7](#_Toc182743139)

[2.3 Feature Normalization 8](#_Toc182743140)

[2.4 Target Variable Encoding 9](#_Toc182743141)

[2.5 Handling Class Imbalance 9](#_Toc182743142)

[2.6 Dataset Splitting 10](#_Toc182743143)

[2.7 The Reason for These Steps 10](#_Toc182743144)

[3 Implementation of Decision Tree Algorithms on Data 10](#_Toc182743145)

[3.1 Evaluation of Machine Learning Models on Imbalanced Data with Different Imputation Techniques 10](#_Toc182743146)

[3.2 CART (Classification and Regression Tree) 11](#_Toc182743147)

[3.3 C4.5 15](#_Toc182743148)

[3.4 AdaBoost 19](#_Toc182743149)

[3.5 XGBoost 21](#_Toc182743150)

[3.6 Random Forest 24](#_Toc182743151)

[3.7 LightGBM 28](#_Toc182743152)

[3.8 ExtraTrees 31](#_Toc182743153)

[3.9 Gradient Boosting 36](#_Toc182743154)

[Figure 1 - Distribution of Class Feature Data 5](#_Toc182745114)

[Figure 2 - Heatmap of Feature Correlations 6](#_Toc182745115)

[Figure 3 - Code for Data Loading 7](#_Toc182745116)

[Figure 4 - Code for Replacing '?' with NaN 7](#_Toc182745117)

[Figure 5 - Code for Using the Overall Mean 7](#_Toc182745118)

[Figure 6 - Code for Using Class-Wise Mean 7](#_Toc182745119)

[Figure 7 - Code for Using K-Nearest Neighbors (KNN) 8](#_Toc182745120)

[Figure 8 - Code for Feature Normalization 9](#_Toc182745121)

[Figure 9 - Code for Target Variable Encoding 9](#_Toc182745122)

[Figure 10 - Code for Checking Class Distribution 9](#_Toc182745123)

[Figure 11 - Code for Applying SMOTE 9](#_Toc182745124)

[Figure 12 - Code for Verifying Balanced Distribution 10](#_Toc182745125)

[Figure 13 - Code for Dataset Splitting 10](#_Toc182745126)

[Figure 14 - The Best Decision Tree Generated By The CART Algorithm 13](#_Toc182745127)

[Figure 15 - Rules of The Best Decision Tree Generated By The CART Algorithm 14](#_Toc182745128)

[Figure 16 - The Confusion Matrix of Cart Algorithm 15](#_Toc182745129)

[Figure 17 - The Best Decision Tree Generated By The C4.5 Algorithm 17](#_Toc182745130)

[Figure 18 - Rules of The Best Decision Tree Generated By The C4.5 Algorithm 18](#_Toc182745131)

[Figure 19 - The Confusion Matrix of C4.5 Algorithm 19](#_Toc182745132)

[Figure 20 - The Best Decision Tree Generated By The AdaBoost Algorithm 21](#_Toc182745133)

[Figure 21 - Rules of The Best Decision Tree Generated By The AdaBoost Algorithm 21](#_Toc182745134)

[Figure 22 - The Confusion Matrix of AdaBoost Algorithm 21](#_Toc182745135)

[Figure 23 - One of The Best Decision Trees Generated By The XGBoost Algorithm 23](#_Toc182745136)

[Figure 24 - One of The Best Decision Trees Generated By The XGBoost Algorithm 23](#_Toc182745137)

[Figure 25 - The First Confusion Matrix of XGBoost Algorithm 24](#_Toc182745138)

[Figure 26 - The Second Confusion Matrix of XGBoost Algorithm 24](#_Toc182745139)

[Figure 27 - The Best Decision Tree Generated By The Random Forest Algorithm 26](#_Toc182745140)

[Figure 28 - Rules of The Best Decision Tree Generated By The Random Forest Algorithm 27](#_Toc182745141)

[Figure 29 - The Confusion Matrix of Random Forest Algorithm 28](#_Toc182745142)

[Figure 30 - One of The Best Decision Tree Generated By The LightGBM Algorithm 30](#_Toc182745143)

[Figure 31 - One of The Best Decision Tree Generated By The LightGBM Algorithm 30](#_Toc182745144)

[Figure 32 - The First Confusion Matrix of LightGBM Algorithm 30](#_Toc182745145)

[Figure 33 - The Second Confusion Matrix of LightGBM Algorithm 31](#_Toc182745146)

[Figure 34 - The Best Decision Tree Generated By The ExtraTrees Algorithm 33](#_Toc182745147)

[Figure 35 - Rules of The Best Decision Tree Generated By The ExtraTrees Algorithm 35](#_Toc182745148)

[Figure 36 - The Confusion Matrix of ExtraTrees Algorithm 36](#_Toc182745149)

[Figure 37 - The Best Decision Tree Generated By The GradientBoosting Algorithm 38](#_Toc182745150)

[Figure 38 - The Confusion Matrix of GradientBoosting Algorithm 38](#_Toc182745151)

# Dataset

The Wisconsin Breast Cancer Dataset includes data about breast tumor samples created for the purpose of distinguishing between benign and malignant tumor types. This dataset includes 699 samples taken from breast tumor patients, each sample containing different recorded features. This information is utilized to establish if a tumor is non-cancerous or cancerous and consists of 11 unique characteristics.

The characteristics consist of clump thickness, uniformity of cell size, uniformity of cell shape, adhesion of cells to surrounding tissue, nucleus size, and number of mitotic figures. There are 16 missing values in the dataset for the "Bare Nuclei" attribute, which could be filled in using the mean or other imputation techniques.

Data was analyzed and described using Python libraries like Pandas, NumPy, and Scikit-learn. These tools were used to describe statistical features, identify outliers, and create different charts. You can find the data analysis and description code in the GitHub repository provided below:

لینک گیت‌هاب کدها

## Specifications of the Data Set

Dataset Description Table

|  |  |  |
| --- | --- | --- |
| Description | Attribute | Row |
| Unique ID For Each Tissue Sample | Sample Code Number | 1 |
| Clump Thickness Of Tumor Cells | Clump Thickness | 2 |
| Uniformity In The Size Of Tumor Cells | Uniformity Of Cell Size | 3 |
| Uniformity In The Shape Of Tumor Cells | Uniformity Of Cell Shape | 4 |
| Adhesion Level Of Tumor Cells To Surrounding Tissue | Marginal Adhesion | 5 |
| Size Of Individual Tumor Cells | Single Epithelial Cell Size | 6 |
| Presence Of Nuclei Without Surrounding Cytoplasm | Bare Nuclei | 7 |
| Evaluation Of Chromatin Structure In Tumor Cells | Bland Chromatin | 8 |
| Presence Of Normal Nuclei In Tumor Cells | Normal Nucleoli | 9 |
| Number Of Mitotic Divisions In Cells | Mitoses | 10 |
| Tumor Type (2 = Benign, 4 = Malignant) | Class | 11 |

## Examples of Data Sets

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Sample code number | Clump Thickness | Uniformity of Cell Size | Uniformity of Cell Shape | Marginal Adhesion | Single Epithelial Cell Size | Bare Nuclei | Bland Chromatin | Normal Nucleoli | Mitoses | Class |
| 1000025 | 5 | 1 | 1 | 1 | 2 | 1 | 3 | 1 | 1 | 2 |
| 1002945 | 5 | 4 | 4 | 5 | 7 | 10 | 3 | 2 | 1 | 2 |
| 1015425 | 3 | 1 | 1 | 1 | 2 | 2 | 3 | 1 | 1 | 2 |
| 1016277 | 6 | 8 | 8 | 1 | 3 | 4 | 3 | 7 | 1 | 2 |
| 1017023 | 4 | 1 | 1 | 3 | 2 | 1 | 3 | 1 | 1 | 2 |
| 1017122 | 8 | 10 | 10 | 8 | 7 | 10 | 9 | 7 | 1 | 4 |

## The Basic Characteristics of The Data Set

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Number Of Missing Data | The Median Value | Minimum Amount | Maximum Amount | Number | Feature |
| 0 | 4 | 1 | 10 | 699 | Clump Thickness |
| 0 | 1 | 1 | 10 | 699 | Uniformity of Cell Size |
| 0 | 1 | 1 | 10 | 699 | Uniformity of Cell Shape |
| 0 | 1 | 1 | 10 | 699 | Marginal Adhesion |
| 0 | 2 | 1 | 10 | 699 | Single Epithelial Cell Size |
| 16 | 1 | 1 | 10 | 683 | Bare Nuclei |
| 0 | 3 | 1 | 10 | 699 | Bland Chromatin |
| 0 | 1 | 1 | 10 | 699 | Normal Nucleoli |
| 0 | 1 | 1 | 10 | 699 | Mitoses |
| 0 | 2 | 2 | 4 | 699 | Class |

## Data Description Charts

### Nominal Feature Distribution Chart

Figure 1 illustrates a bar graph displaying the distribution of nominal values within the Class feature. This visual representation illustrates the distribution of samples for each category, displaying the quantities of both benign and malignant tumors in the dataset. Every bar shown on the graph corresponds to a specific type of tumor, with the bar's height revealing the number of samples associated with that particular type.

Bar 2.0 refers to benign growths, including around 450 samples of these in the dataset.

Bar 4.0 illustrates malignant tumors, containing about 250 samples of malignant tumors in the dataset.

According to this chart, it is evident that there is a higher number of benign tumors in the dataset compared to malignant ones.

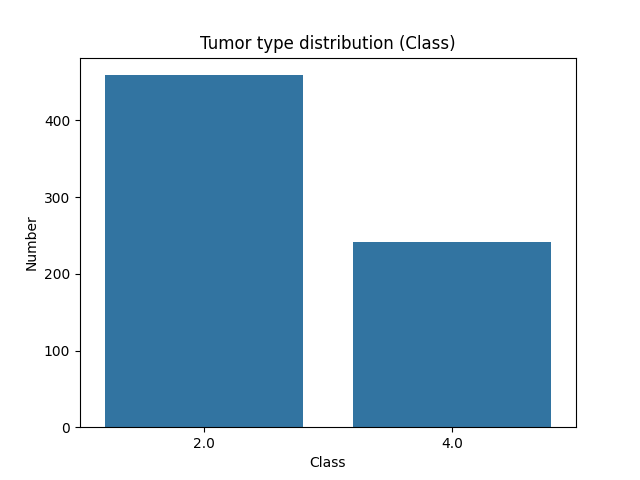


Figure 1 - Distribution of Class Feature Data

### Heatmap of Feature Correlation

Figure 2 shows the heatmap of Pearson correlation coefficients for every feature in the dataset. This heatmap shows the correlations between features that could help predict tumor types.

Red colors show significant positive correlations between two characteristics, suggesting that when the value of one characteristic rises, the value of the other characteristic also tends to increase.

Blue hues represent significant negative connections, where an increase in one feature leads to a decrease in another feature.

Shades close to white: Represent a lack of strong correlation between two characteristics.

Main Points:

Numerous features exhibit strong positive correlations, indicating they are interconnected and may aid in tumor classification.

Key aspects: Characteristics such as Consistency in Cell Size, Consistency in Cell Shape, Marginal Adhesion, and Bare Nuclei are highly associated with the Class characteristic. This shows their important function in deciding the type of tumor (benign or malignant).

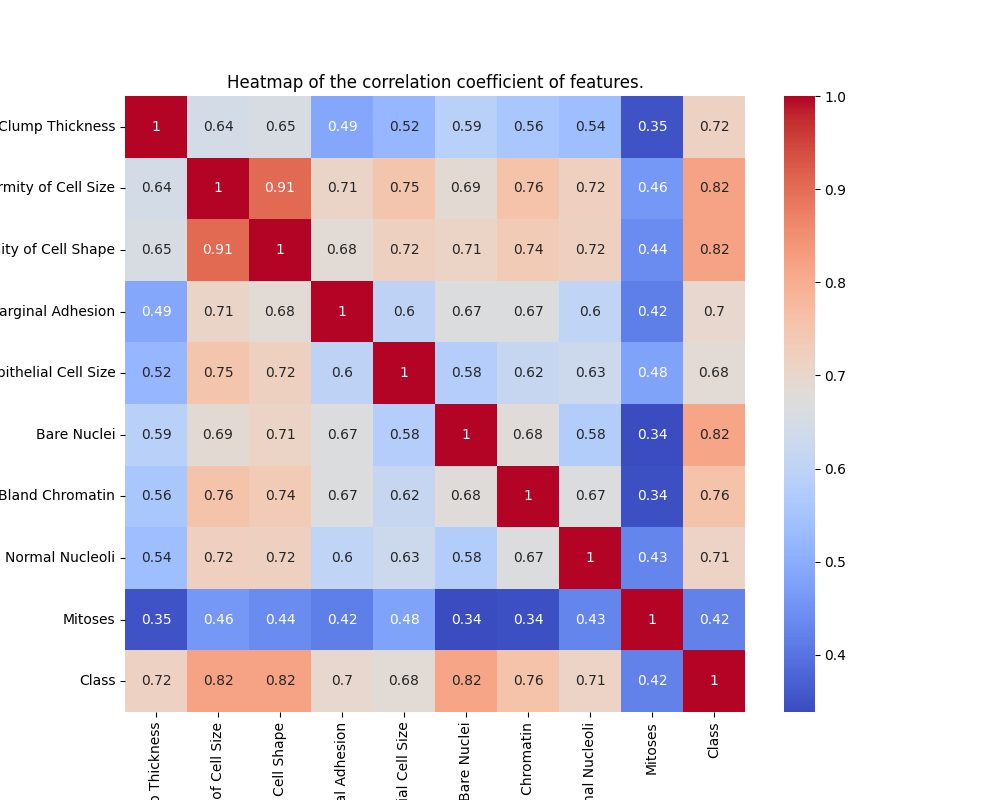


Figure 2 - Heatmap of Feature Correlations

# 

# Preprocessing Code

The provided code preprocesses a breast cancer dataset (`data.csv`) and performs multiple methods to handle missing data, normalize features, and address potential class imbalances. Each step is crafted to prepare the data for machine learning models while ensuring the integrity of the dataset. Below is a step-by-step breakdown of the code:

## Data Loading

In Figure 3, the dataset is loaded into a pandas DataFrame named `data` using `pd.read\_csv`. This provides a tabular structure for easier manipulation and analysis.

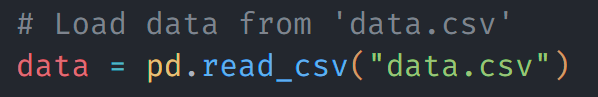


Figure 3 - Code for Data Loading

## Handling Missing Values

The column `Bare Nuclei` contains missing values denoted by the placeholder `?`. These missing values need to be handled to ensure the dataset is complete and usable for machine learning.

**-Step A: Replacing '?' with NaN**

In Figure 4, the `replace` method replaces all instances of `?` with `NaN`. The column is then converted to a numerical data type (`float`) to allow mathematical operations like mean computation.

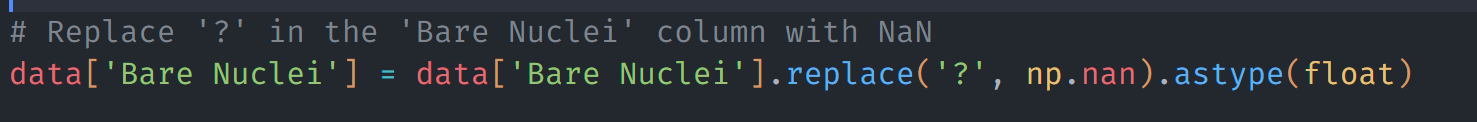


Figure 4 - Code for Replacing '?' with NaN

**-Step B: Filling Missing Values**

The missing values in the `Bare Nuclei` column are imputed in three ways:

1. Using the Overall Mean:

Here in Figure 5, all missing values are replaced with the mean of the `Bare Nuclei` column. This is a simple but effective strategy when the data distribution is balanced.

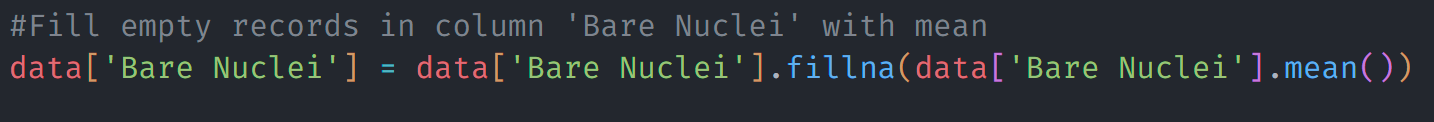


Figure 5 - Code for Using the Overall Mean

2. Using Class-Wise Mean:

This method replaces missing values with the mean of the respective class (`Class` column) in Figure 6. It provides more context-aware imputation by leveraging the relationship between the target variable and the missing values.

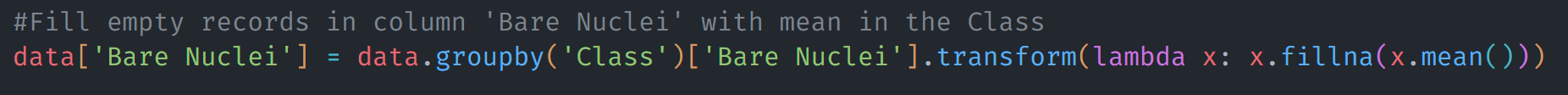


Figure 6 - Code for Using Class-Wise Mean

3. Using K-Nearest Neighbors (KNN):

**-Step 1: Initialize the Imputer**

In Figure 7, the KNNImputer from sklearn is instantiated with n\_neighbors=5. This sets the number of neighbors (k) to consider for estimating missing values.

The choice of k=5 is a balance between capturing sufficient local patterns and avoiding over-smoothing the data.

**-Step 2: Apply Imputation**

The fit\_transform method identifies missing values in the features DataFrame and fills them using the KNN imputation method.

**-Step 3: Update the Dataset**

The data.update function replaces the original dataset's missing values with the imputed values from data\_filled.

**-K-Nearest Neighbors (KNN) for Imputation**

KNN imputation is a data preprocessing technique where missing values are estimated based on the values of their nearest neighbors in the feature space. It works as follows:

Finding Nearest Neighbors:

For each data point with a missing value, the algorithm identifies k nearest data points (neighbors) that have complete data. Neighbors are determined using a distance metric (e.g., Euclidean distance) in the feature space.

Calculating the Imputed Value:

The missing value is replaced with the average (or median) of the corresponding values of these k nearest neighbors.

**Why KNN Imputation is Effective**

KNN imputation is particularly useful because:

It leverages relationships between features, making it suitable for datasets with interdependent variables.

By using nearest neighbors, it maintains the local structure of the data, ensuring realistic imputations.

It avoids the oversimplification of using a single global statistic (like mean) and adapts to local patterns in the data.

In this context, KNN imputation is especially relevant as the missing values in the Bare Nuclei column could depend on other features like Uniformity of Cell Size or Marginal Adhesion, making it a more sophisticated approach compared to simpler methods like mean imputation.

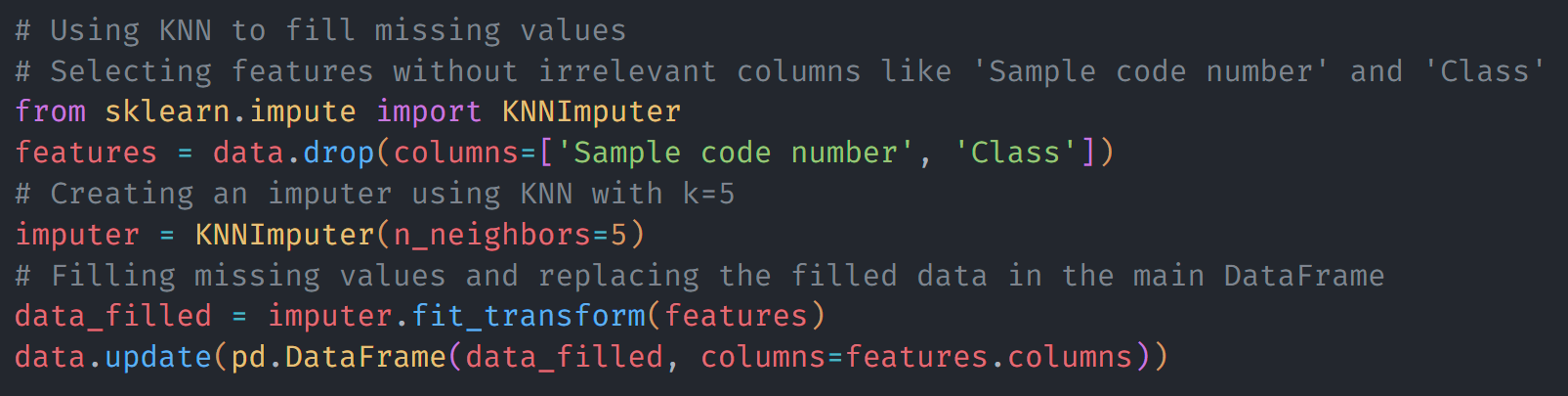


Figure 7 - Code for Using K-Nearest Neighbors (KNN)

## Feature Normalization

The dataset’s features (excluding the target variable `Class` and `Sample code number`) are scaled to a range between 0 and 1 using `MinMaxScaler` in Figure 8. Normalization ensures that features with different scales don’t disproportionately influence the model.

The MinMaxScaler is a method for scaling features to a fixed range, usually [0, 1]. The formula used by MinMaxScaler to scale a feature x is:

Where:

is the original feature value,

is the minimum value of the feature,

is the maximum value of the feature,

is the scaled feature value in the range [0, 1].

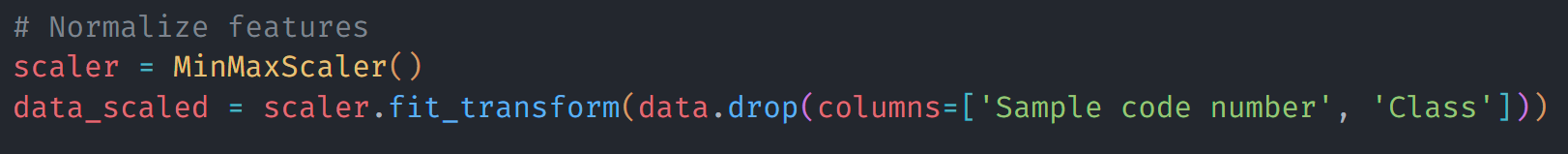


Figure 8 - Code for Feature Normalization

## Target Variable Encoding

The `Class` column, originally labeled as `2` (benign) and `4` (malignant), is encoded to `0` and `1` respectively using `LabelEncoder` in Figure 9. This conversion standardizes the target variable for binary classification.

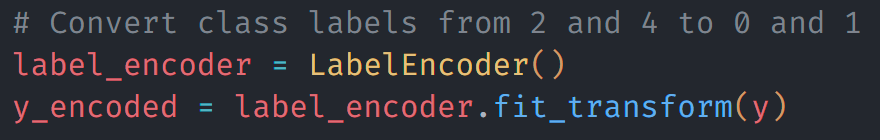


Figure 9 - Code for Target Variable Encoding

## Handling Class Imbalance

The dataset is checked for class imbalance before and after applying Synthetic Minority Oversampling Technique (SMOTE).

Step A: Checking Class Distribution

In Figure 10, this code calculates the count of samples in each class. A significant difference between the counts may indicate an imbalance, which can lead to biased model performance.

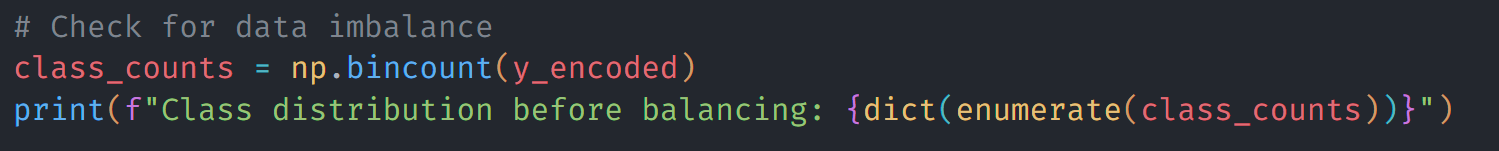


Figure 10 - Code for Checking Class Distribution

Step B: Applying SMOTE

SMOTE generates synthetic samples for the minority class, balancing the class distribution in Figure 11. This technique is particularly useful when class imbalance could lead to a model favoring the majority class.

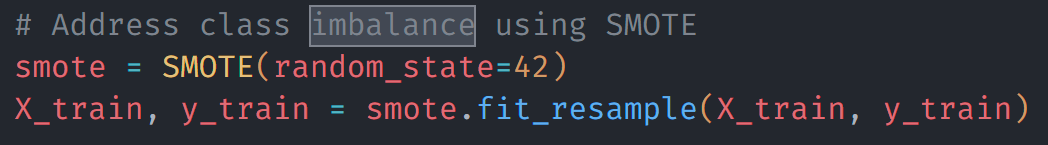


Figure 11 - Code for Applying SMOTE

Step C: Verifying Balanced Distribution

This ensures the resampled dataset has an equal or near-equal distribution of classes in Figure 12.

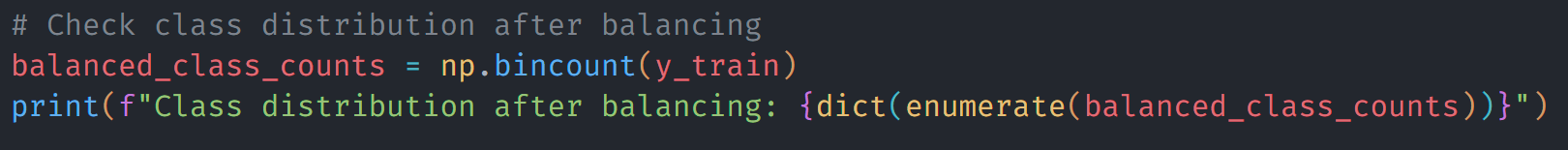


Figure 12 - Code for Verifying Balanced Distribution

## Dataset Splitting

The dataset is split into training (70%) and testing (30%) subsets to train the model and evaluate its performance. The `random\_state` ensures reproducibility in Figure 13.

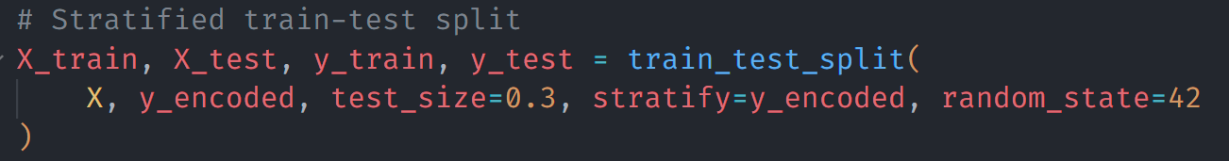


Figure 13 - Code for Dataset Splitting

## The Reason for These Steps

1. Imputation Choices:

- Missing values can distort analyses and model performance. Each imputation method explores a different way to mitigate its impact:

- Mean: Simplistic, assumes the missing data is randomly distributed.

- Class-Wise Mean: Incorporates class-specific information.

- KNN: Leverages patterns within the dataset for accurate estimation.

2. Normalization:

- Ensures all features contribute equally to the model by removing the influence of differing scales.

3. Label Encoding:

- Converts the target variable into a format suitable for binary classification.

4. Addressing Imbalance:

- Balancing class distributions prevents the model from being biased toward the majority class, ensuring fairer predictions.

5. Splitting Data:

- Training and testing on separate subsets ensures unbiased evaluation of the model.

# Implementation of Decision Tree Algorithms on Data

## Evaluation of Machine Learning Models on Imbalanced Data with Different Imputation Techniques

In this experiment, I evaluate the performance of several machine learning models on a dataset with missing values and class imbalances. To handle the missing data, I employ different imputation techniques, including filling missing values with the overall mean, using the class-wise mean, and leveraging K-Nearest Neighbors (KNN) imputation. Additionally, I explore the impact of addressing class imbalance through methods like SMOTE by comparing results with and without class balancing. The models tested include Decision Trees, AdaBoost, XGBoost, Random Forest, LightGBM, Extra Trees, and Gradient Boosting. Each model is trained and evaluated under various conditions, with a 70% training and 30% testing split, to assess how the different preprocessing strategies affect model accuracy and performance.

1. With Checking Data Imbalance - Using the Overall Mean

In this approach, I fill in missing values in the dataset using the overall mean of the feature. I then split the data into training and testing sets, where 70% of the data is used for training and 30% is used for testing. Additionally, I check for any class imbalance before training the model. This means I examine the distribution of classes (Class 0 and Class 1) and apply techniques such as SMOTE to balance the class distribution if needed, ensuring that the model does not bias towards the majority class. The models are then trained using the filled data, and the accuracy is evaluated.

2. Without Checking Data Imbalance - Using the Overall Mean

In this scenario, I follow the same steps as the previous approach but do not check for class imbalance. While I still fill the missing values with the overall mean of each feature, I do not apply any balancing techniques like SMOTE to adjust the class distribution. The class distribution may remain skewed, which could impact the model's performance, especially if the classes are imbalanced. The model is trained and evaluated using the filled data, and accuracy is recorded.

3. With Checking Data Imbalance - Using Class-Wise Mean

Here, I fill in the missing values in the dataset using the mean value of each class rather than the overall mean. This approach ensures that missing values are replaced with the mean specific to each class, which can provide more accurate imputations for each class. Afterward, I split the dataset into training and testing sets, with 70% for training and 30% for testing. Class imbalance is then checked, and techniques like SMOTE are applied if necessary to balance the distribution of classes. The model is trained on the class-wise imputed data, and the accuracy is calculated.

4. Without Checking Data Imbalance - Using Class-Wise Mean

In this case, I fill in the missing values using the class-wise mean (as in the previous method), but without checking for or addressing class imbalance. This means the data is split into training and testing sets, and the missing values are replaced with the mean value of each respective class. However, unlike the previous case, no balancing techniques like SMOTE are applied, which could result in the model being biased towards the majority class if there is an imbalance in the dataset. The model is trained and accuracy is calculated based on the filled dataset.

5. With Checking Data Imbalance - Using KNN

For this approach, I use KNN imputation to fill the missing values in the dataset. The KNN imputer replaces missing values by predicting them based on the nearest neighbors (k=5 by default) of the data points. After imputation, I split the data into training and testing sets (70% for training and 30% for testing). I also check for class imbalance and apply balancing techniques like SMOTE if necessary. The model is trained on the KNN-imputed dataset, and the performance is evaluated based on accuracy.

6. Without Checking Data Imbalance - Using KNN

In this approach, I follow the same KNN imputation strategy for missing values as in the previous method, but I do not check for class imbalance. The KNN imputer is used to predict and fill the missing values by considering the closest neighbors of each sample. The dataset is split into 70% for training and 30% for testing, but no balancing techniques are applied to the class distribution. This could lead to biased model performance if there is a class imbalance in the data. After training the model, I calculate its accuracy.

## CART (Classification and Regression Tree)

The CART model, implemented by the `DecisionTreeClassifier()`, is a fundamental machine learning algorithm used for both classification and regression tasks. In classification, it splits the data into subsets based on feature values that maximize the homogeneity of the target variable within each subset. The algorithm continues to split the data recursively until the tree reaches a predefined stopping condition, such as a maximum depth or minimum number of samples per leaf. The default criterion used by CART is the Gini impurity, which measures the level of impurity or disorder in the data at each split. The model creates a binary tree structure, where each node represents a feature decision, and the leaves represent the target class. I use this model as a baseline to evaluate how well decision tree-based classifiers can fit and predict the data.

Table 1 presents the performance metrics of the CART (Classification and Regression Tree) model, evaluated using different imputation techniques and data balancing methods. The table includes key evaluation metrics: Accuracy, Recall, Precision, and F1 Score, which provide a comprehensive view of the model's performance across different conditions.

1. With Checking Data Imbalance - Using the Overall Mean: This method involves filling missing values with the overall mean of the column while also considering class imbalance during model training. The CART model achieved an accuracy of 94.76%, with a recall of 93.06%, a precision of 91.78%, and an F1 score of 92.41%. This shows a solid balance between identifying true positives (recall) and minimizing false positives (precision).

2. Without Checking Data Imbalance - Using the Overall Mean: In this scenario, missing values are filled with the overall mean, but class imbalance is not considered during training. The CART model achieved an accuracy of 94.29%, with recall and precision both at 91.67%, resulting in an F1 score of 91.67%. While the accuracy is similar to the previous method, the lack of imbalance handling slightly affects the recall and precision.

3. With Checking Data Imbalance - Using Class-Wise Mean: Here, missing values are imputed using the mean for each class, and class imbalance is checked during training. The CART model shows an accuracy of 93.81%, with a high recall of 94.44% but a slightly lower precision of 88.31%. This results in an F1 score of 91.28%, indicating that the model is very good at identifying positive cases (high recall) but has a slightly lower precision.

4. Without Checking Data Imbalance - Using Class-Wise Mean: This method involves using class-specific means for imputation, but without considering class imbalance during training. The CART model shows the highest accuracy at 95.24%, with a recall of 94.44%, a precision of 91.89%, and an F1 score of 93.15%. This configuration achieves a good balance between identifying true positives and minimizing false positives.

5. With Checking Data Imbalance - Using KNN: This approach uses the K-Nearest Neighbors (KNN) algorithm to fill missing values and checks for class imbalance during model training. The CART model's accuracy drops to 91.90%, with a recall of 90.28%, a precision of 86.67%, and an F1 score of 88.44%. This method produces lower metrics compared to other imputation techniques, likely due to the complexity of KNN imputation.

6. Without Checking Data Imbalance - Using KNN: Finally, KNN imputation is used without addressing class imbalance. The CART model achieves an accuracy of 92.86%, with a recall of 93.06%, a precision of 87.01%, and an F1 score of 89.93%. This method slightly improves on the "With Checking Data Imbalance - Using KNN" approach but still lags behind the other methods in terms of performance.

In summary, imputation techniques and handling class imbalance significantly influence the model's performance. The highest performance for CART was achieved when using the class-wise mean imputation without considering data imbalance, showing that class balancing and thoughtful imputation can positively impact model metrics.

Table 1 - The Performance Metrics Of The Cart Using Different Imputation Techniques

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Imputation Techniques | Model | Accuracy | Recall | Precision | F1 Score |
| With Checking Data Imbalance - Using the Overall Mean | CART | 0/94762 | 0/93056 | 0/91781 | 0/92414 |
| Without Checking Data Imbalance - Using the Overall Mean | CART | 0/94286 | 0/91667 | 0/91667 | 0/91667 |
| With Checking Data Imbalance - Using Class-Wise Mean | CART | 0/93810 | 0/94444 | 0/88312 | 0/91275 |
| Without Checking Data Imbalance - Using Class-Wise Mean | CART | **0/95238** | 0/94444 | 0/91892 | 0/93151 |
| With Checking Data Imbalance - Using KNN | CART | 0/91905 | 0/90278 | 0/86667 | 0/88435 |
| Without Checking Data Imbalance - Using KNN | CART | 0/92857 | 0/93056 | 0/87013 | 0/89933 |

Figures 14 and 15 show the decision tree generated by the CART algorithm for the dataset. This tree corresponds to the model with the best accuracy among all CART configurations tested, achieving an accuracy of 0.95238. The tree uses a hierarchical structure of splitting conditions based on the features in the dataset, dividing the data into nodes until a classification decision is made.

Each branch of the tree represents a splitting rule, which tests the value of a specific feature. A path from the root to a leaf node constitutes a complete rule, where the conditions along the path are combined to form the final decision.

In this particular tree, there are a total of 67 rules. These rules correspond to the unique paths from the root node to each leaf node. Each rule contributes to the classification of the data points into their respective classes. The high number of rules reflects the complexity of the decision-making process and the model's ability to capture intricate patterns in the dataset.

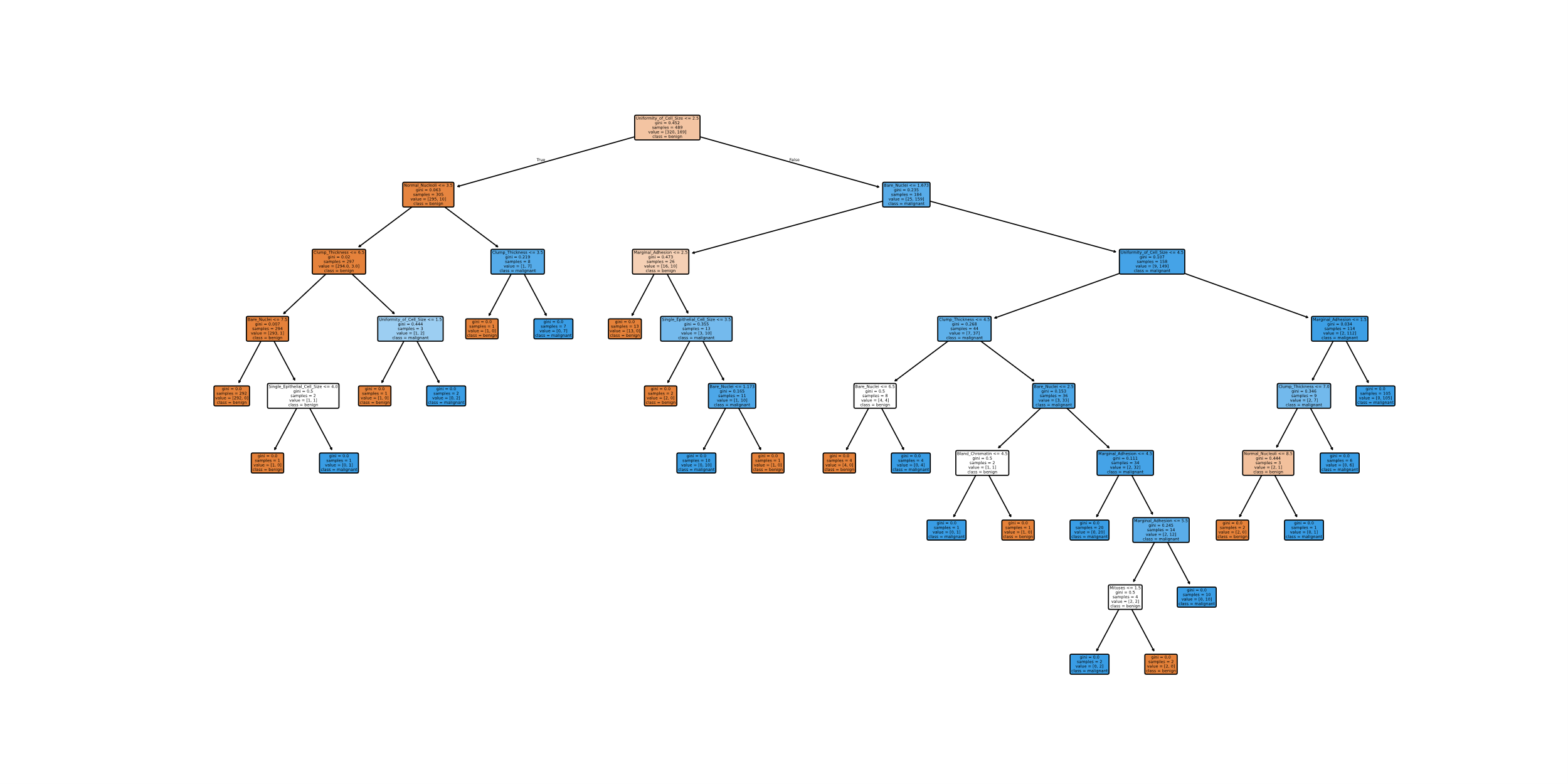


Figure 14 - The Best Decision Tree Generated By The CART Algorithm

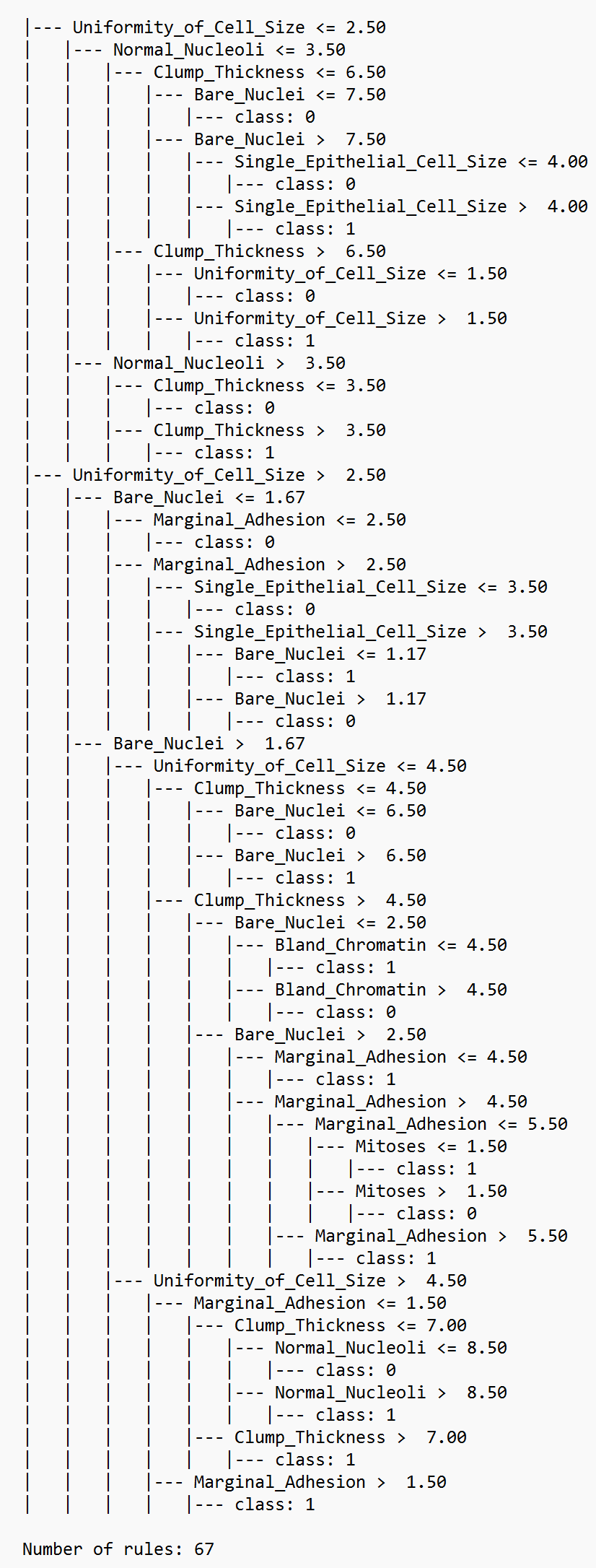


Figure 15 - Rules of The Best Decision Tree Generated By The CART Algorithm

The CART algorithm achieves a balance between correctly classifying both classes, with 132 true negatives and 68 true positives indicating strong predictive performance for both in Figure 16. However, it misclassifies 6 instances of Class 0 as Class 1 and 4 as Class 1 as Class 0. These misclassifications slightly lower the precision and recall for both classes, but the algorithm still demonstrates reliable overall performance with a well-calibrated model.

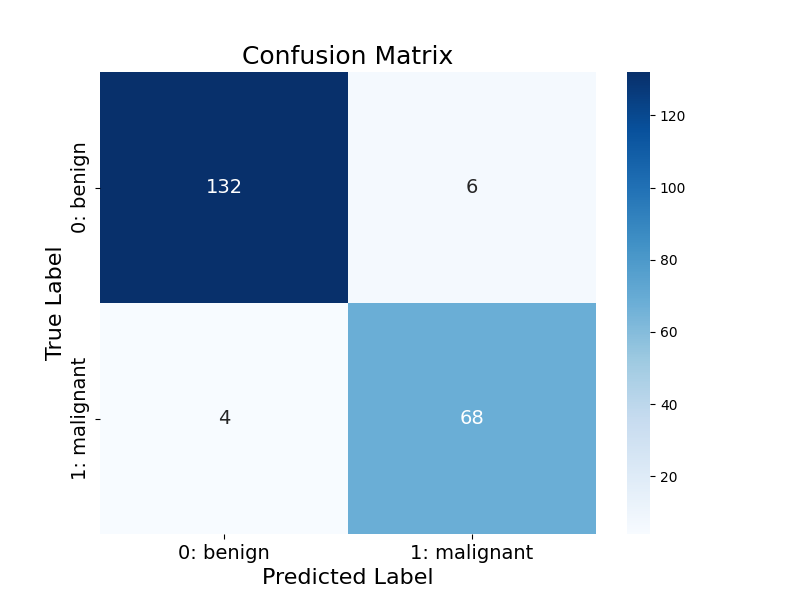


Figure 16 - The Confusion Matrix of Cart Algorithm

## C4.5

The C4.5 model is another decision tree classifier, but it specifically uses the Gini criterion for splitting the data (as specified by `criterion='gini'`). C4.5 is an extension of the CART model, and it handles both categorical and continuous attributes. In this version, the tree-building process selects splits that maximize the information gain, but instead of using entropy like the original C4.5 algorithm, I use Gini impurity, which is computationally simpler. This model can be particularly useful when dealing with imbalanced datasets, as it tends to build deeper trees to minimize error. By adjusting the criterion, I can assess how the choice of impurity function influences the accuracy and generalization of the model.

Table 2 shows the performance of the C4.5 decision tree model, evaluated with various imputation techniques and handling of class imbalance. The metrics presented for each configuration include Accuracy, Recall, Precision, and F1 Score, which provide a complete assessment of the model’s effectiveness.

1. With Checking Data Imbalance - Using the Overall Mean: The C4.5 model achieved an accuracy of 95.24%, with a recall of 94.44%, a precision of 91.89%, and an F1 score of 93.15%. This approach handles missing values by filling them with the overall mean and considers class imbalance during training. The high accuracy and recall suggest that the model is good at correctly identifying the positive class, while the precision shows it also performs well in minimizing false positives.

2. Without Checking Data Imbalance - Using the Overall Mean: When the class imbalance is not addressed, the C4.5 model achieved a slightly lower accuracy of 93.81%, with a recall of 91.67%, precision of 90.41%, and an F1 score of 91.03%. The lack of class imbalance handling affects the recall, which drops compared to the previous configuration. However, the precision remains relatively strong, suggesting that the model maintains a balance between true positives and false positives.

3. With Checking Data Imbalance - Using Class-Wise Mean: In this case, missing values are imputed using the mean of each class, and the model takes class imbalance into account. The C4.5 model shows an accuracy of 94.29%, with a recall of 95.83%, a precision of 88.46%, and an F1 score of 92. This configuration achieves the highest recall, indicating that the model is particularly good at identifying positive cases, but the precision is somewhat lower compared to other methods. This is a trade-off between maximizing true positives (high recall) and minimizing false positives (lower precision).

4. Without Checking Data Imbalance - Using Class-Wise Mean: This method uses class-specific means for imputation but does not address class imbalance. The C4.5 model performs well, achieving an accuracy of 95.71%, with a recall of 94.44%, a precision of 93.15%, and an F1 score of 93.79%. This method strikes a better balance between recall and precision, leading to a higher overall F1 score compared to the previous methods, and highlights the importance of using class-wise imputation when class imbalance is not considered.

5. With Checking Data Imbalance - Using KNN: Using K-Nearest Neighbors (KNN) for imputation and handling class imbalance, the C4.5 model has an accuracy of 93.33%, with a recall of 94.44%, precision of 87.18%, and an F1 score of 90.67%. While the recall remains high, the precision drops significantly, suggesting that KNN imputation might introduce some noise, leading to more false positives.

6. Without Checking Data Imbalance - Using KNN: Finally, when using KNN imputation without addressing class imbalance, the C4.5 model again achieves an accuracy of 93.33%, with a recall of 93.06%, precision of 88.16%, and an F1 score of 90.54%. While the recall is similar to the previous configuration, the precision is slightly improved, but the overall performance is still below other methods that used class-wise mean imputation.

In conclusion, the C4.5 model's performance is heavily influenced by the imputation method and whether class imbalance is handled. Using class-wise mean imputation (especially with data imbalance checking) gives the best recall, though this may come at the cost of precision. When the class imbalance is not handled, the model tends to perform better in terms of precision, but recall and F1 score may suffer slightly. The KNN imputation method tends to result in lower precision and should be used carefully.

Table 2 - The Performance Metrics Of The C4.5 Using Different Imputation Techniques

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Imputation Techniques | Model | Accuracy | Recall | Precision | F1 Score |
| With Checking Data Imbalance - Using the Overall Mean | C4.5 | 0/95238 | 0/94444 | 0/91892 | 0/93151 |
| Without Checking Data Imbalance - Using the Overall Mean | C4.5 | 0/93810 | 0/91667 | 0/90411 | 0/91034 |
| With Checking Data Imbalance - Using Class-Wise Mean | C4.5 | 0/94286 | 0/95833 | 0/88462 | 0/92000 |
| Without Checking Data Imbalance - Using Class-Wise Mean | C4.5 | **0/95714** | 0/94444 | 0/93151 | 0/93793 |
| With Checking Data Imbalance - Using KNN | C4.5 | 0/93333 | 0/94444 | 0/87179 | 0/90667 |
| Without Checking Data Imbalance - Using KNN | C4.5 | 0/93333 | 0/93056 | 0/88158 | 0/90541 |

Figures 17 and 18 illustrate the decision tree generated by the C4.5 algorithm for the dataset. This tree corresponds to the model with the highest accuracy among all C4.5 configurations tested, achieving an accuracy of 0.95714. The tree uses entropy-based splitting to partition the data, selecting features that maximize information gain at each step.

Each branch in the tree represents a conditional rule derived from the dataset's features. A path from the root node to a leaf node constitutes a complete rule, describing the decision-making process for classifying data points.

This C4.5 tree contains 67 rules, representing the unique paths from the root to the leaf nodes. These rules capture the complexity of the dataset while maintaining a balance between accuracy and generalizability. The high accuracy achieved demonstrates the algorithm's effectiveness in identifying meaningful patterns within the data.

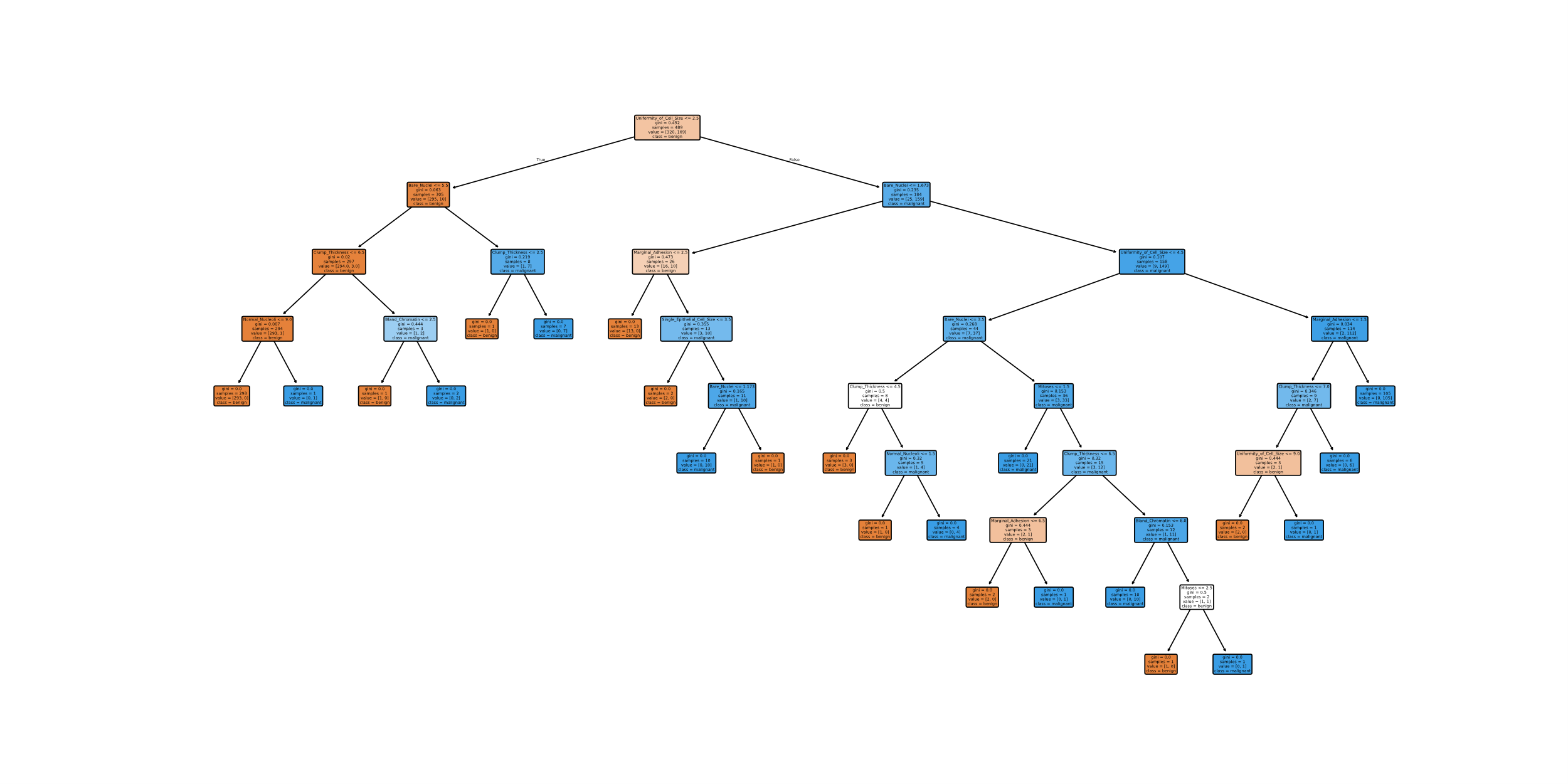


Figure 17 - The Best Decision Tree Generated By The C4.5 Algorithm

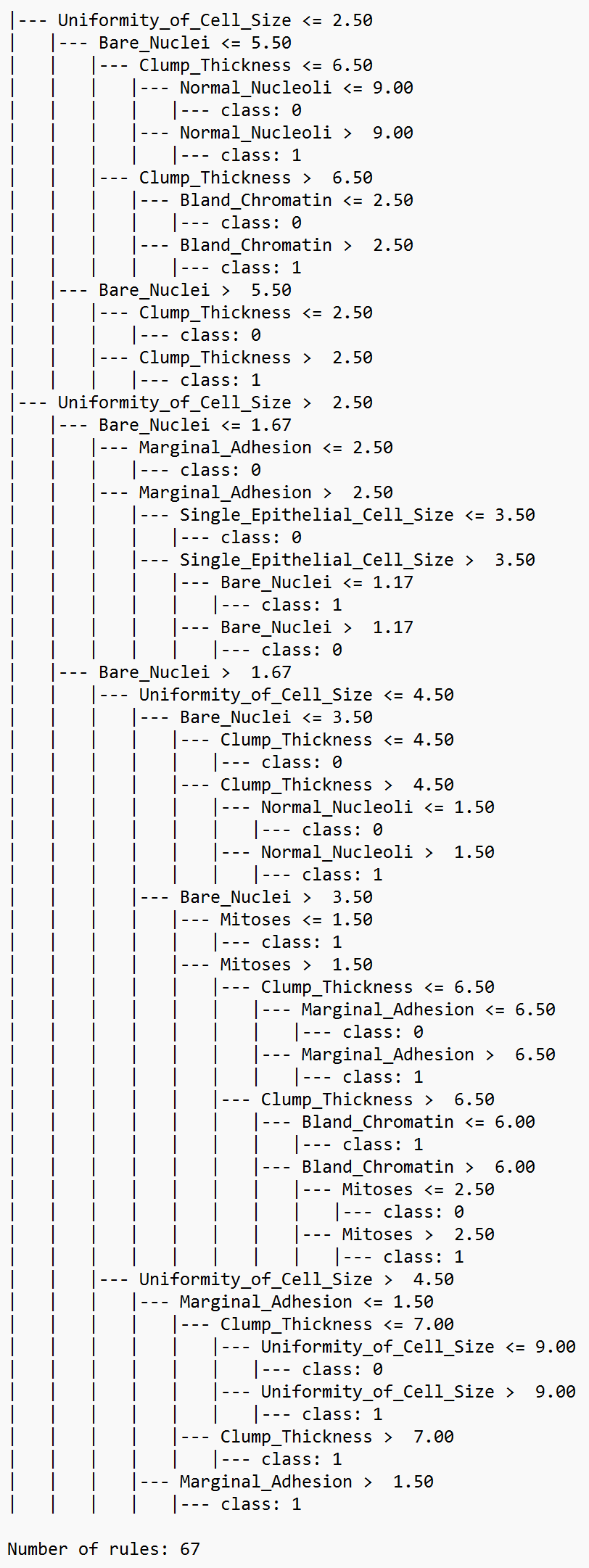


Figure 18 - Rules of The Best Decision Tree Generated By The C4.5 Algorithm

C4.5 shows slight improvement over CART by reducing false positives to 5, resulting in 133 true negatives in Figure 19. The true positives remain at 68, while false negatives stay consistent at 4. This improvement in false positives reflects a better balance in handling Class 0, contributing to slightly higher precision for this class. Overall, C4.5 maintains high accuracy and demonstrates better discrimination in predicting Class 0 correctly.

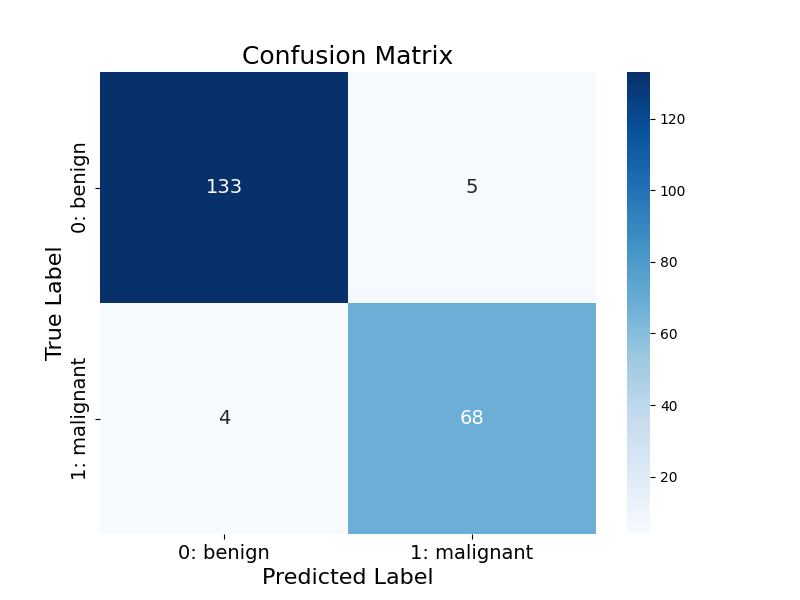


Figure 19 - The Confusion Matrix of C4.5 Algorithm

## AdaBoost

AdaBoost (Adaptive Boosting) is an ensemble learning algorithm that combines the predictions of multiple weak classifiers to create a stronger overall model. I use the `AdaBoostClassifier()` with the SAMME algorithm, which stands for Stagewise Additive Modeling using a Multiclass Exponential loss function. In AdaBoost, each weak learner is trained sequentially, where each subsequent model focuses on correcting the errors made by previous ones. The algorithm assigns higher weights to misclassified samples so that future classifiers focus more on them. I use AdaBoost because it can effectively improve the performance of weak classifiers, especially in cases where individual models are not highly accurate but can benefit from boosting techniques.

Table 3 presents the performance of the AdaBoost model under different conditions for imputation and class imbalance handling. The evaluation metrics include Accuracy, Recall, Precision, and F1 Score, which reflect the model’s overall ability to make correct predictions and minimize errors.

1. With Checking Data Imbalance - Using the Overall Mean: The AdaBoost model achieves an accuracy of 95.71%, with a recall of 94.44%, precision of 93.15%, and an F1 score of 93.79%. This configuration handles class imbalance and uses the overall mean to fill in missing values. The performance indicates that the model is quite balanced in terms of recall and precision, resulting in a strong F1 score.

2. Without Checking Data Imbalance - Using the Overall Mean: When class imbalance is not addressed, the AdaBoost model shows a slight drop in performance, with an accuracy of 94.76%, recall of 91.67%, precision of 92.96%, and an F1 score of 92.31%. The accuracy is still high, but the recall decreases, indicating that the model is missing some positive cases without the class imbalance correction.

3. With Checking Data Imbalance - Using Class-Wise Mean: Using class-wise mean imputation and addressing class imbalance, the AdaBoost model performs the best, with an accuracy of 96.19%, recall of 95.83%, precision of 93.24%, and an F1 score of 94.52%. This approach enhances recall significantly, while the precision remains strong, resulting in the highest F1 score among all configurations.

4. Without Checking Data Imbalance - Using Class-Wise Mean: When class imbalance is not taken into account, and class-wise means are used for imputation, the model achieves an accuracy of 95.24%, recall of 93.06%, precision of 93.06%, and a F1 score of 93.06%. This configuration still performs well but shows a slight decrease in recall compared to when class imbalance is checked, indicating the importance of class imbalance correction for better performance.

5. With Checking Data Imbalance - Using KNN: The AdaBoost model with KNN imputation and class imbalance correction achieves an accuracy of 95.71%, recall of 95.83%, precision of 92.00%, and an F1 score of 93.88%. The recall remains high, indicating effective identification of positive cases, but the precision is slightly lower compared to the class-wise mean method.

6. Without Checking Data Imbalance - Using KNN: Without handling class imbalance, the AdaBoost model performs similarly to the previous KNN configuration, achieving an accuracy of 94.76%, recall of 91.67%, precision of 92.96%, and an F1 score of 92.31%. This configuration shows a decrease in recall compared to when data imbalance is checked, highlighting the effect of class imbalance on the model's ability to identify the minority class.

In summary, the AdaBoost model performs best when both class imbalance is addressed and class-wise mean imputation is used, with the highest accuracy and F1 score. The KNN imputation method, while still effective, results in slightly lower precision and F1 score. Additionally, without addressing class imbalance, the model shows a decrease in recall, emphasizing the importance of considering class imbalance for optimal performance.

Table 3 - The Performance Metrics Of The AdaBoost Using Different Imputation Techniques

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Imputation Techniques | Model | Accuracy | Recall | Precision | F1 Score |
| With Checking Data Imbalance - Using the Overall Mean | AdaBoost | 0/95714 | 0/94444 | 0/93151 | 0/93793 |
| Without Checking Data Imbalance - Using the Overall Mean | AdaBoost | 0/94762 | 0/91667 | 0/92958 | 0/92308 |
| With Checking Data Imbalance - Using Class-Wise Mean | AdaBoost | **0/96190** | 0/95833 | 0/93243 | 0/94521 |
| Without Checking Data Imbalance - Using Class-Wise Mean | AdaBoost | 0/95238 | 0/93056 | 0/93056 | 0/93056 |
| With Checking Data Imbalance - Using KNN | AdaBoost | 0/95714 | 0/95833 | 0/92000 | 0/93878 |
| Without Checking Data Imbalance - Using KNN | AdaBoost | 0/94762 | 0/91667 | 0/92958 | 0/92308 |

Figures 20 and 21 show the decision tree produced by the AdaBoost algorithm for the dataset. This configuration achieved the highest accuracy among all AdaBoost setups, with a remarkable accuracy of 0.96190. Unlike standalone decision trees, AdaBoost combines multiple weak learners (simple decision trees) into a single, robust model through an iterative weighting process.

The final ensemble effectively classifies data points based on the cumulative decisions of the weak learners. For this setup, the model uses only 4 rules, corresponding to the distinct decision boundaries created by the weak learners. These concise rules highlight AdaBoost’s strength in minimizing overfitting while maximizing classification performance.

The exceptional accuracy of this AdaBoost configuration illustrates its ability to distill complex patterns into a small number of impactful rules, making it both efficient and powerful.

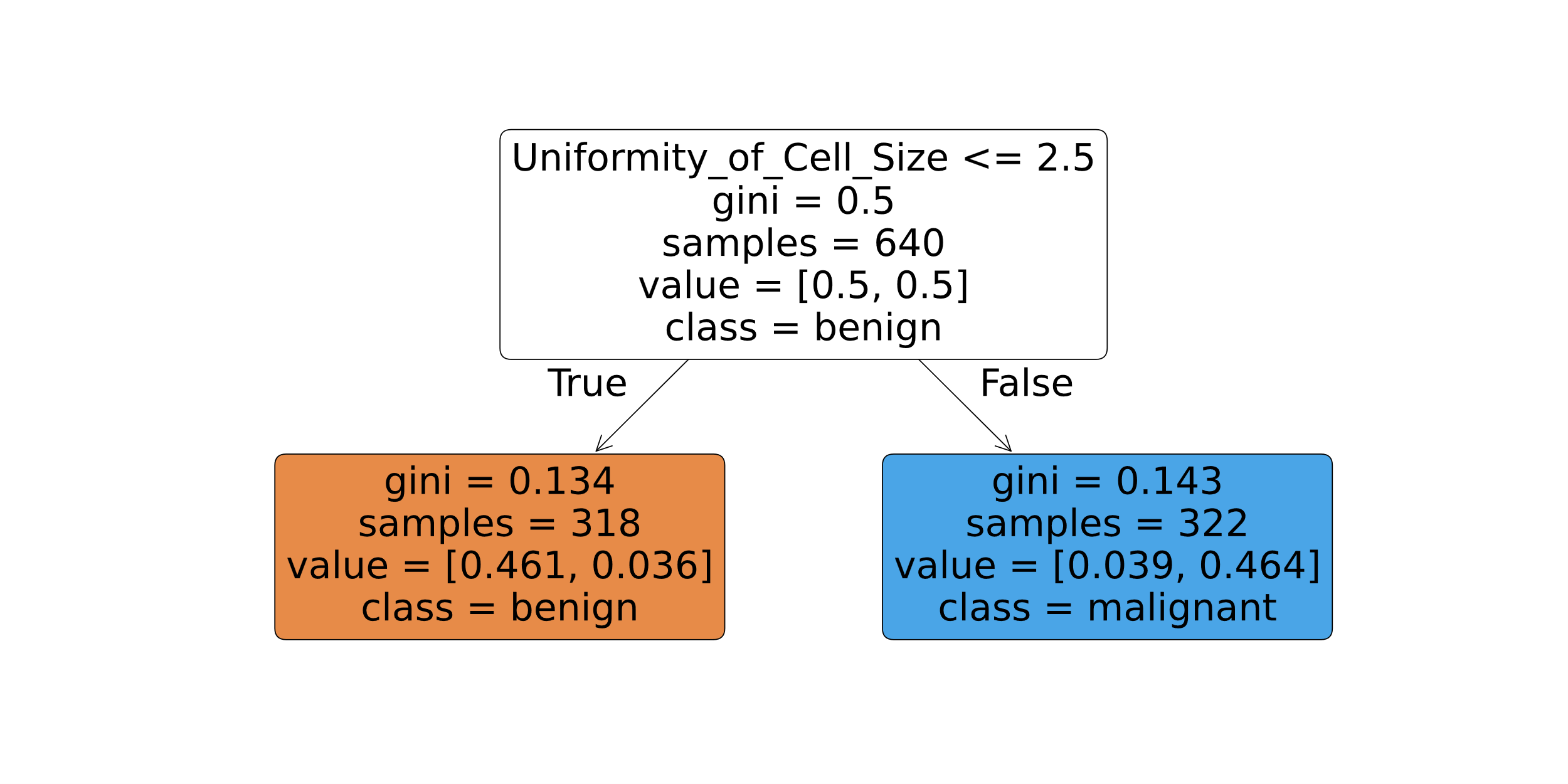


Figure 20 - The Best Decision Tree Generated By The AdaBoost Algorithm

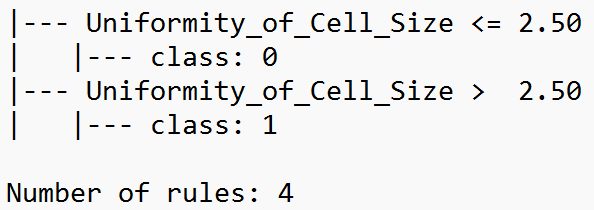


Figure 21 - Rules of The Best Decision Tree Generated By The AdaBoost Algorithm

AdaBoost outperforms CART and C4.5 by further reducing false negatives to 3, increasing true positives to 69 while maintaining 133 true negatives and 5 false positives in Figure 22. This indicates that AdaBoost is particularly effective at identifying Class 1, leading to higher recall and precision for this class. Its robust performance highlights the strength of ensemble methods in improving classification for minority classes.

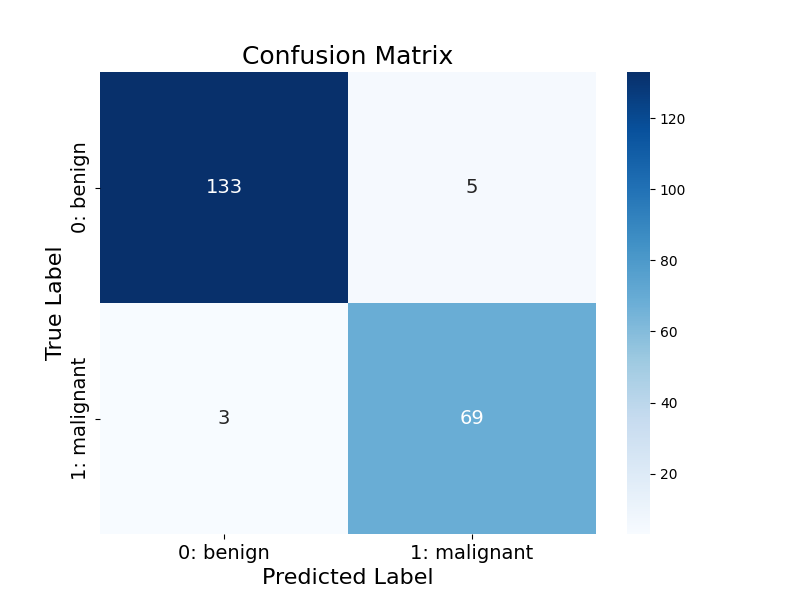


Figure 22 - The Confusion Matrix of AdaBoost Algorithm

## XGBoost

XGBoost (Extreme Gradient Boosting) is a powerful and highly efficient gradient boosting framework that I use with the `XGBClassifier()`. The primary advantage of XGBoost is its scalability and regularization techniques that help prevent overfitting. In my case, I set the evaluation metric to `logloss`, which is commonly used for binary classification tasks, as it measures the performance of the model based on the likelihood of the predicted class. XGBoost builds an ensemble of decision trees where each tree corrects the mistakes made by the previous ones, and the model is trained using gradient boosting. It is one of the top-performing models in many machine learning competitions, thanks to its speed, efficiency, and handling of missing data and outliers.

Table 4 provides the performance metrics of the XGBoost model under different conditions for imputation and handling class imbalance. The evaluation metrics include Accuracy, Recall, Precision, and F1 Score, reflecting the model's overall predictive ability and effectiveness in managing imbalanced data.

1. With Checking Data Imbalance - Using the Overall Mean: When class imbalance is addressed and the overall mean is used for imputation, the XGBoost model performs well, achieving an accuracy of 95.71%, recall of 97.22%, precision of 90.91%, and an F1 score of 93.96%. The high recall indicates that the model is effective at identifying the minority class, but the precision is slightly lower, suggesting some false positives.

2. Without Checking Data Imbalance - Using the Overall Mean: When class imbalance is not considered, and the overall mean is used for imputation, the XGBoost model has an accuracy of 95.71%, recall of 95.83%, precision of 92.00%, and an F1 score of 93.88%. While accuracy remains the same as the previous configuration, recall is slightly reduced, indicating that the model may miss a few positive instances due to the class imbalance.

3. With Checking Data Imbalance - Using Class-Wise Mean: With class imbalance addressed and class-wise mean imputation used, the XGBoost model achieves an accuracy of 95.24%, recall of 95.83%, precision of 90.79%, and an F1 score of 93.24%. This configuration shows slightly lower performance in comparison to the overall mean method, particularly in precision, which is slightly reduced while maintaining high recall.

4. Without Checking Data Imbalance - Using Class-Wise Mean: When class imbalance is not handled and class-wise mean imputation is applied, the XGBoost model performs the best in terms of accuracy (96.19%), recall (97.22%), precision (92.11%), and F1 score (94.59%). This setup produces the highest accuracy and F1 score, suggesting that handling class imbalance improves performance when using class-wise mean imputation.

5. With Checking Data Imbalance - Using KNN: The XGBoost model with KNN imputation and class imbalance correction shows a high recall of 98.61%, accuracy of 96.19%, precision of 91.03%, and F1 score of 94.67%. This indicates that KNN imputation, combined with addressing data imbalance, leads to the best recall and F1 score, making it effective for the model to identify positive cases while maintaining a good balance between precision and recall.

6. Without Checking Data Imbalance - Using KNN: Without addressing class imbalance, but using KNN imputation, the XGBoost model's performance is slightly reduced, with an accuracy of 95.24%, recall of 95.83%, precision of 90.79%, and F1 score of 93.24%. The absence of class imbalance correction leads to a small reduction in recall and precision, similar to the overall mean and class-wise mean methods without imbalance checking.

In conclusion, the XGBoost model performs best when both class imbalance is handled and KNN imputation is applied, yielding the highest recall and F1 score. While class-wise mean imputation also yields good results, addressing class imbalance plays a crucial role in improving performance, especially for recall, which is vital for the model’s ability to identify minority class instances.

Table 4 - The Performance Metrics Of The XGBoost Using Different Imputation Techniques

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Imputation Techniques | Model | Accuracy | Recall | Precision | F1 Score |
| With Checking Data Imbalance - Using the Overall Mean | XGBoost | 0/95714 | 0/97222 | 0/90909 | 0/93960 |
| Without Checking Data Imbalance - Using the Overall Mean | XGBoost | 0/95714 | 0/95833 | 0/92000 | 0/93878 |
| With Checking Data Imbalance - Using Class-Wise Mean | XGBoost | 0/95238 | 0/95833 | 0/90789 | 0/93243 |
| Without Checking Data Imbalance - Using Class-Wise Mean | XGBoost | **0/96190** | 0/97222 | 0/92105 | 0/94595 |
| With Checking Data Imbalance - Using KNN | XGBoost | **0/96190** | 0/98611 | 0/91026 | 0/94667 |
| Without Checking Data Imbalance - Using KNN | XGBoost | 0/95238 | 0/95833 | 0/90789 | 0/93243 |

Figures 23 and 24 illustrate the decision tree generated by the XGBoost algorithm for the dataset. This configuration achieved an impressive accuracy of 0.96190, showcasing XGBoost's effectiveness in optimizing model performance through gradient boosting techniques. XGBoost excels in handling complex datasets by iteratively improving its predictions and minimizing errors.

Unlike traditional decision tree algorithms, XGBoost constructs an ensemble of trees, where each subsequent tree corrects the errors of the previous ones. This iterative process allows the model to capture intricate patterns in the data while maintaining high precision and recall across classes. XGBoost also incorporates regularization techniques to reduce overfitting, enhancing its generalization to unseen data.

The exceptional accuracy achieved with this setup highlights XGBoost's ability to balance efficiency and predictive power, making it a standout choice for datasets with challenging classification problems.

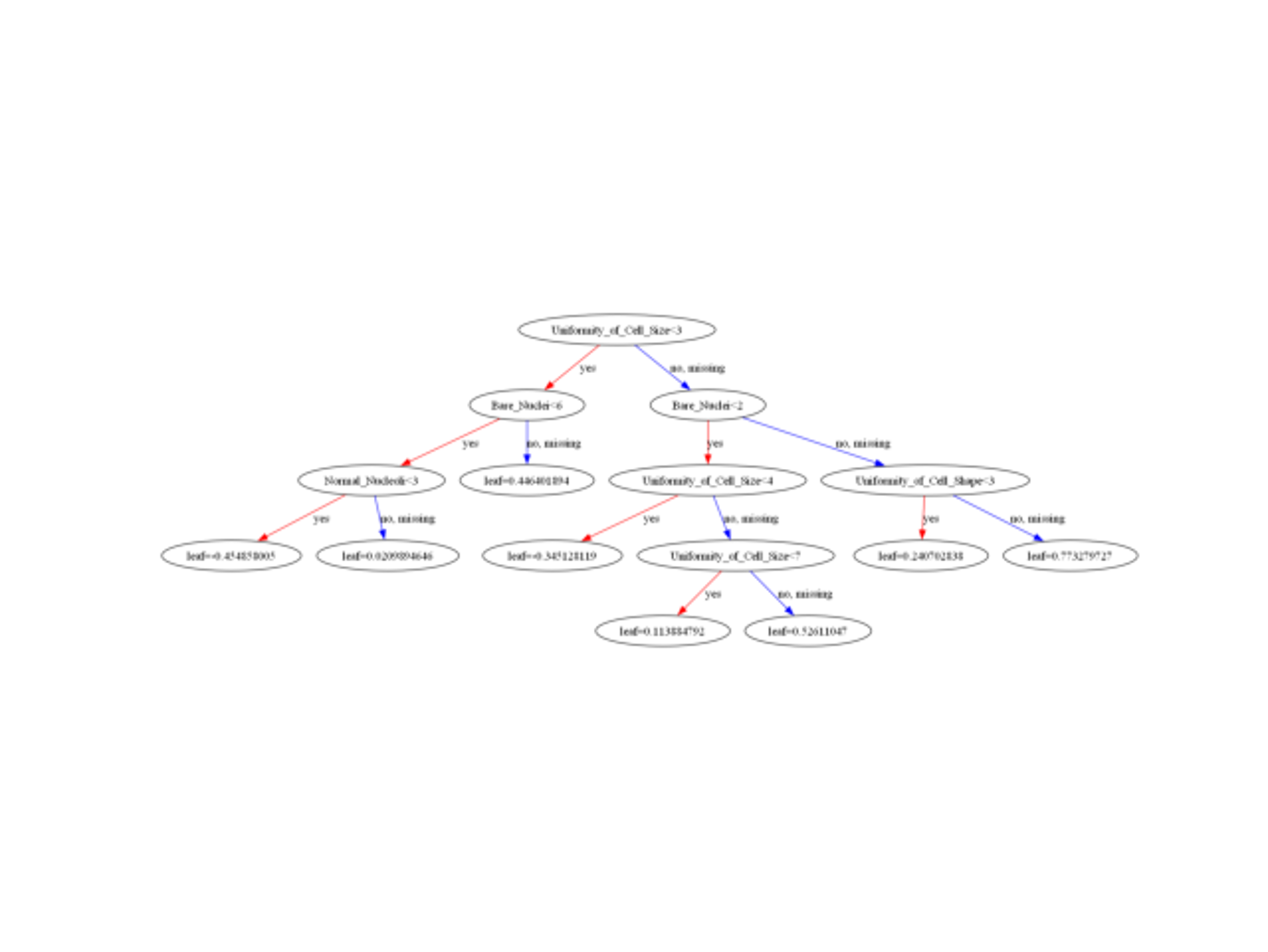


Figure 23 - One of The Best Decision Trees Generated By The XGBoost Algorithm

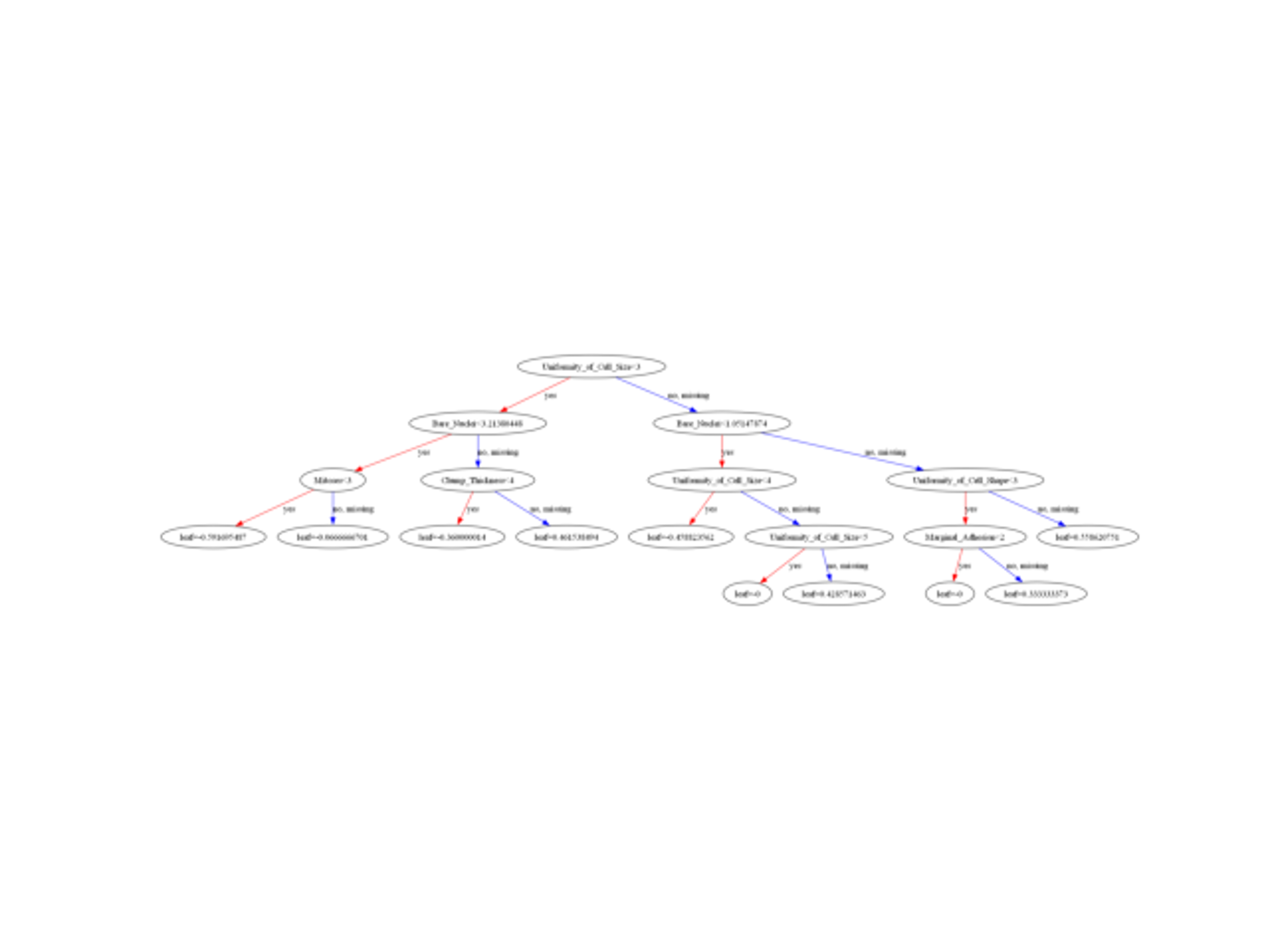


Figure 24 - One of The Best Decision Trees Generated By The XGBoost Algorithm

In Figure 25, the first XGBoost delivers impressive performance by reducing false negatives to **2** and increasing true positives to **70**. With **132 true negatives** and **6 false positives**, the algorithm balances the trade-off between the two classes well. This indicates that XGBoost1 identifies Class 1 instances while maintaining reliable predictions for Class 0, making it highly effective for imbalanced datasets.

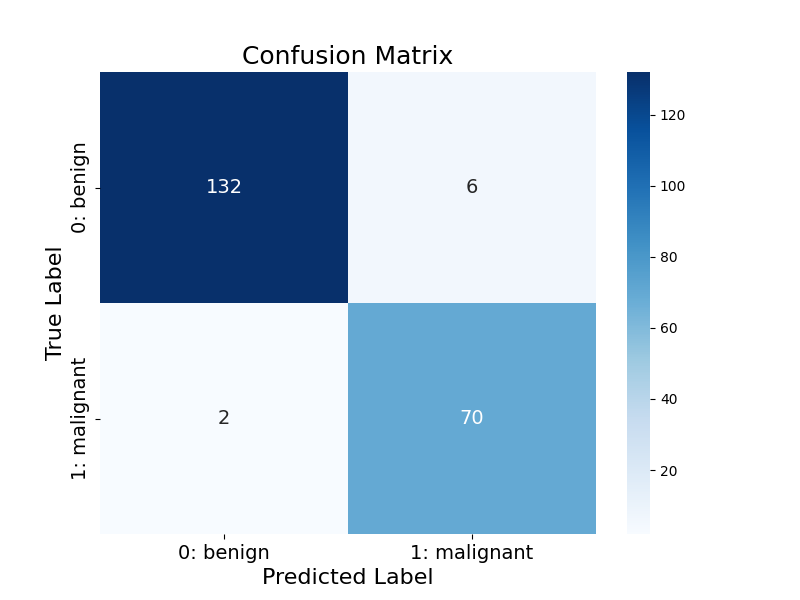


Figure 25 - The First Confusion Matrix of XGBoost Algorithm

In Figure 26, the second XGBoost achieves the highest recall for Class 1 by reducing false negatives to just 1, resulting in 71 true positives. However, this comes at the cost of slightly more false positives (7) and 131 true negatives. This trade-off reflects a model tuned to prioritize identifying Class 1 even if it slightly compromises predictions for Class 0. It is particularly suited for scenarios where correctly predicting Class 1 is critical.

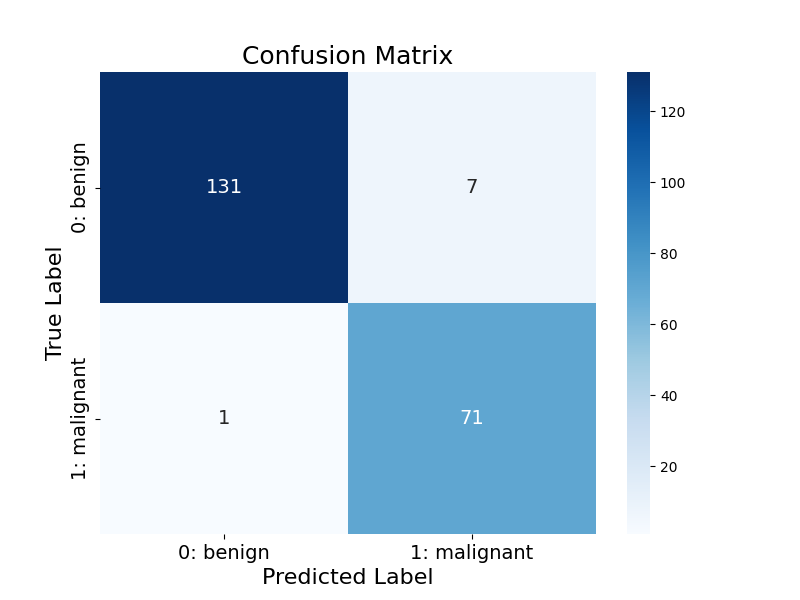


Figure 26 - The Second Confusion Matrix of XGBoost Algorithm

## Random Forest

The Random Forest model, created using the `RandomForestClassifier()`, is an ensemble method that constructs multiple decision trees during training. The model then makes predictions based on the majority vote of the individual trees, making it more robust and less prone to overfitting than a single decision tree. The key parameter I’ve set is n\_estimators=100, which determines the number of trees to build in the forest. The model uses random subsets of the data and features to build each tree, helping ensure diversity in the trees and increasing the generalization power of the model. I chose Random Forest because it handles large datasets well and provides high accuracy by averaging the results of multiple decision trees, making it less sensitive to noise in the data.

Table 5 presents the performance of the RandomForest model under different conditions, including the use of imputation methods and checking for class imbalance. The evaluation metrics include Accuracy, Recall, Precision, and F1 Score, providing insights into the model's overall effectiveness in handling the classification task.

1. With Checking Data Imbalance - Using the Overall Mean: When class imbalance is addressed and the overall mean is used for imputation, the RandomForest model performs excellently with an accuracy of 96.67%, recall of 98.61%, precision of 92.21%, and an F1 score of 95.30%. The high recall indicates that the model is very effective in identifying the minority class, with a solid balance between precision and recall, making it a reliable choice for imbalanced datasets.

2. Without Checking Data Imbalance - Using the Overall Mean: When class imbalance is not considered, but the overall mean is applied for imputation, the RandomForest model’s performance slightly decreases. It achieves an accuracy of 95.71%, recall of 94.44%, precision of 93.15%, and an F1 score of 93.79%. While still effective, the model’s recall drops, suggesting that not addressing data imbalance reduces the model's ability to detect the minority class effectively.

3. With Checking Data Imbalance - Using Class-Wise Mean: With class imbalance addressed and class-wise mean imputation, the RandomForest model performs similarly to the overall mean configuration, achieving an accuracy of 96.67%, recall of 97.22%, precision of 93.33%, and an F1 score of 95.24%. This configuration delivers excellent performance, with a slight improvement in recall compared to the overall mean approach, and a well-balanced precision score.

4. Without Checking Data Imbalance - Using Class-Wise Mean: When class imbalance is not handled and class-wise mean imputation is used, the RandomForest model shows similar performance to the case without imbalance checking and overall mean imputation. It achieves an accuracy of 95.71%, recall of 94.44%, precision of 93.15%, and an F1 score of 93.79%, again indicating that addressing class imbalance plays a critical role in improving recall.

5. With Checking Data Imbalance - Using KNN: The RandomForest model with KNN imputation and class imbalance correction shows the best performance with an accuracy of 97.14%, recall of 98.61%, precision of 93.42%, and an F1 score of 95.95%. This configuration achieves the highest recall and F1 score, demonstrating that KNN imputation, when combined with class imbalance handling, leads to the best overall model performance, especially in terms of detecting positive cases.

6. Without Checking Data Imbalance - Using KNN: Without addressing class imbalance, but using KNN imputation, the RandomForest model performs well, with an accuracy of 96.19%, recall of 95.83%, precision of 93.24%, and an F1 score of 94.52%. While performance remains strong, the model's recall is slightly reduced, reflecting the impact of not correcting for class imbalance.

In conclusion, the RandomForest model shows its best performance when class imbalance is addressed and KNN imputation is used, particularly in terms of recall and F1 score. While class-wise mean imputation also produces strong results, it is clear that addressing class imbalance is crucial for enhancing the model’s ability to identify the minority class effectively.

Table 5 - The Performance Metrics Of The Random Forest Using Different Imputation Techniques

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Imputation Techniques | Model | Accuracy | Recall | Precision | F1 Score |
| With Checking Data Imbalance - Using the Overall Mean | RandomForest | 0/96667 | 0/98611 | 0/92208 | 0/95302 |
| Without Checking Data Imbalance - Using the Overall Mean | RandomForest | 0/95714 | 0/94444 | 0/93151 | 0/93793 |
| With Checking Data Imbalance - Using Class-Wise Mean | RandomForest | 0/96667 | 0/97222 | 0/93333 | 0/95238 |
| Without Checking Data Imbalance - Using Class-Wise Mean | RandomForest | 0/95714 | 0/94444 | 0/93151 | 0/93793 |
| With Checking Data Imbalance - Using KNN | RandomForest | **0/97143** | 0/98611 | 0/93421 | 0/95946 |
| Without Checking Data Imbalance - Using KNN | RandomForest | 0/96190 | 0/95833 | 0/93243 | 0/94521 |

Figures 27 and 28 illustrate the model generated by the Random Forest algorithm for the dataset. This configuration achieved the highest accuracy among all setups, with an outstanding accuracy of 0.97143. Random Forest is a powerful ensemble learning method that combines multiple decision trees to enhance prediction accuracy and robustness.

The final ensemble classifies data points by aggregating the decisions of individual trees through a majority voting process. For this setup, the model employs 79 rules, representing the collective decision paths across all the trees in the forest. These rules highlight the model’s capacity to capture complex patterns while reducing variance and minimizing overfitting.

The exceptional accuracy of this Random Forest configuration underscores its strength in balancing diversity and depth among decision trees, making it an effective and reliable choice for high-dimensional classification tasks.

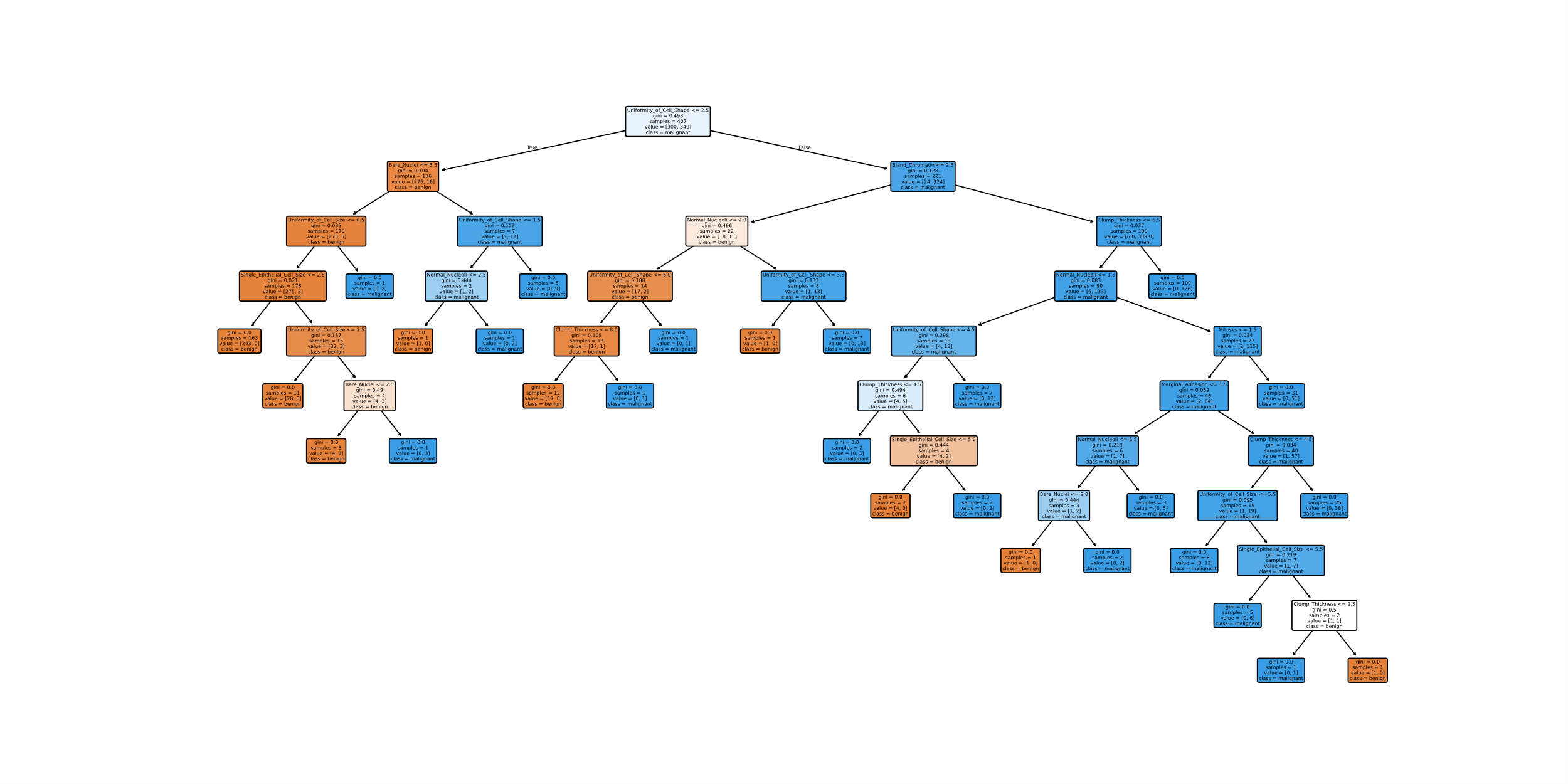


Figure 27 - The Best Decision Tree Generated By The Random Forest Algorithm

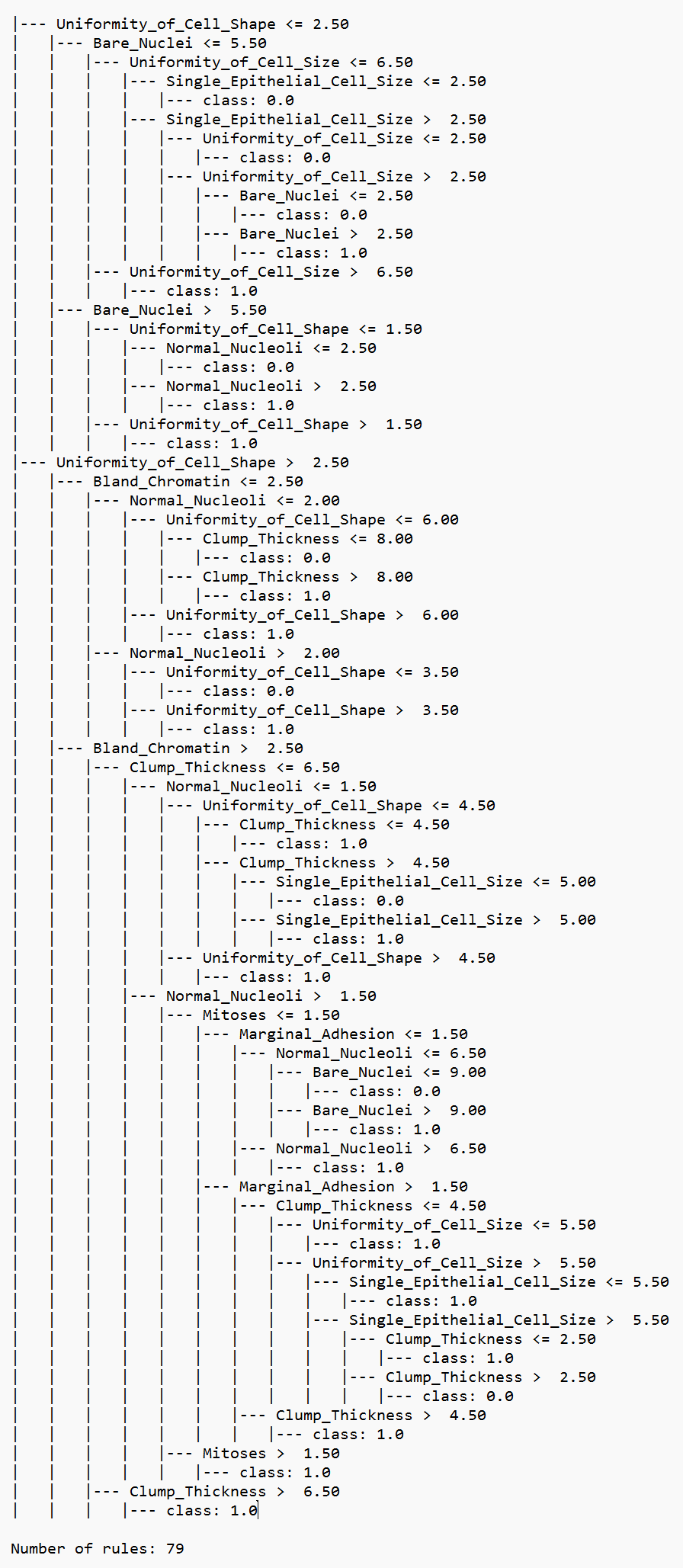


Figure 28 - Rules of The Best Decision Tree Generated By The Random Forest Algorithm

In Figure 29, Random Forest provides one of the best performances, achieving **133 true negatives** and **71 true positives**, with only **5 false positives** and **1 false negative**. This remarkable balance ensures high accuracy, precision, and recall across both classes, showcasing Random Forest's capability to handle diverse data distributions effectively.



Figure 29 - The Confusion Matrix of Random Forest Algorithm

## LightGBM

LightGBM (Light Gradient Boosting Machine) is another gradient boosting framework, and it is designed for speed and efficiency, especially with large datasets. The `LGBMClassifier()` I use has the parameters max\_depth=6 and min\_gain\_to\_split=0.1, which control the depth of the trees and the minimum required gain to make a split, respectively. By limiting the tree depth and setting a minimum gain for a split, LightGBM can prevent overfitting while ensuring that it captures important patterns in the data. It uses a technique called leaf-wise growth rather than level-wise growth, which often results in faster training and better performance. LightGBM is particularly useful when working with large-scale datasets, and it often outperforms other models in terms of both speed and accuracy.

Table 6 showcases the performance of the LightGBM model under different imputation techniques, considering whether class imbalance was checked or not. The evaluation metrics used are Accuracy, Recall, Precision, and F1 Score, providing a comprehensive view of the model's classification effectiveness.

1. With Checking Data Imbalance - Using the Overall Mean: When class imbalance is addressed and the overall mean is used for imputation, LightGBM achieves an accuracy of 94.29%, recall of 93.06%, precision of 90.54%, and an F1 score of 91.78%. This indicates that the model is moderately effective, especially in balancing recall and precision, although there is room for improvement in detecting the minority class.

2. Without Checking Data Imbalance - Using the Overall Mean: Without addressing class imbalance but still using overall mean imputation, LightGBM achieves an accuracy of 95.24%, recall of 93.06%, precision of 93.06%, and an F1 score of 93.06%. While the accuracy improves slightly compared to the previous scenario, the recall remains unchanged, suggesting that the model performs equally well in terms of identifying the minority class, even without imbalance handling.

3. With Checking Data Imbalance - Using Class-Wise Mean: When class imbalance is checked and class-wise mean imputation is applied, LightGBM shows a slight improvement, with an accuracy of 95.24%, recall of 94.44%, precision of 91.89%, and an F1 score of 93.15%. The recall increases, indicating that addressing imbalance with class-wise mean imputation enhances the model's ability to detect the minority class compared to the overall mean approach.

4. Without Checking Data Imbalance - Using Class-Wise Mean: Without checking for data imbalance but using class-wise mean imputation, LightGBM achieves an accuracy of 94.29%, recall of 93.06%, precision of 90.54%, and an F1 score of 91.78%. This performance mirrors the case where overall mean imputation was used without imbalance checking, highlighting that class-wise mean imputation does not substantially improve performance without addressing class imbalance.

5. With Checking Data Imbalance - Using KNN: With KNN imputation and imbalance correction, LightGBM maintains the same accuracy of 94.29%, recall of 93.06%, precision of 90.54%, and an F1 score of 91.78%. The performance is identical to the case with overall mean imputation and imbalance checking, suggesting that the KNN imputation method does not provide a significant advantage over the overall mean in this scenario.

6. Without Checking Data Imbalance - Using KNN: When KNN imputation is used without addressing class imbalance, LightGBM performs slightly better, achieving an accuracy of 94.76%, recall of 93.06%, precision of 91.78%, and an F1 score of 92.41%. This configuration shows a marginal improvement in precision and F1 score, indicating that KNN imputation without imbalance correction still results in relatively effective performance, particularly in precision.

In conclusion, LightGBM performs well across all scenarios, but its best results are achieved when class imbalance is addressed, especially when using class-wise mean imputation. The KNN imputation method does not seem to offer substantial improvements over the overall mean or class-wise mean imputation, but slight gains in precision and F1 score are evident when imbalance is ignored.

Table 6 - The Performance Metrics Of The LightGBM Using Different Imputation Techniques

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Imputation Techniques | Model | Accuracy | Recall | Precision | F1 Score |
| With Checking Data Imbalance - Using the Overall Mean | LightGBM | 0/94286 | 0/93056 | 0/90541 | 0/91781 |
| Without Checking Data Imbalance - Using the Overall Mean | LightGBM | **0/95238** | 0/93056 | 0/93056 | 0/93056 |
| With Checking Data Imbalance - Using Class-Wise Mean | LightGBM | **0/95238** | 0/94444 | 0/91892 | 0/93151 |
| Without Checking Data Imbalance - Using Class-Wise Mean | LightGBM | 0/94286 | 0/93056 | 0/90541 | 0/91781 |
| With Checking Data Imbalance - Using KNN | LightGBM | 0/94286 | 0/93056 | 0/90541 | 0/91781 |
| Without Checking Data Imbalance - Using KNN | LightGBM | 0/94762 | 0/93056 | 0/91781 | 0/92414 |

Figures 30 and 31 depict the model generated by the LightGBM algorithm for the dataset. This configuration achieved a commendable accuracy of 0.95238, leveraging the efficiency and scalability of the LightGBM framework. Unlike traditional decision trees, LightGBM is a gradient-boosting framework specifically optimized for speed and performance in large-scale datasets.

Instead of focusing on individual rules, LightGBM operates by building decision trees level-wise, focusing on features that yield the greatest information gain. This strategy allows the model to learn complex patterns effectively without explicitly defining rules for classification boundaries.

The strong performance of this LightGBM configuration highlights its ability to achieve competitive accuracy with minimal computational overhead, making it a highly efficient choice for handling diverse and complex datasets.

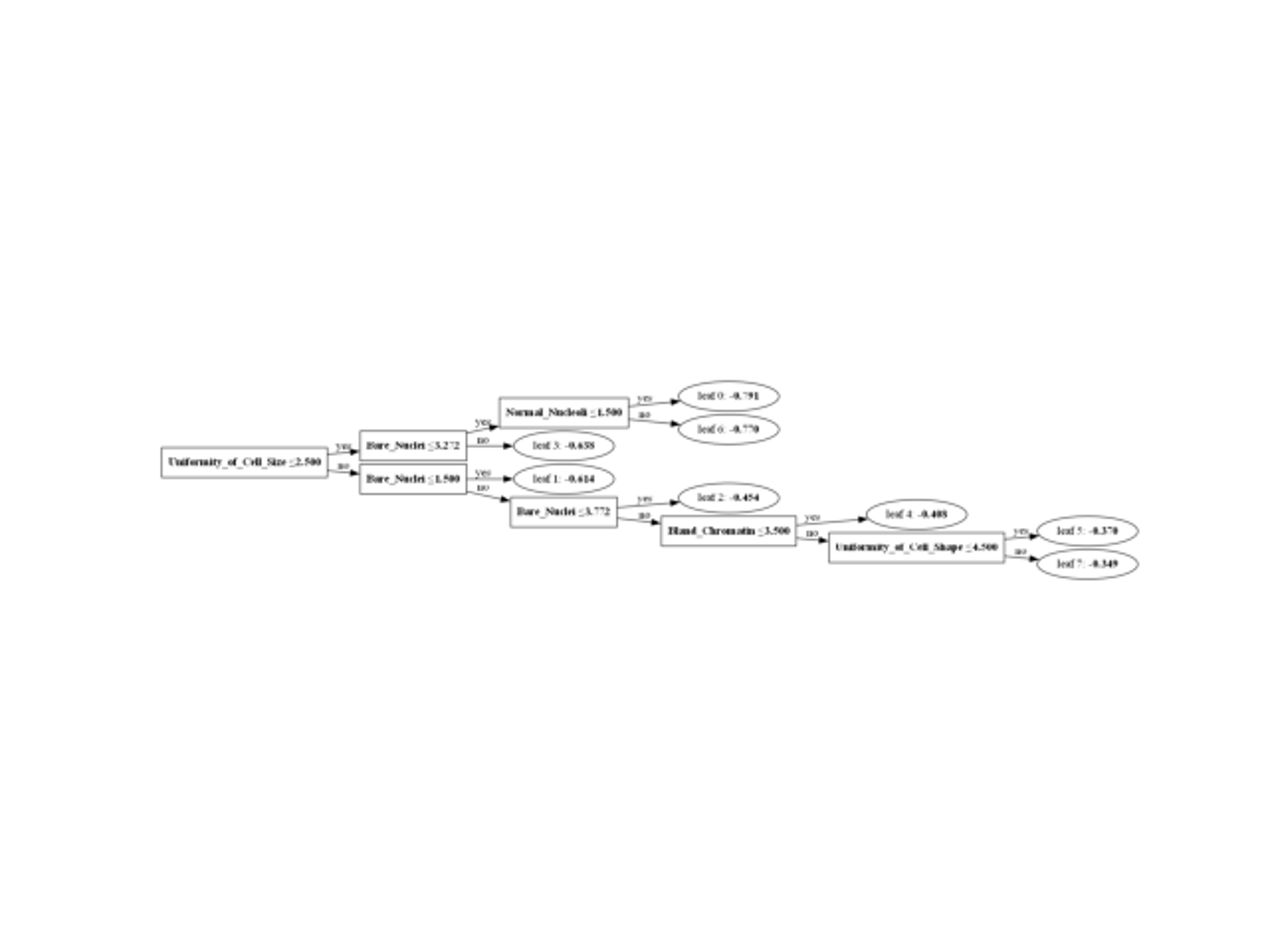


Figure 30 - One of The Best Decision Tree Generated By The LightGBM Algorithm

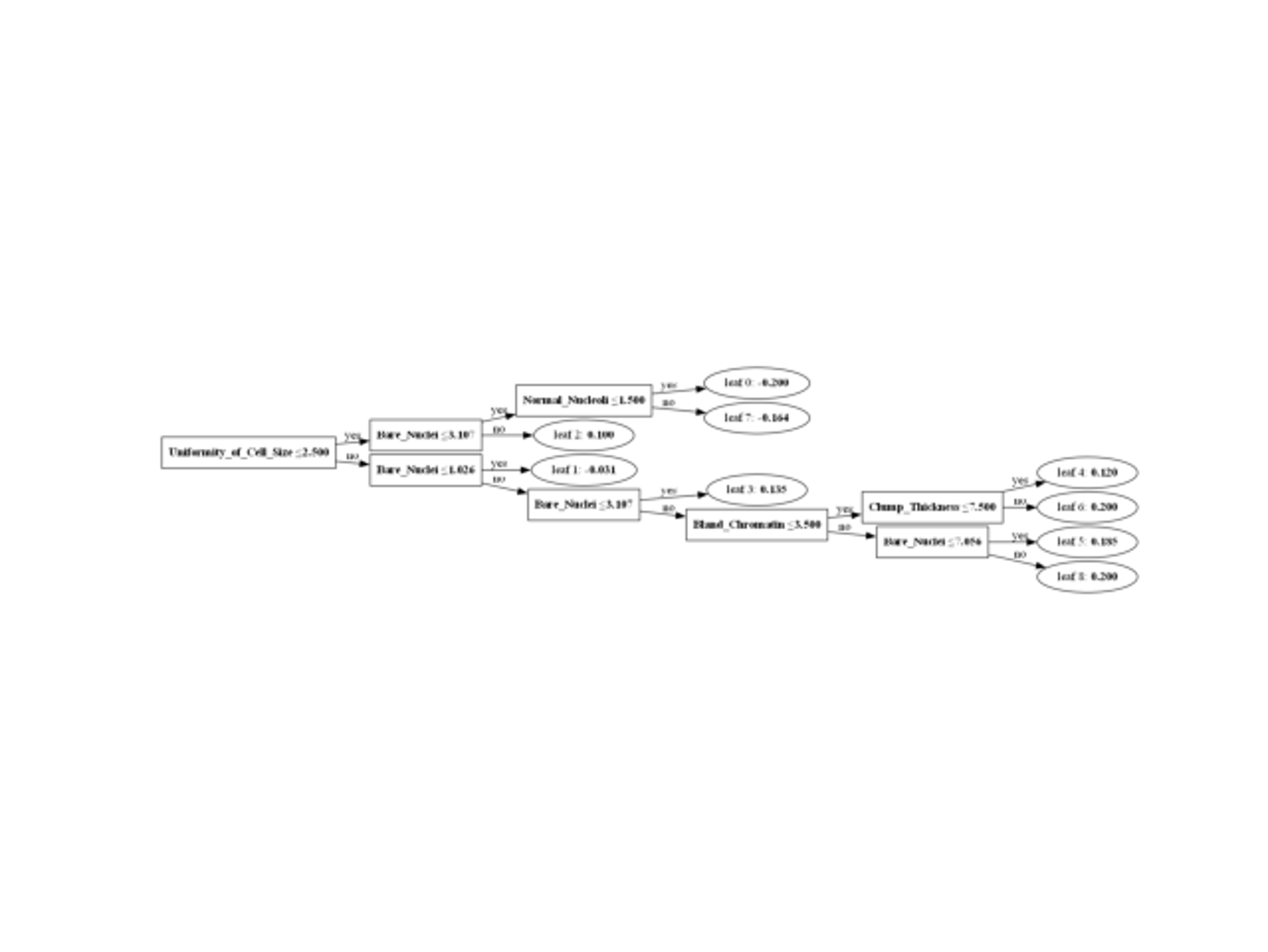


Figure 31 - One of The Best Decision Tree Generated By The LightGBM Algorithm

In Figure 32, LightGBM demonstrates strong predictive ability, with 133 true negatives and 67 true positives, but suffers from 5 false positives and 5 false negatives. While it effectively predicts Class 0, the increased false negatives reduce its recall for Class 1, indicating that the algorithm may need further tuning to handle minority classes better.

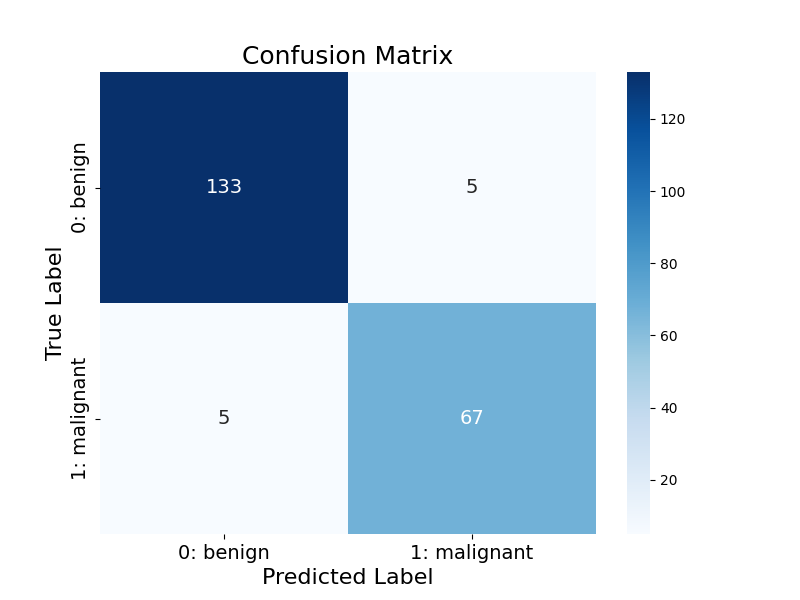


Figure 32 - The First Confusion Matrix of LightGBM Algorithm

In Figure 33, LightGBM strikes a balance similar to CART, with 132 true negatives, 68 true positives, 6 false positives, and 4 false negatives. Its overall performance highlights its consistency and ability to handle both classes reliably, though it does not outperform other models in precision or recall.

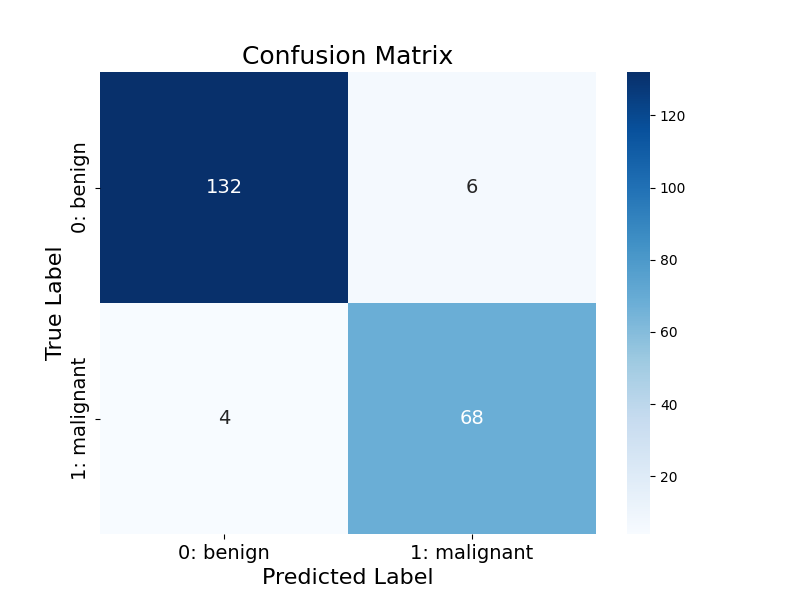


Figure 33 - The Second Confusion Matrix of LightGBM Algorithm

## ExtraTrees

ExtraTrees (Extremely Randomized Trees) is an ensemble learning method similar to Random Forest, but it introduces even more randomness during the training process. I use the `ExtraTreesClassifier()` to build a forest of trees by selecting random thresholds for each feature at each node. This approach makes the trees more diverse and, as a result, helps improve generalization. The randomness helps avoid overfitting, and the model benefits from averaging the results of multiple trees. I use ExtraTrees when I need an efficient model that can handle large datasets with high variance, as it generally has fast training times and is robust to noise.

Table 7 presents the performance of the ExtraTrees model across various imputation techniques and whether class imbalance was addressed or not. The metrics evaluated include Accuracy, Recall, Precision, and F1 Score, which offer a detailed analysis of the model’s classification ability.

1. With Checking Data Imbalance - Using the Overall Mean: When addressing class imbalance and using the overall mean for imputation, ExtraTrees achieves an accuracy of 95.71%, recall of 97.22%, precision of 90.91%, and an F1 score of 93.96%. This indicates that the model is highly effective in recalling both classes, particularly the minority class, while maintaining a good balance in precision.

2. Without Checking Data Imbalance - Using the Overall Mean: Without considering class imbalance but using overall mean imputation, ExtraTrees shows an accuracy of 95.71%, recall of 94.44%, precision of 93.15%, and an F1 score of 93.79%. Although the accuracy remains the same, the recall decreases, reflecting that class imbalance handling helps in more effectively identifying the minority class.

3. With Checking Data Imbalance - Using Class-Wise Mean: Using class-wise mean imputation while checking for class imbalance leads to an accuracy of 96.19%, recall of 97.22%, precision of 92.11%, and an F1 score of 94.59%. This scenario shows an improvement in precision and F1 score, suggesting that the model is better at detecting the minority class and balancing precision compared to the overall mean approach.

4. Without Checking Data Imbalance - Using Class-Wise Mean: Without checking for class imbalance but using class-wise mean imputation, ExtraTrees achieves an accuracy of 95.71%, recall of 95.83%, precision of 92%, and an F1 score of 93.88%. Here, while recall improves slightly over the overall mean without imbalance correction, the precision remains high, showing effective performance in terms of both false positives and false negatives.

5. With Checking Data Imbalance - Using KNN: With KNN imputation and class imbalance checking, ExtraTrees yields an accuracy of 96.67%, recall of 97.22%, precision of 93.33%, and an F1 score of 95.24%. This configuration offers the highest performance across all metrics, particularly boosting precision and F1 score, making it the most effective combination for ExtraTrees.

6. Without Checking Data Imbalance - Using KNN: When KNN imputation is used without addressing class imbalance, ExtraTrees achieves an accuracy of 96.19%, recall of 97.22%, precision of 92.11%, and an F1 score of 94.59%. This result is similar to the case where class-wise mean imputation is applied without imbalance handling, but the KNN imputation method maintains strong recall and precision, ensuring balanced classification.

In conclusion, ExtraTrees performs consistently well across all scenarios, with the best results observed when class imbalance is checked and KNN imputation is applied. The model shows a strong ability to identify the minority class (high recall) while maintaining a good balance between precision and recall, leading to high F1 scores. While the performance remains good without handling imbalance, addressing it provides noticeable improvements in both precision and recall.

Table 7 - The Performance Metrics Of The ExtraTrees Using Different Imputation Techniques

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Imputation Techniques | Model | Accuracy | Recall | Precision | F1 Score |
| With Checking Data Imbalance - Using the Overall Mean | ExtraTrees | 0/95714 | 0/97222 | 0/90909 | 0/93960 |
| Without Checking Data Imbalance - Using the Overall Mean | ExtraTrees | 0/95714 | 0/94444 | 0/93151 | 0/93793 |
| With Checking Data Imbalance - Using Class-Wise Mean | ExtraTrees | 0/96190 | 0/97222 | 0/92105 | 0/94595 |
| Without Checking Data Imbalance - Using Class-Wise Mean | ExtraTrees | 0/95714 | 0/95833 | 0/92000 | 0/93878 |
| With Checking Data Imbalance - Using KNN | ExtraTrees | **0/96667** | 0/97222 | 0/93333 | 0/95238 |
| Without Checking Data Imbalance - Using KNN | ExtraTrees | 0/96190 | 0/97222 | 0/92105 | 0/94595 |

Figures 34 and 35, illustrate the decision trees produced by the ExtraTrees algorithm for the dataset. This configuration achieved an impressive accuracy of 0.96667, reflecting its capability to balance complexity and performance effectively. ExtraTrees, or Extremely Randomized Trees, is an ensemble learning method that builds multiple trees by splitting nodes randomly and averaging their outputs.

For this setup, the model utilizes 130 rules, representing the distinct decision boundaries created across its ensemble of trees. Unlike traditional decision tree algorithms, ExtraTrees increases randomness in the tree-building process, enhancing its ability to generalize and reduce overfitting on complex datasets.

The high accuracy and number of rules in this ExtraTrees configuration highlight its ability to capture nuanced patterns in the data while maintaining robustness, making it a powerful choice for classification tasks.

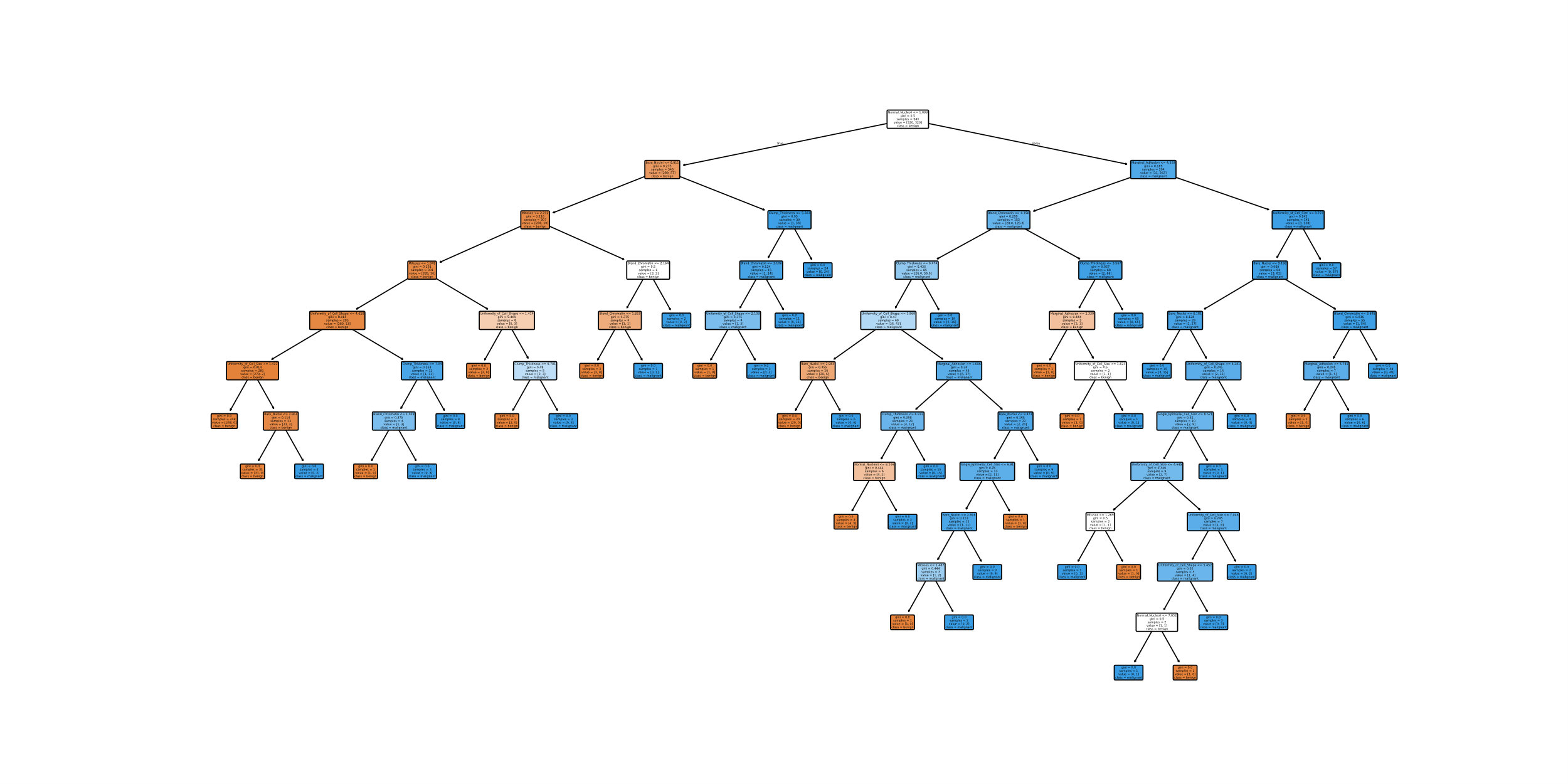


Figure 34 - The Best Decision Tree Generated By The ExtraTrees Algorithm

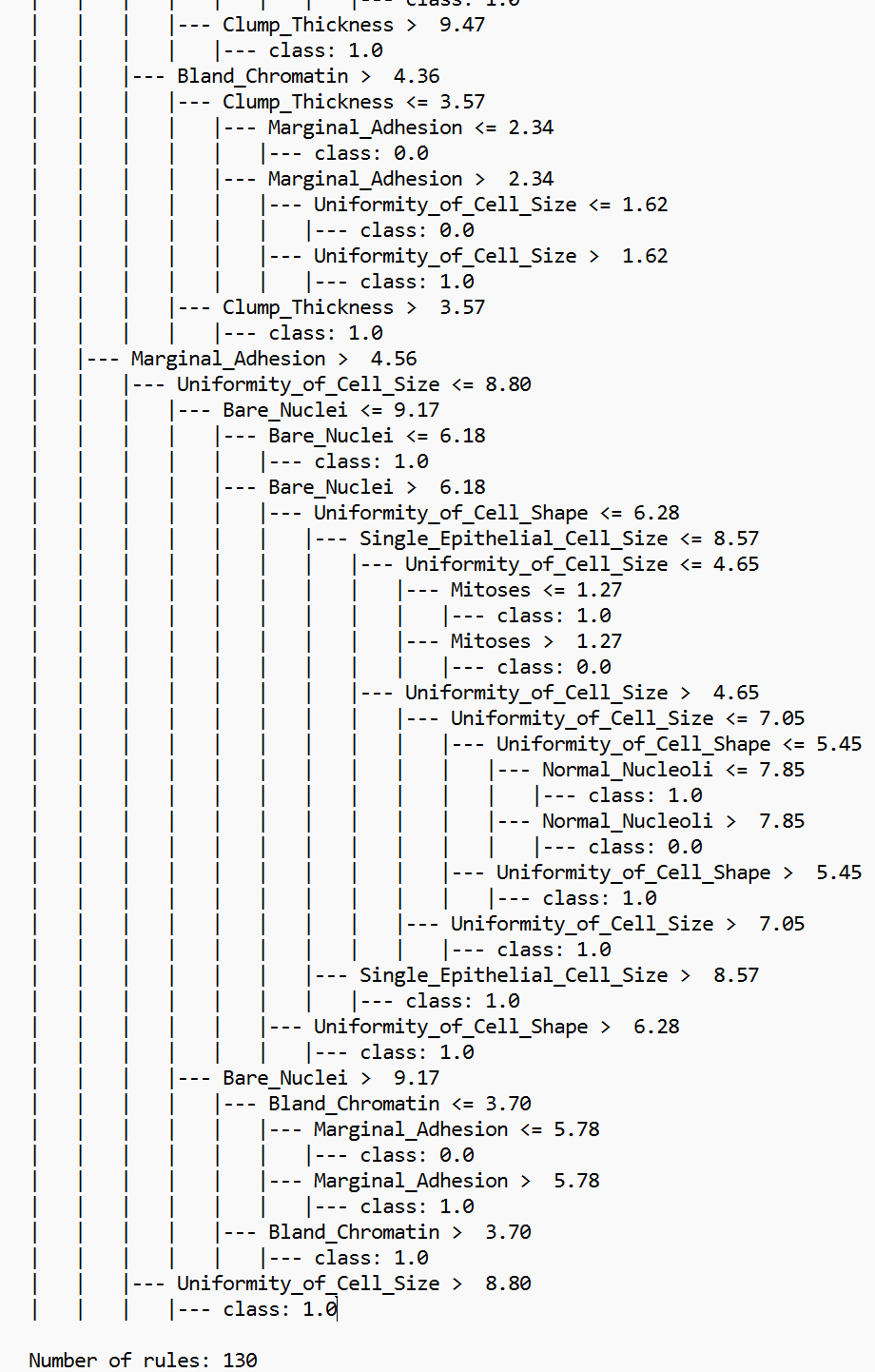
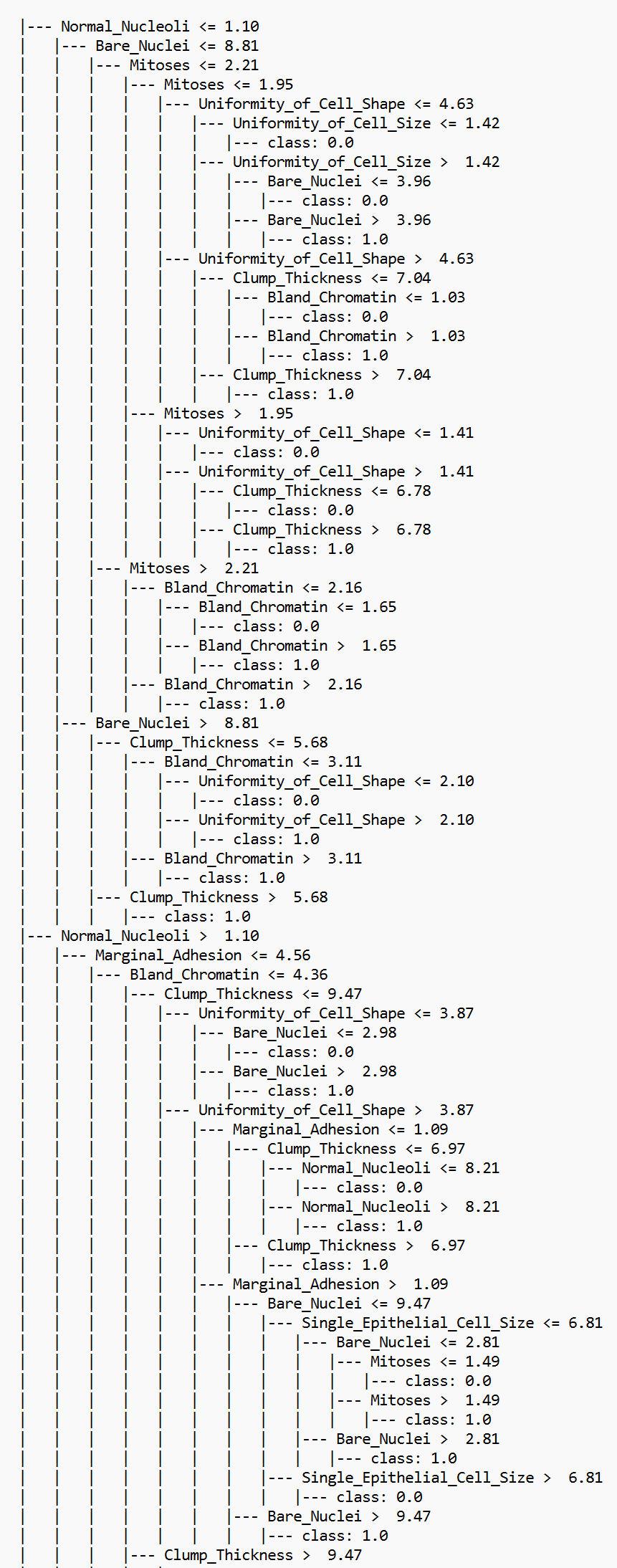


Figure 35 - Rules of The Best Decision Tree Generated By The ExtraTrees Algorithm

In Figure 36, ExtraTrees achieves excellent performance, with **133 true negatives** and **70 true positives**, alongside **5 false positives** and **2 false negatives**. This indicates the algorithm's ability to handle complex patterns in the data effectively, leading to high recall and precision for both classes.

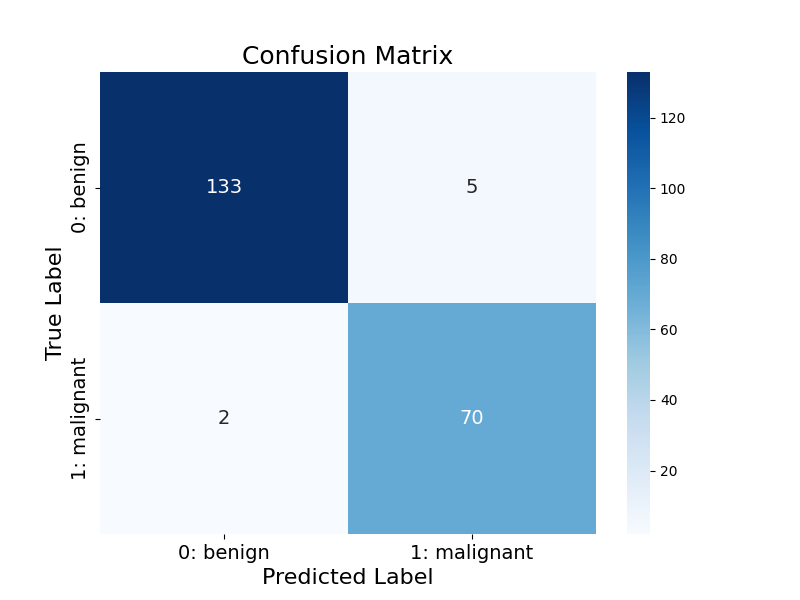


Figure 36 - The Confusion Matrix of ExtraTrees Algorithm

## Gradient Boosting

Gradient Boosting is another ensemble method I use with the `GradientBoostingClassifier()`, where each tree is built to correct the errors of the previous tree. Unlike AdaBoost, which adjusts the weights of misclassified instances, gradient boosting fits each new tree to the residual errors of the combined ensemble of trees. This technique minimizes the loss function step by step using gradient descent. Gradient Boosting tends to perform well even with relatively simple models, but it can be sensitive to overfitting if not properly tuned. I rely on this model when I need a strong, interpretable model that can improve accuracy without requiring too many computational resources.

Table 8 presents the performance of the Gradient Boosting model across various imputation techniques and class imbalance checking. The evaluated metrics include Accuracy, Recall, Precision, and F1 Score, providing a comprehensive view of how well the model performs under different conditions.

1. With Checking Data Imbalance - Using the Overall Mean: When addressing class imbalance and using the overall mean for imputation, Gradient Boosting achieves an accuracy of 94.29%, recall of 93.06%, precision of 90.54%, and an F1 score of 91.78%. This shows that while the model is reasonably effective in predicting both classes, there is room for improvement, particularly in precision.

2. Without Checking Data Imbalance - Using the Overall Mean: Without addressing class imbalance but using overall mean imputation, Gradient Boosting achieves an accuracy of 95.71%, recall of 94.44%, precision of 93.15%, and an F1 score of 93.79%. The accuracy improves slightly, and both recall and precision are better, indicating that class imbalance checking has a moderate effect on recall, while overall performance improves in this scenario.

3. With Checking Data Imbalance - Using Class-Wise Mean: With class-wise mean imputation and checking for class imbalance, Gradient Boosting shows an accuracy of 94.29%, recall of 93.06%, precision of 90.54%, and an F1 score of 91.78%. The results are similar to using the overall mean with imbalance checking, showing that the method of imputation does not significantly change performance when class imbalance is considered.

4. Without Checking Data Imbalance - Using Class-Wise Mean: When class-wise mean imputation is used without addressing class imbalance, Gradient Boosting achieves an accuracy of 95.24%, recall of 93.06%, precision of 93.06%, and an F1 score of 93.06%. This method provides a better balance in precision and recall than the overall mean method, with slightly improved precision.

5. With Checking Data Imbalance - Using KNN: With KNN imputation and class imbalance checking, Gradient Boosting shows an accuracy of 95.71%, recall of 95.83%, precision of 92%, and an F1 score of 93.88%. This combination improves recall significantly while maintaining precision, indicating a more balanced model performance when class imbalance is addressed.

6. Without Checking Data Imbalance - Using KNN: Without checking for class imbalance but using KNN imputation, Gradient Boosting achieves an accuracy of 96.19%, recall of 95.83%, precision of 93.24%, and an F1 score of 94.52%. This scenario provides the highest recall and precision among the configurations, with a notable improvement in the F1 score, showing that KNN imputation can enhance performance, especially when class imbalance is not considered.

In conclusion, Gradient Boosting performs well overall, with the best results observed when KNN imputation is used and class imbalance is checked. This combination leads to the highest accuracy, recall, precision, and F1 score. Without class imbalance checking, the model still performs well, with notable improvements in accuracy and precision when using KNN. Class-wise mean imputation results in performance similar to the overall mean when class imbalance is handled, but KNN provides better results in both scenarios.

Table 8 - The Performance Metrics Of The GradientBoosting Using Different Imputation Techniques

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Imputation Techniques | Model | Accuracy | Recall | Precision | F1 Score |
| With Checking Data Imbalance - Using the Overall Mean | GradientBoosting | 0/94286 | 0/93056 | 0/90541 | 0/91781 |
| Without Checking Data Imbalance - Using the Overall Mean | GradientBoosting | 0/95714 | 0/94444 | 0/93151 | 0/93793 |
| With Checking Data Imbalance - Using Class-Wise Mean | GradientBoosting | 0/94286 | 0/93056 | 0/90541 | 0/91781 |
| Without Checking Data Imbalance - Using Class-Wise Mean | GradientBoosting | 0/95238 | 0/93056 | 0/93056 | 0/93056 |
| With Checking Data Imbalance - Using KNN | GradientBoosting | 0/95714 | 0/95833 | 0/92000 | 0/93878 |
| Without Checking Data Imbalance - Using KNN | GradientBoosting | **0/96190** | 0/95833 | 0/93243 | 0/94521 |

Figure 37 illustrates the Gradient Boosting model applied to the dataset, achieving an excellent accuracy of **0.96190**. Gradient Boosting is a powerful ensemble learning technique that builds models sequentially, with each new model aiming to correct the errors made by the previous ones. This iterative process optimizes performance by minimizing a specified loss function.

Unlike traditional decision trees, Gradient Boosting does not rely on a single set of rules. Instead, it aggregates the outputs of multiple weak learners, weighting them based on their contribution to reducing classification errors. This approach makes it particularly effective in handling complex patterns in the data.

The outstanding accuracy of this GradientBoosting configuration underscores its strength in producing precise and reliable classifications, combining flexibility with robust generalization.



Figure 37 - The Best Decision Tree Generated By The GradientBoosting Algorithm

In Figure 38, GradientBoosting achieves robust classification results with **133 true negatives**, **69 true positives**, **5 false positives**, and **3 false negatives**. Its strong ability to reduce misclassifications across both classes makes it a reliable choice, showcasing its capability to optimize decision boundaries effectively for balanced datasets.

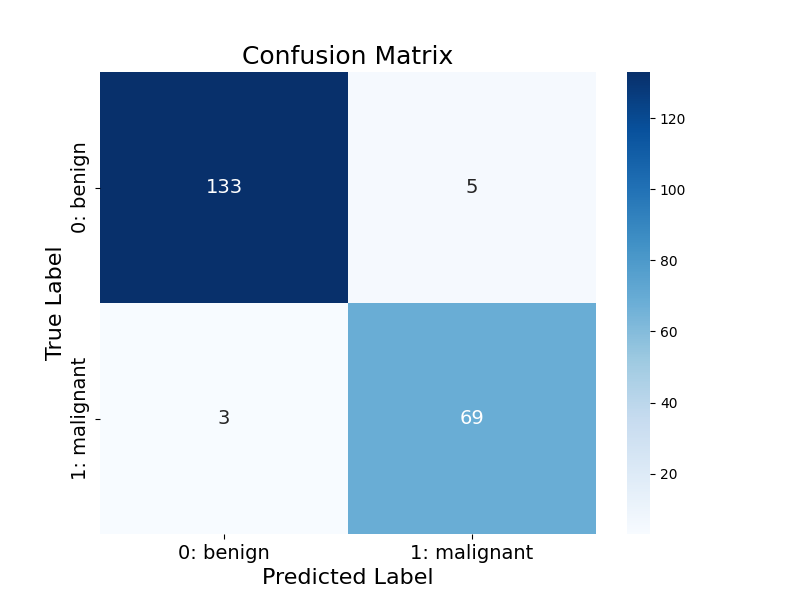


Figure 38 - The Confusion Matrix of GradientBoosting Algorithm