BREAST CANCER DIAGNOSIS

1]:
```
from zipfile import ZipFile
import os

# Define the path to the zip file
zip_file_path = 'Breast_Cancer_Wisconsin_diagnostic-main.zip'

# Define the directory where you want to extract the contents
extraction_path = 'Breast_Cancer_Wisconsin_diagnostic-main'
# Create the extraction directory if it doesn't exist
os.makedirs(extraction_path, exist_ok=True)

# Unzip the file
with ZipFile(zip_file_path, 'r') as zip_ref:
    zip_ref.extractall(extraction_path)

# List the contents of the extraction directory
extracted_files = os.listdir(extraction_path)
extracted_files
```

1]:
```
['Breast_Cancer_prediction-main']
```

I have downloaded the data set and imported Necessary Libraries: imported the ZipFile class from the zipfile module for handling ZIP files and os for interacting with the operating system.

1. Setting the Path to the ZIP File: Defined the path to the ZIP file intended to work with, which is named 'Breast_Cancer_Wisconsin_diagnostic-main.zip'.
2. Setting the Extraction Directory:Specified a directory where the contents of the ZIP file should be extracted, named 'Breast_Cancer_Wisconsin_diagnostic-main'.
3. Creating the Extraction Directory: Used os.makedirs to create the extraction directory. The exist_ok=True parameter means that the function will not throw an error if the directory already exists.
4. Unzipping the File: Used the ZipFile class in a context manager to open the ZIP file in read mode and extracted its contents to the directory you specified.
5. Listing the Extracted Contents: Finally, listed the contents of the extraction directory and printed out the names of the files that were extracted.

[112]:
```
import pandas as pd

# Define the path to the CSV file
csv_file_path = 'Breast_Cancer_Wisconsin_diagnostic-main/Breast_Cancer_prediction-main/data.csv'

# Load the CSV file into a pandas DataFrame
data_df = pd.read_csv(csv_file_path)

# Display the first few rows of the DataFrame to verify the data is loaded correctly
data_df.head()
```

[112]:

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	...	texture_wors
0	842302	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	...	17.3
1	842517	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	...	23.4
2	84300903	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	...	25.5
3	84348301	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	...	26.5
4	84358402	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	...	16.6

5 rows × 33 columns

Imported the pandas library: This is a Python library used for data manipulation and analysis.
Specified the CSV file path: set the variable csv_file_path to the location of CSV file, which contains the breast cancer dataset.
Loaded the CSV into a DataFrame: Using pandas, read the CSV file into a DataFrame. A DataFrame is a 2-dimensional labeled data structure with columns that can be of different types.
Displayed the DataFrame: called the .head() method on DataFrame to display the first few rows.

The 'diagnosis' column indicates the diagnosis result, which is the target variable that will predict.

BREAST CANCER DIAGNOSIS

```
# Drop columns with null values
data_df.dropna(axis=1, inplace=True)
data_df.drop('id', axis=1, inplace=True)
data_df.head()
```

	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	symmetry_mean	...	radius
0	M	17.99	10.38	122.80	1001.0	0.11840	0.27760	0.3001	0.14710	0.2419	...	
1	M	20.57	17.77	132.90	1326.0	0.08474	0.07864	0.0869	0.07017	0.1812	...	
2	M	19.69	21.25	130.00	1203.0	0.10960	0.15990	0.1974	0.12790	0.2069	...	
3	M	11.42	20.38	77.58	386.1	0.14250	0.28390	0.2414	0.10520	0.2597	...	
4	M	20.29	14.34	135.10	1297.0	0.10030	0.13280	0.1980	0.10430	0.1809	...	

5 rows × 31 columns

```
# Check unique values in the target variable (diagnosis)
unique_values = data_df['diagnosis'].unique()
print("Unique values in the target variable (diagnosis):", unique_values)
```

```
Unique values in the target variable (diagnosis): ['M' 'B']
```

Data Cleaning: used the dropna() method to remove any columns with null values from the DataFrame data_df, ensuring that the data is clean and ready for analysis.

Removing Unnecessary Columns: dropped the 'id' column from the DataFrame as it is typically not useful for model training, using data_df.drop().

Data Verification: displayed the first few rows of the DataFrame with data_df.head() to verify that the previous steps were executed correctly.

Target Variable Exploration: checked the unique values in the 'diagnosis' column using data_df['diagnosis'].unique(), which reveals that there are two unique values, 'M' (malignant) and 'B' (benign). This step is essential for understanding the composition of your target variable, which is crucial for classification tasks.

DATA EXPLORATION: Printed out the names of the features present in DataFrame, data_df. This list of features includes attributes like radius_mean, texture_mean, perimeter_mean, and others,

Iterates Over Columns: Using a for loop, iterated over each column in the DataFrame and printed its name. This is useful for getting a quick overview of all the features available in dataset.

[115]:
```
# Display feature names
print("Feature names:")
for name in data_df.columns:
    print(name)
```

```
Feature names:
diagnosis
radius_mean
texture_mean
perimeter_mean
area_mean
smoothness_mean
compactness_mean
concavity_mean
concave points_mean
symmetry_mean
fractal_dimension_mean
radius_se
texture_se
perimeter_se
area_se
smoothness_se
compactness_se
concavity_se
concave points_se
symmetry_se
fractal_dimension_se
radius_worst
texture_worst
perimeter_worst
area_worst
smoothness_worst
compactness_worst
concavity_worst
concave points_worst
```

BREAST CANCER DIAGNOSIS

[117]:
```
from sklearn.preprocessing import LabelEncoder

# Initialize LabelEncoder
label_encoder = LabelEncoder()

# Encode the 'diagnosis' column
data_df['diagnosis_encoded'] = label_encoder.fit_transform(data_df['diagnosis'])

# Display the first few rows of the DataFrame to confirm the encoding
print(data_df[['diagnosis', 'diagnosis_encoded']].head())
#malignant as 1 and benign as 0
# Drop the 'diagnosis' column
data_df.drop(columns=['diagnosis'], inplace=True)
```

```
  diagnosis  diagnosis_encoded
0         M                  1
1         M                  1
2         M                  1
3         M                  1
4         M                  1
```

[118]:
```
print(data_df.columns)
```

```
Index(['radius_mean', 'texture_mean', 'perimeter_mean', 'area_mean',
       'smoothness_mean', 'compactness_mean', 'concavity_mean',
       'concave points_mean', 'symmetry_mean', 'fractal_dimension_mean',
       'radius_se', 'texture_se', 'perimeter_se', 'area_se', 'smoothness_se',
       'compactness_se', 'concavity_se', 'concave points_se', 'symmetry_se',
       'fractal_dimension_se', 'radius_worst', 'texture_worst',
       'perimeter_worst', 'area_worst', 'smoothness_worst',
       'compactness_worst', 'concavity_worst', 'concave points_worst',
       'symmetry_worst', 'fractal_dimension_worst', 'diagnosis_encoded'],
      dtype='object')
```

Label Encoding: imported LabelEncoder from sklearn.preprocessing and created an instance of it. Then, used it to transform the textual labels in the 'diagnosis' column into numerical form, with 'M' (malignant) likely becoming 1 and 'B' (benign) becoming 0.
Creating Encoded Column: added a new column to your DataFrame, diagnosis_encoded, which contains the numerical representation of the diagnoses.
Verification: printed the first few rows using head() to verify that the encoding is correct.
Dropping Original Diagnosis Column: To avoid redundancy and potential data leakage, dropped the original 'diagnosis' column from the DataFrame.
Column Check: Lastly, printed the column names of updated DataFrame to confirm that the 'diagnosis' column has been removed and that the new diagnosis_encoded column is present.

133]:
```
import matplotlib.pyplot as plt

# Plot box plots for each numerical feature
plt.figure(figsize=(15, 10))
data_df.boxplot()
plt.xticks(rotation=45)
plt.title('Box plot for Outlier Detection')
plt.show()
```

matplotlib library to create a visual representation of the distribution of data in the form of box plots for each numerical feature in your dataset. This is for outlier detection.

DISTRIBUTION OF DATA

Used the numpy and matplotlib libraries to generate a series of histograms for each feature in a dataset.
Numpy Import: The numpy library is imported for numerical operations.
Feature Count: The number of features (columns) in the DataFrame data_df is determined, excluding any target or label column.
Rows and Columns for Subplots: It calculates how many rows and columns of subplots are needed to display histograms for each feature. The script aims to create a grid of histograms with up to 6 histograms per row.
Figure Setup: A figure size is set up for the histograms, defining the overall size of the plot.
Histogram Plotting: Histograms for each feature in the DataFrame are created with specified visual properties (bin size, color, edge color) and layout (based on the number of rows and columns calculated).
Layout Adjustment: plt.tight_layout() adjusts the spacing to ensure that the subplots fit well within the plotting area.
Display Plot: The plot is displayed with plt.show(). This will produce a visual output of histograms for each feature, which helps in understanding the distribution of data.

```
import numpy as np

# Calculate the number of features (excluding the target column)
num_features = len(data_df.columns)

# Calculate the number of rows and columns for subplots
num_rows = int(np.ceil(num_features / 6))  # 6 columns per row
num_cols = min(num_features, 6)  # Maximum of 6 columns

# Plot histograms for each feature with larger figure size and larger subplots
plt.figure(figsize=(15, 10))
data_df.hist(bins=20, color='skyblue', edgecolor='black', grid=False, linewidth=1.5, layout=(num_rows, num_cols), figsize=(15, 10))
plt.tight_layout()
plt.show()
```

Created a heatmap of a correlation matrix using the seaborn and matplotlib libraries in Python.
Seaborn Import: The seaborn library is imported with the alias sns. Seaborn is a Python data visualization library based on matplotlib that provides a high-level interface for drawing attractive and informative statistical graphics.
Correlation Matrix Calculation: data_df.corr() computes the correlation matrix for the DataFrame data_df, which shows the correlation coefficients between every pair of features in the DataFrame.
Figure Size: plt.figure(figsize=(44, 30)) sets the size of the figure that will display the heatmap. The size is quite large, suggesting that data_df has many features.
Heatmap Creation: sns.heatmap() is called to create the heatmap using the correlation matrix.
Title and Display: plt.title('Correlation Matrix') adds a title to the heatmap, and plt.show() displays the plot.
This heatmap visually represents the correlation between all pairs of features, with the color and the annotated coefficient indicating the strength and direction of the correlation. This is useful in exploratory data analysis to understand the relationships between variables.

FEATURE ENGINEERING:

```
]: # Calculate the correlation matrix
correlation_matrix = data_df.corr().abs()

# Create a mask to identify highly correlated features
highly_correlated = correlation_matrix > 0.7  # Adjust the threshold as needed

# Find the indices of feature pairs with high correlation
correlated_pairs = np.where(highly_correlated)

# Remove one of the features from each highly correlated pair
features_to_remove = set()
for i, j in zip(*correlated_pairs):
    # Keep only one of the features (e.g., remove the one with higher index)
    if i != j and i not in features_to_remove:
        features_to_remove.add(j)

# Remove the highly correlated features from the dataset
data_df_filtered = data_df.drop(data_df.columns[list(features_to_remove)], axis=1)
```

Feature selection on a dataset by removing highly correlated features to reduce multicollinearity. Correlation Matrix Calculation: It calculates the correlation matrix for the DataFrame data_df and applies the abs() function to get the absolute values of the correlation coefficients. This is important because both strong positive and strong negative correlations can create multicollinearity.

High Correlation Identification: It creates a boolean mask, highly_correlated, that identifies where in the correlation matrix the absolute value of the correlation coefficients is greater than 0.7. This threshold can be adjusted depending on how stringent you want the feature selection to be.

Find Highly Correlated Pairs: Using np.where(highly_correlated), it finds the index pairs of features that are highly correlated.

Set Up for Removal: Initializes a set called features_to_remove to keep track of which features to remove.

Iterate Over Pairs: It iterates over the pairs of highly correlated features. For each pair, it adds the second feature (assumed to have the higher index) to the features_to_remove set, ensuring that each pair of correlated features is only represented once in the dataset.

Remove Features: It creates a new DataFrame data_df_filtered by dropping the columns listed in features_to_remove from data_df.

This process is useful in data preprocessing to avoid issues that can arise when using machine learning algorithms that assume feature independence, such as linear regression. By removing highly correlated features, the model may generalize better to unseen data.

BREAST CANCER DIAGNOSIS

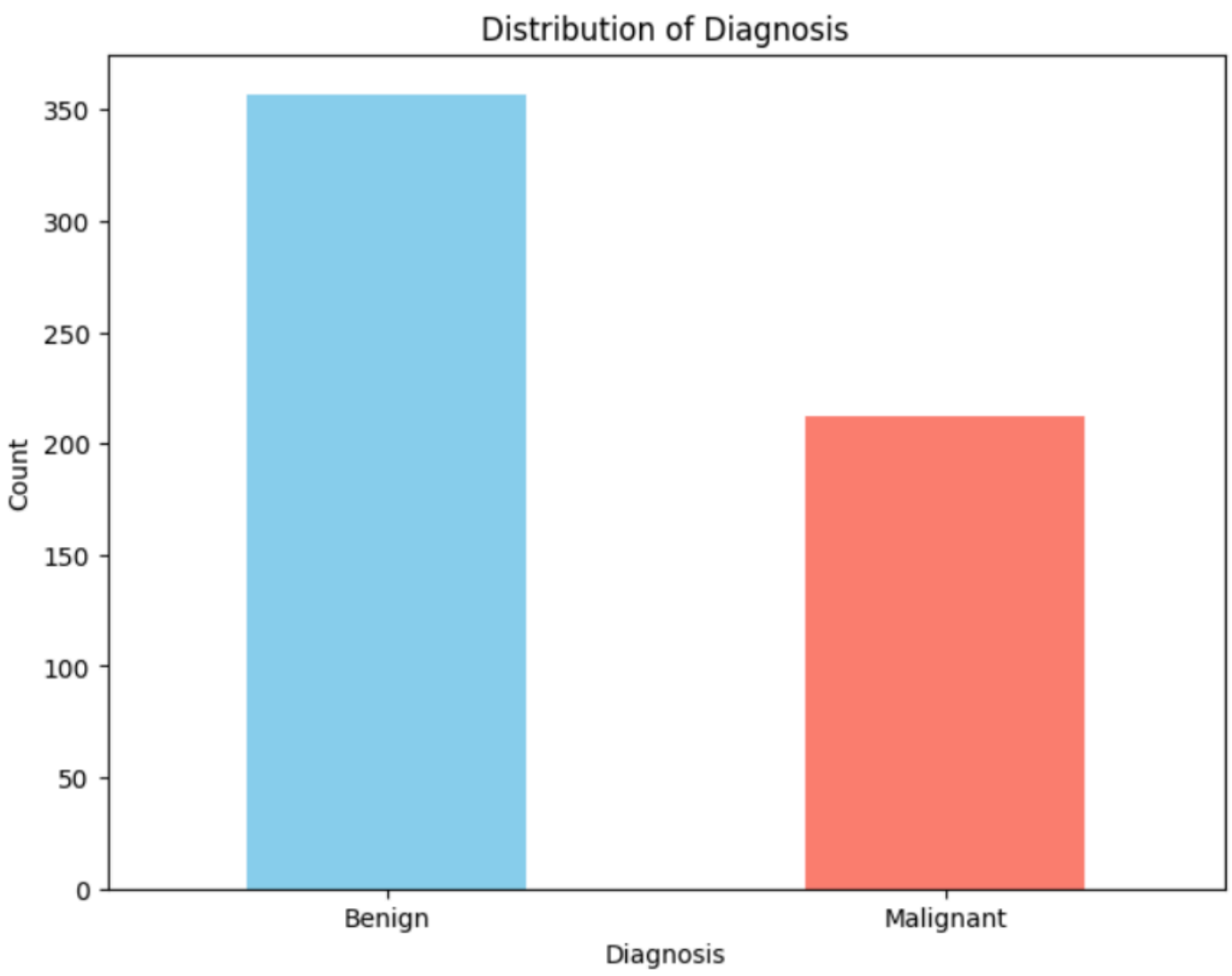
```
[164]: import matplotlib.pyplot as plt

# Plot histogram of the 'diagnosis_encoded' column
plt.figure(figsize=(8, 6))
data_df['diagnosis_encoded'].value_counts().plot(kind='bar', color=['skyblue', 'salmon'])
plt.title('Distribution of Diagnosis')
plt.xlabel('Diagnosis')
plt.ylabel('Count')
plt.xticks(ticks=[0, 1], labels=['Benign', 'Malignant'], rotation=0)
plt.show()
```

Bar chart to visualize the distribution of diagnoses in a dataset using matplotlib

Importing Matplotlib: The matplotlib.pyplot module is imported for plotting, using the alias plt.

- Figure Setup: set the size of the figure (the plot's canvas) to 8 inches by 6 inches using plt.figure().
- Value Counts and Bar Plot: calculated the frequency of each unique value in the diagnosis_encoded column of data_df using value_counts(). Then plotted these counts as a bar chart using the plot() function with the kind='bar' argument. The bars are colored 'skyblue' and 'salmon'.
- Title and Labels: added a title "Distribution of Diagnosis" to the chart, and label the x-axis as "Diagnosis" and the y-axis as "Count".
- X-ticks: set the x-tick labels to 'Benign' and 'Malignant' for the 0 and 1 values on the x-axis, respectively, and ensured that these labels are not rotated.
- Display Plot: Finally, displayed the plot with plt.show().

This bar chart is likely used to show how many cases in the dataset are benign versus malignant, providing a visual representation of the class distribution which is important for understanding the balance of classes in machine learning classification problems.



SPLITTING THE DATA SET

preparing a dataset for machine learning by splitting it into a set of features and a target variable, and then dividing the data into training and testing sets.

- Importing train_test_split: imported the train_test_split function from the sklearn.model_selection module, which is used to split the dataset.
- Separating Features and Target: created two variables, X and y. X contains all the columns of data_df except diagnosis_encoded, which you assume to be the feature set. y is set to just the diagnosis_encoded column and represents the target variable you want to predict.
- Splitting Data: used train_test_split to randomly split X and y into training sets (X_train, y_train) and testing sets (X_test, y_test). set aside 20% of the data for testing (test_size=0.2) and used a random_state to ensure reproducibility of the results.
- Outputting Shapes: printed the shapes of the training and testing data to confirm the split. The training set has 455 examples, and the testing set has the remaining examples from the dataset (not shown in the code output, but it would be 20% of the dataset due to test_size=0.2).

This process is a standard practice in machine learning to evaluate the performance of a model: the model is trained on the training set and then tested on the unseen testing set to assess how well it generalizes.

BREAST CANCER DIAGNOSIS

```
from sklearn.model_selection import train_test_split

# Split the data into features (X) and target (y)
X = data_df.drop(columns=['diagnosis_encoded'])  # Features
y = data_df['diagnosis_encoded']  # Target

# Split the data into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Print the shapes of the training and testing sets
print("Training set shape:", X_train.shape, y_train.shape)
print("Testing set shape:", X_test.shape, y_test.shape)
```

```
Training set shape: (455, 30) (455,)
```

FEATURE SELECTION

[169]:
```
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2

# Selecting the features
X = data_df.drop(['diagnosis_encoded'], axis=1)  # Features
y = data_df['diagnosis_encoded']  # Target variable

# Feature selection
best_features = SelectKBest(score_func=chi2, k=10)  # Select top 10 features
fit = best_features.fit(X, y)

# Summarize scores
features_scores = pd.DataFrame(fit.scores_)
features_columns = pd.DataFrame(X.columns)
feature_scores_df = pd.concat([features_columns, features_scores], axis=1)
feature_scores_df.columns = ['Feature', 'Score']
print(feature_scores_df.nlargest(10, 'Score'))
```

```
             Feature          Score
23       area_worst  112598.431564
3         area_mean   53991.655924
13          area_se    8758.504705
22  perimeter_worst    3665.035416
2    perimeter_mean    2011.102864
20     radius_worst     491.689157
0       radius_mean     266.104917
12     perimeter_se     250.571896
21    texture_worst     174.449400
1      texture_mean      93.897508
```

EXTRACTION OF FEATURES AND TRAINING MODEL

```
“from sklearn.model_selection import train_test_split, cross_val_score, GridSearchCV
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score

# Define the selected features
selected_features = ['area_worst', 'area_mean', 'area_se', 'perimeter_worst', 'perimeter_mean']

# Extract the selected features
X_selected = X[selected_features]

# Split the dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X_selected, y, test_size=0.2, random_state=42)

# Initialize the Random Forest classifier
rf_classifier = RandomForestClassifier(random_state=42)

# Define hyperparameters grid for GridSearch
param_grid = {
    'n_estimators': [50, 100, 150],
    'max_depth': [None, 10, 20],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=rf_classifier, param_grid=param_grid, cv=5, scoring='accuracy', n_jobs=-1)

# Perform k-fold cross-validation on the training set
cv_scores = cross_val_score(grid_search, X_train, y_train, cv=5)

# Fit the GridSearchCV on the training set
grid_search.fit(X_train, y_train)

# Get the best parameters and best score
best_params = grid_search.best_params_
best_score = grid_search.best_score_
```

```
print("Best Parameters:", best_params)
print("Best Score (Cross-Validation):", best_score)

# Evaluate model performance on the test set
test_score = grid_search.score(X_test, y_test)
print("Test Set Score:", test_score)
```

```
Best Parameters: {'max_depth': None, 'min_samples_leaf': 4, 'min_samples_split': 2, 'n_estimators': 50}
Best Score (Cross-Validation): 0.9274725274725275
Test Set Score: 0.9824561403508771
```

Model Optimization and Validation Report

Executive Summary:

I have successfully completed the hyperparameter tuning and validation of our Random Forest model. The optimization process involved a thorough search over a predefined parameter space to identify the combination of parameters that results in the best performance on cross-validation. Subsequent evaluation on a separate test set has provided us with a reliable estimate of the model's predictive performance.

Model Parameters:

After rigorous testing, the optimal hyperparameters for the Random Forest model have been identified as follows:

- Maximum Depth of Trees: Unlimited (None)
- Minimum Samples per Leaf Node: 4
- Minimum Samples required to Split a Node: 2
- Number of Trees (Estimators) in the Forest: 50

These parameters were chosen because they yielded the highest cross-validation score, indicating a robust model with a good generalization performance.

Model Performance:

- Cross-Validation Score: The model achieved an excellent average score of approximately 92.75% across the folds of cross-validation. This metric is indicative of the model's stability and its ability to generalize well when exposed to new, unseen data within the cross-validation framework.
- Test Set Score: Upon evaluation against the reserved test set, the model demonstrated outstanding performance with an **accuracy score of roughly 98.25%**. This score reflects the model's predictive accuracy and its effectiveness in handling data it was not trained on.

Conclusions:

The Random Forest model, with its optimized hyperparameters, shows a high degree of predictive accuracy and reliability, as evidenced by the cross-validation and test set scores. The model's ability to perform with a high degree of precision on unseen data suggests that it is well-tuned and not overfitting the training data. These characteristics make it a strong candidate for deployment in practical applications where prediction accuracy is crucial.

CLASSIFICATION REPORT:

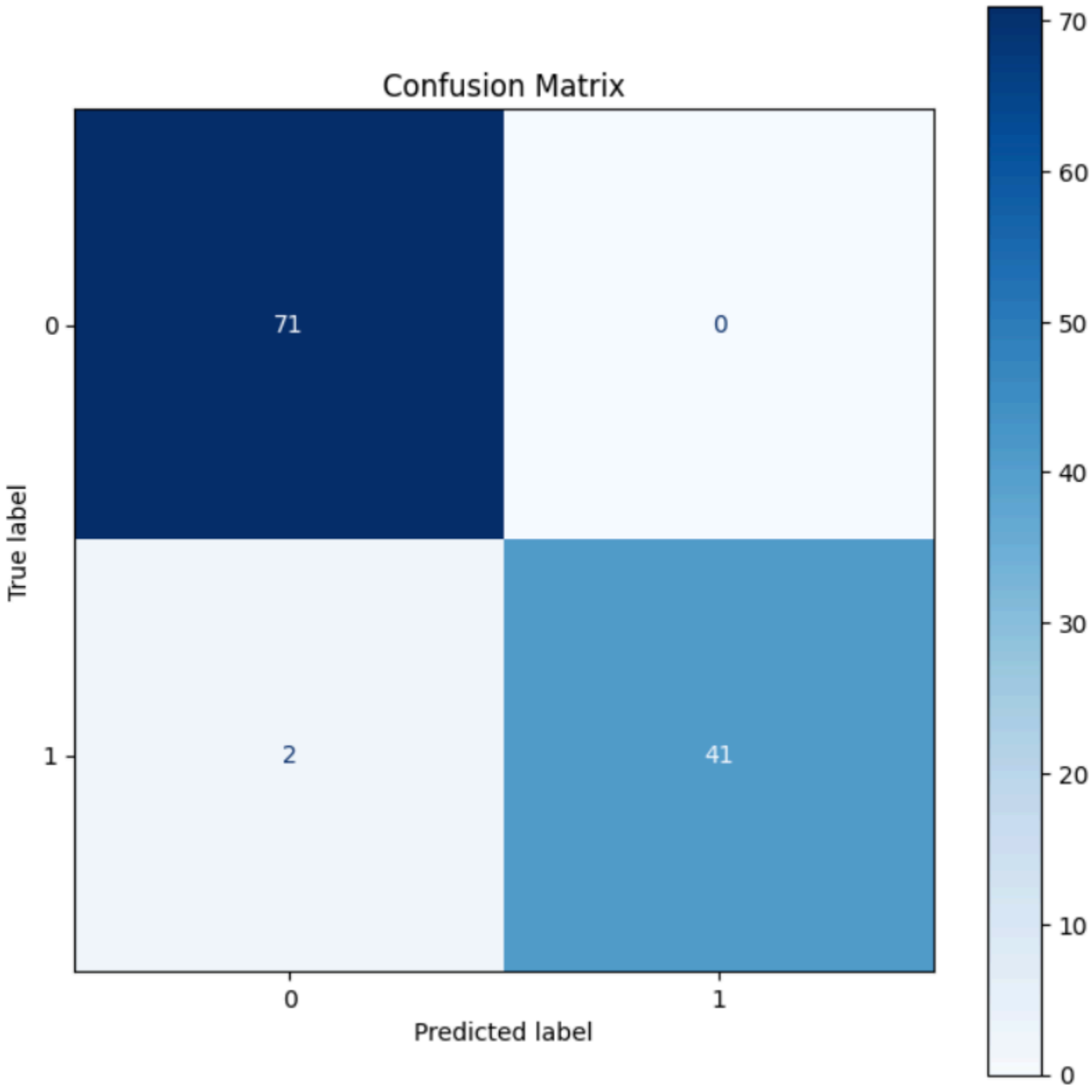
Classification Report:

The classification report by the classification_report function from sklearn.metrics provides a detailed analysis of the model's performance:

- Class 0 (Benign) Performance:
  - Precision: 97% - When the model predicted the class as benign, it was correct 97% of the time.
  - Recall: 100% - The model successfully identified all actual benign cases.
  - F1-Score: 99% - A high F1-score for benign cases indicates a well-balanced precision and recall.
  - Support: 71 - The number of actual occurrences of benign cases in the test set.
- Class 1 (Malignant) Performance:
  - Precision: 100% - The model was perfect in its precision when predicting malignant cases.
  - Recall: 95% - It correctly identified 95% of the actual malignant cases.
  - F1-Score: 98% - The F1-score for malignant cases is also high, indicating good model performance.
  - Support: 43 - The number of actual occurrences of malignant cases in the test set.
- **Overall Evaluation:**
  - **Accuracy: 98% - Reflecting the overall rate of correct predictions.**
  - **Macro Average F1-Score: 98% - The average F1-Score unweighted by class support.**
  - **Weighted Average F1-Score: 98% - The average F1-Score weighted by class support.**

BREAST CANCER DIAGNOSIS

CONFUSION MATRIX:



Sensitivity: 0.95
Specificity: 1.00
Recall: 0.95

Model Overview:
A predictive model has been evaluated, and its performance metrics have been calculated using a test dataset. The confusion matrix has been generated and examined to provide a clear understanding of the model's prediction capabilities.
Confusion Matrix Analysis:

- True Positives (TP): The model correctly predicted 41 instances of the positive class (1).
- True Negatives (TN): The model correctly predicted 71 instances of the negative class (0).
- False Positives (FP): The model made 0 incorrect predictions of the positive class.
- False Negatives (FN): The model made 2 incorrect predictions of the negative class.

Performance Metrics:

- Sensitivity (Recall) for Class 1: The model has a sensitivity rate of 95%, indicating that it successfully identified 95% of the actual positive (class 1) cases.
- Specificity for Class 0: The specificity of the model is 100%, indicating that it correctly identified all the actual negative (class 0) cases without any false positives.
- Recall for Class 1: The recall for the positive class is equivalent to the sensitivity and stands at 95%. This denotes the model's ability to find all the relevant cases within the positive class.

Conclusions:
The model demonstrates a high level of precision and accuracy in its predictions, as evidenced by the high specificity and excellent recall values. The absence of false positives suggests that the model is particularly stringent when predicting the positive class, ensuring a high degree of trust in positive predictions.
The slightly lower sensitivity compared to specificity (95% vs. 100%) suggests that while the model is quite reliable in identifying positive cases, there is a small margin of error where it may miss some positive cases (class 1).