Cairo University

Faculty of Computers and Artificial Intelligence

Machine Learning

Assignment 2

Section IS S1&S2

Team Members

Name	ID
Salma Mamdoh Sabry	**20210162**
Roaa Talat Mohamed	**20210138**
Youssef Ehab Mohamed	**20210466**
Zeyad Ehab Maamoun	**20211043**
Youssef Mohamed Salah Eldin Anwar	**20210483**

About the Dataset

This dataset captures details on how **weather-related features** such as temperature, humidity, wind speed, cloud cover, and pressure relate to the likelihood of **rain**. The dataset contains information on weather conditions and is compiled across a period, providing insights into how various weather attributes influence precipitation. The dataset includes **2,500 rows** and **6 columns**.

Key Information

- **Weather Features**:
    - **Temperature**: The ambient temperature in degrees Celsius.
    - **Humidity**: The percentage of moisture in the air.
    - **Wind Speed**: The speed of the wind in meters per second.
    - **Cloud Cover**: The percentage of sky covered by clouds.
    - **Pressure**: The atmospheric pressure in hectopascals (hPa).
- **Target Variable**:
    - **Rain**: Indicates whether it rained or not (binary classification: "rain" or "no rain").

Column Descriptions

- **Temperature**: Ambient temperature in degrees Celsius.
- **Humidity**: The percentage of moisture present in the air.
- **Wind Speed**: The speed of wind measured in meters per second.
- **Cloud Cover**: The percentage of sky covered by clouds.
- **Pressure**: The atmospheric pressure recorded in hectopascals.
- **Rain**: The target variable, indicating whether it rained (1) or did not rain (0) based on the weather conditions.

This dataset can be used to predict the likelihood of rain based on various weather parameters like temperature, humidity, and wind speed, which can be valuable for weather forecasting and climate studies.

Task 1: Preprocessing

1. Does the dataset contain any missing data? Identify them.

Data Cleaning

In [197]:
```
def Missing_Data_Check(df):
    print("\nMissing Data Check:")
    missing_data = df.isnull().sum()
    print(missing_data)
```

In [198]:
```
Missing_Data_Check(df)
```

```
Missing Data Check:
Temperature    25
Humidity       40
Wind_Speed     32
Cloud_Cover    33
Pressure       27
Rain            0
dtype: int64
```

Data Have missing Values lets identify them

In [199]:
```
# display rows with missing data
print("\nRows with Missing Data:")
print(df[df.isnull().any(axis=1)])
```

```
Rows with Missing Data:
      Temperature   Humidity  Wind_Speed  Cloud_Cover     Pressure     Rain
8             NaN  89.077804    4.842197    83.941093  1029.932706  no rain
25      26.420959  72.283460         NaN     0.812305  1018.818494  no rain
59      11.069078  89.683583    5.804538          NaN   992.303157  no rain
68            NaN  58.981077    6.261278    37.580222  1019.684713  no rain
74      33.078976  81.000650    5.744880    86.933978          NaN  no rain
...           ...        ...         ...          ...          ...      ...
2429          NaN  93.920582   13.302477    90.346087   998.183246     rain
2436    16.838551  86.248171   13.326615          NaN  1004.497445     rain
2445    14.279301        NaN   19.789469    95.934640  1031.653350     rain
2446    13.695217  95.727543         NaN    65.020145   983.800057     rain
2483    17.449257  70.094641         NaN    64.609907  1041.623220     rain

[153 rows x 6 columns]
```

Missing Data Analysis

1. **Summary of Missing Data:**
    - The dataset contains missing values across several columns:
        - **Temperature:** 25 missing entries
        - **Humidity:** 40 missing entries
        - **Wind_Speed:** 32 missing entries
        - **Cloud_Cover:** 33 missing entries
        - **Pressure:** 27 missing entries
        - **Rain:** No missing entries

This indicates that almost all key weather variables have some degree of missing data, which may affect subsequent analysis if not addressed.

2. **Rows with Missing Data:**

- A total of **153 rows** contain at least one missing value, as identified from the dataset.
- These rows span various columns, with missing values distributed across different observations. For example:
  - **Row 8:** Missing Temperature
  - **Row 25:** Missing Wind_Speed
  - **Row 68:** Missing Temperature and Cloud_Cover

This highlights the need for a strategy to handle missing values, such as imputation or removal, depending on the analysis requirements.

2. **Apply the two techniques to handle missing data, dropping missing values and replacing them with the average of the feature.**

Apply the two techniques to handle missing data, dropping missing values and replacing them with the average of the feature.

```
In [203]: def handle_missing_data(df, method='replace'):
    df_copy = df.copy()

    if method == 'replace':
        df_copy.fillna(df_copy.select_dtypes(include=['float64']).mean(), inplace=True)
        print("Missing values have been replaced with the mean of each feature.")
        return df_copy
    elif method == 'drop':
        df_copy.dropna(inplace=True)
        print(f"Rows with missing values have been dropped. Remaining rows: {len(df_copy)}.")
        return df_copy
    else:
        print("Invalid method! Please use 'replace' or 'drop'.")
        return df_copy
```

```
In [204]: df_cleaned_using_Replace = handle_missing_data(df, method='replace')
```

```
Missing values have been replaced with the mean of each feature.
```

```
In [205]: df_cleaned_using_drop = handle_missing_data(df, method='drop') # original data 2500 row
```

```
Rows with missing values have been dropped. Remaining rows: 2347.
```

Handling Missing Data

1. **Techniques Applied:**
   - Two approaches were used to address missing data:
     - **Replacing Missing Values with Mean:** Missing values in numerical columns were replaced with the mean of the respective column.
     - **Dropping Rows with Missing Values:** Rows containing any missing values were removed from the dataset.

2. **Results:**

   - **Replacing Missing Values:**
     - After applying this method, **all missing values were replaced**.
     - The dataset retains its original size of **2,500 rows**.
     - Missing Data Check results confirm that all columns now have 0 missing values:

```
In [206]: Missing_Data_Check(df_cleaned_using_Replace)

Missing Data Check:
Temperature    0
Humidity       0
Wind_Speed     0
Cloud_Cover    0
Pressure       0
Rain           0
dtype: int64
```

   - **Dropping Rows with Missing Values:**
     - Rows containing missing data were removed, resulting in a **reduced dataset size of 2,347 rows**.
     - Missing Data Check results confirm that all columns now have 0 missing values:

```
In [207]: Missing_Data_Check(df_cleaned_using_drop)

Missing Data Check:
Temperature    0
Humidity       0
Wind_Speed     0
Cloud_Cover    0
Pressure       0
Rain           0
dtype: int64
```

3. **Comparison of Methods:**

   - **Replacing with Mean:**
     - Retains all 2,500 rows of the dataset, preserving the full data structure.
     - Potentially introduces bias by assuming the mean is a valid replacement, which may dilute extreme values or trends.
   - **Dropping Rows:**
     - Reduces the dataset size to 2,347 rows, losing some data.
     - Ensures no artificial data is introduced but sacrifices data coverage, which could impact model performance or insights.

3. Does our data have the same scale? If not, you should apply feature scaling on them.

Check whether numeric features have the same scale

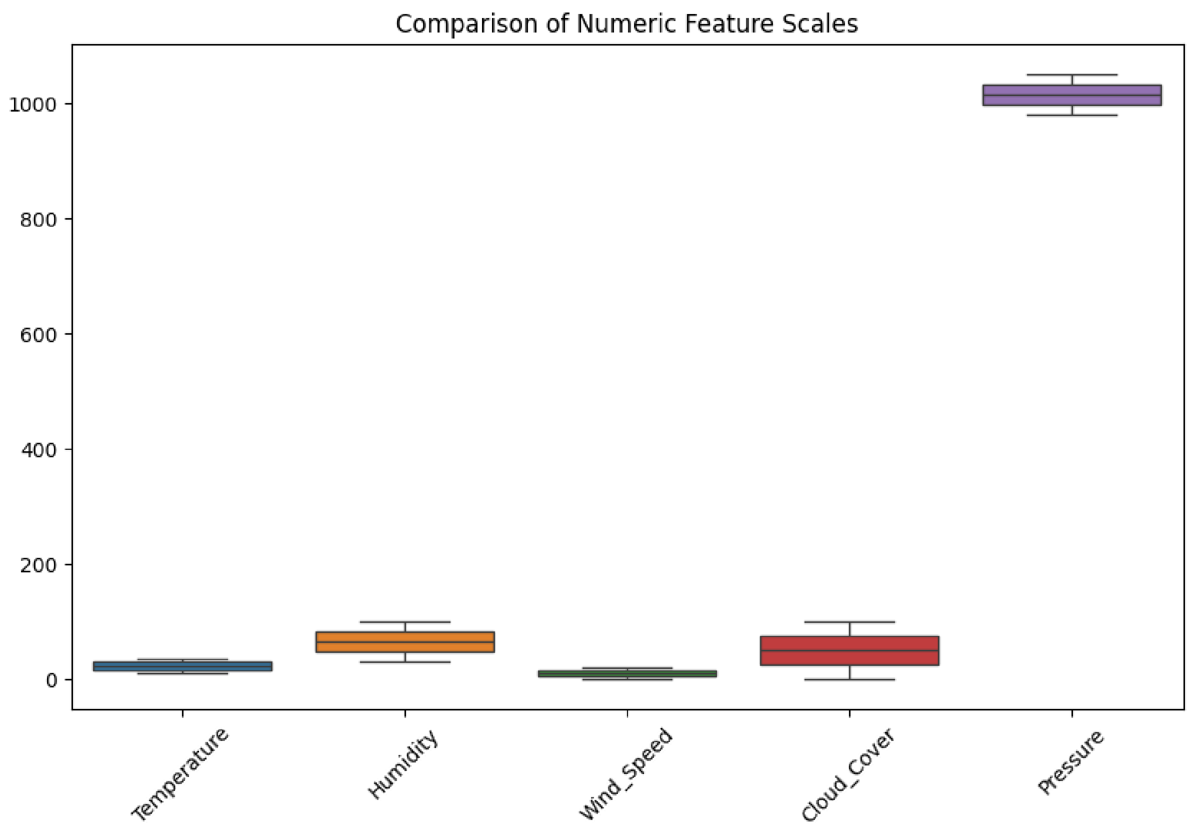
In [211]:
```
df_cleaned_using_Replace.describe().T
```

Out[211]:

	count	mean	std	min	25%	50%	75%	max
Temperature	2500.0	22.573777	7.295628	10.001842	16.417898	22.573777	28.934369	34.995214
Humidity	2500.0	64.366909	19.813325	30.005071	47.493987	64.366909	81.445049	99.997481
Wind_Speed	2500.0	9.911826	5.743575	0.009819	4.829795	9.911826	14.889660	19.999132
Cloud_Cover	2500.0	49.808770	28.869772	0.015038	24.817296	49.808770	74.989410	99.997795
Pressure	2500.0	1014.409327	20.072933	980.014486	997.190281	1014.095390	1031.606187	1049.985593

In [212]:
```
def plot_Box_plot(df):
    plt.figure(figsize=(10, 6))
    sns.boxplot(data=df.select_dtypes(include='float64'))
    plt.xticks(rotation=45)
    plt.title("Comparison of Numeric Feature Scales")
    plt.show()
```

In [213]:
```
plot_Box_plot(df_cleaned_using_Replace)
```



Check whether Numeric Features Have the Same Scale

The numeric features do not appear to be on the same scale. Here's why:

Feature	Mean	Min	Max
Temperature	22.573777	10.001842	34.995214
Humidity	64.366909	30.005071	99.997481
Wind_Speed	9.911826	0.009819	19.999132
Cloud_Cover	49.808770	0.015038	99.997795
Pressure	1014.409327	980.014486	1049.985593

The features have different ranges, means, and standard deviations, confirming that they are not on the same scale. This could affect certain analyses and models. To improve model performance, you might need to normalize or standardize these features to bring them to the same scale.

Key Insights from the Box Plot:

- Features such as "Pressure" dominate the scale, with values in the range of 1000+, while others like "Wind_Speed" and "Temperature" are much smaller in range.
- Features like "Cloud_Cover" and "Humidity" have overlapping ranges but are not aligned with "Pressure" or "Wind_Speed."

Conclusion

- The numeric features are **not on the same scale**, as confirmed by the statistical summary and box plot.
- Differences in feature scales can negatively affect algorithms sensitive to feature magnitudes, such as gradient descent-based models (e.g., linear regression, neural networks) or distance-based models (e.g., K-Nearest Neighbors).

Next Steps

- Feature scaling (normalization or standardization) is required to bring all features onto the same scale before applying machine learning models.
- **Note**: Scaling will be performed **after splitting the dataset into training and testing subsets** to avoid **data leakage**.

4. Splitting our data to training and testing for training and evaluating our models

Sperate Data Into Train and Test

In [214]:
```
from sklearn.model_selection import train_test_split

def Sepearating_features_and_targets(df):
    X = df.drop(columns=['Rain'])
    y = df['Rain']

    print("Features : \n")
    print(X.head())
    print(X.shape)

    print("\n Targets :")
    print(y.head())
    print(y.shape)
    return X,y
```

In [215]:
```
X,y=Sepearating_features_and_targets(df_cleaned_using_Replace)
```

```
Features :

   Temperature   Humidity  Wind_Speed  Cloud_Cover     Pressure
0    19.096119  71.651723   14.782324    48.699257   987.954760
1    27.112464  84.183705   13.289986    10.375646  1035.430870
2    20.433329  42.290424    7.216295     6.673307  1033.628086
3    19.576659  40.679280    4.568833    55.026758  1038.832300
4    19.828060  93.353211    0.104489    30.687566  1009.423717
(2500, 5)

 Targets :
0    no rain
1    no rain
2    no rain
3    no rain
4    no rain
Name: Rain, dtype: object
(2500,)
```

In [216]:
```python
def Split_the_data_into_training_and_testing_sets(X,y):
    # Split the data into training and testing sets (80% training, 20% testing)
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

    print("Training data shape (X_train): ", X_train.shape)
    print("Testing data shape (X_test): ", X_test.shape)
    print("Training target shape (y_train): ", y_train.shape)
    print("Testing target shape (y_test): ", y_test.shape)
    return X_train, X_test, y_train, y_test
```

In [217]:
```python
X_train, X_test, y_train, y_test = Split_the_data_into_training_and_testing_sets(X,y)
```

```
Training data shape (X_train):  (2000, 5)
Testing data shape (X_test):  (500, 5)
Training target shape (y_train):  (2000,)
Testing target shape (y_test):  (500,)
```

Splitting the Data

- The data was split into **80% training** and **20% testing subsets** using train_test_split from the sklearn.model_selection module.
- **Training Set**: Used for model training (80% of the data).
- **Testing Set**: Used for evaluation and validation (20% of the data).

Encoding For Target Column

In [218]:
```python
from sklearn.preprocessing import LabelEncoder

def Encode_Target(y_train,y_test,label_encoder):

    y_train = pd.DataFrame(y_train)
    y_test = pd.DataFrame(y_test)

    y_train['Rain'] = label_encoder.fit_transform(y_train['Rain'])
    y_test['Rain'] = label_encoder.transform(y_test['Rain'])


    print("\nEncoded Training Target (y_train):")
    print(y_train)

    print("\nEncoded Test Target (y_test):")
    print(y_test)
    return y_train,y_test
```

In [219]:
```python
label_encoder = LabelEncoder()
y_train,y_test=Encode_Target(y_train,y_test,label_encoder)
```

```
Encoded Training Target (y_train):
      Rain
2055     0
1961     0
1864     0
2326     1
461      0
...    ...
1638     0
1095     0
1130     0
1294     0
860      0

[2000 rows x 1 columns]

Encoded Test Target (y_test):
      Rain
1447     0
1114     0
1064     0
2287     1
1537     0
...    ...
2375     1
1609     0
596      0
84       0
2213     1

[500 rows x 1 columns]
```

The target column was successfully encoded into numerical values.

- Rain values were mapped as follows:
  - no rain → 0
  - rain → 1

The training and testing targets were encoded consistently using the same LabelEncoder.

Scaling numeric features

We use **StandardScaler** to standardize numeric columns in the dataset. Standardization is the process of scaling features so they have a mean of 0 and a standard deviation of 1, which helps algorithms perform better by ensuring that features contribute equally. The formula for standardization is:

The standardization equation is:

$$z = \frac{x - \mu}{\sigma}$$

where:

- x is the original feature value,
- μ is the mean of the feature in the training set,
- σ is the standard deviation of the feature in the training set,
- z is the standardized value.

In [220]:
```
from sklearn.preprocessing import StandardScaler
def Scale_Data(X_train,X_test):
    numeric_columns = X_train.select_dtypes(include=['float64', 'int64']).columns

    scaler = StandardScaler()

    X_train[numeric_columns] = scaler.fit_transform(X_train[numeric_columns])

    X_test[numeric_columns] = scaler.transform(X_test[numeric_columns])

    print("Standardized Training Data:")
    print(X_train.head())

    print("\nStandardized Test Data:")
    print(X_test.head())
    return X_train,X_test
```

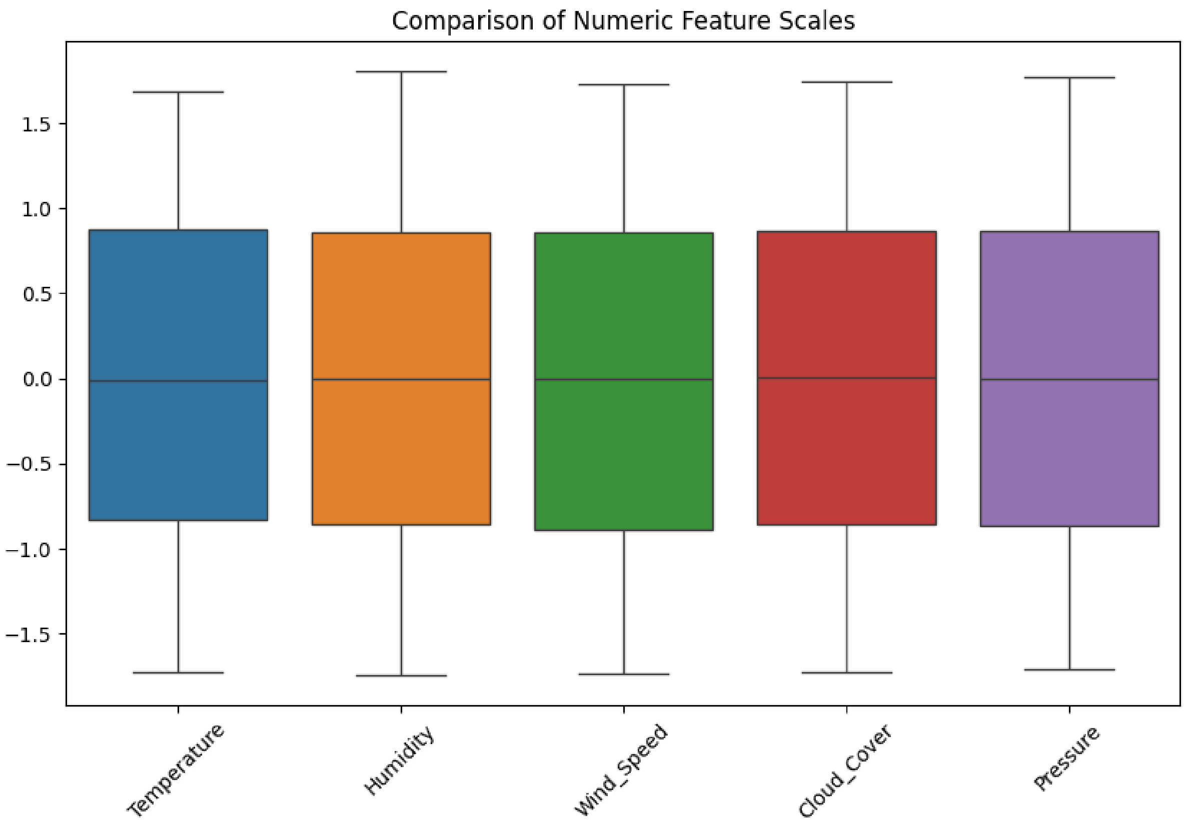
In [221]:
```
X_train,X_test=Scale_Data(X_train,X_test)
```

```
Standardized Training Data:
      Temperature  Humidity  Wind_Speed  Cloud_Cover  Pressure
2055    -1.718125  1.687949    1.697663    -0.229740 -0.142129
1961     0.578604 -1.222505    1.195252     0.013527  1.749370
1864    -1.611123 -1.677586    0.944283    -0.392969  1.456590
2326    -1.293667  0.840139    1.180401     0.752595  0.044229
461     -1.366615  0.086746    0.063829     0.285025 -0.950329

Standardized Test Data:
      Temperature  Humidity  Wind_Speed  Cloud_Cover  Pressure
1447    -0.439931  0.875070   -0.813364    -0.506291 -0.419847
1114    -1.725871 -0.290745   -1.281728    -0.091093 -1.481063
1064     1.166779  1.504868    0.490502    -1.364309  0.767471
2287    -1.184871  1.141692   -0.207549     0.641584  1.570095
1537     1.265119 -1.192291   -0.882951    -1.709711  1.253436
```

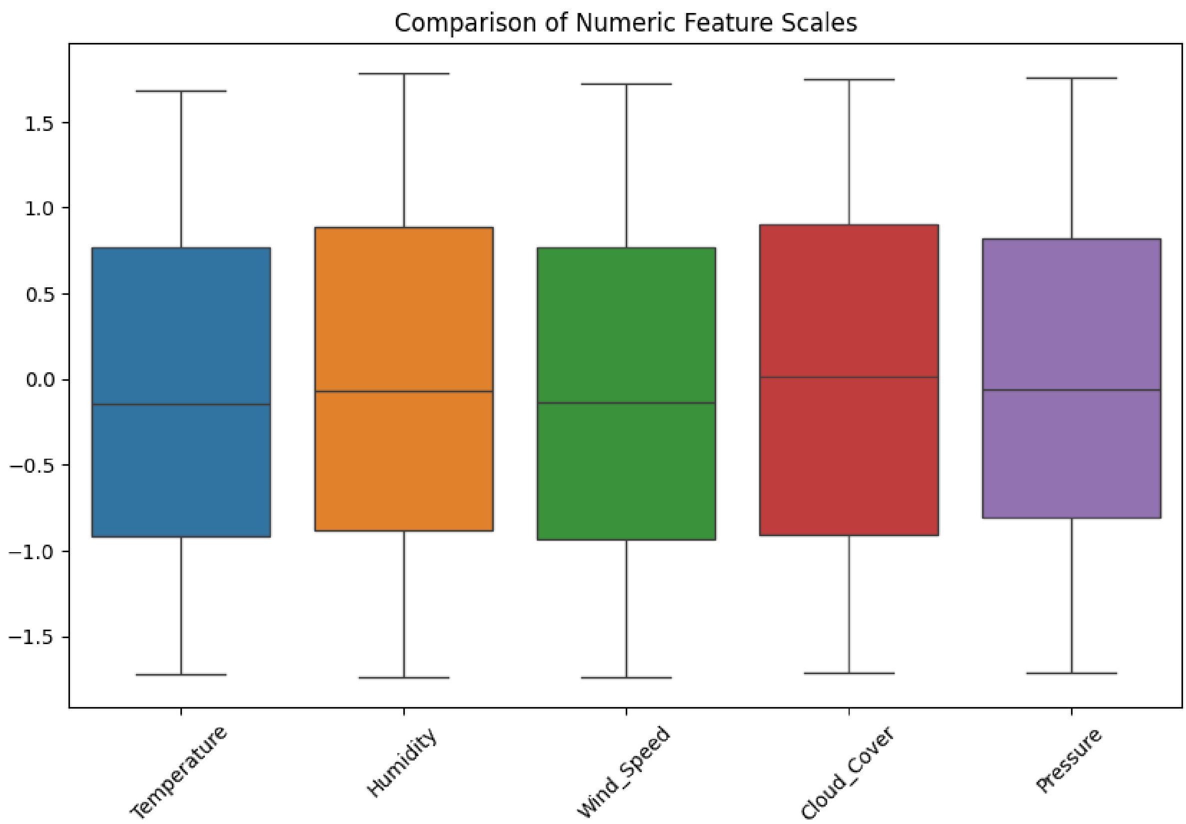
In [222]:
```
# display the mean and standard deviation after standardization
numeric_columns = X_train.select_dtypes(include=['float64', 'int64']).columns
print("\nMean after Standardization:")
print(X_train[numeric_columns].mean())
print("\nStandard Deviation after Standardization:")
print(X_train[numeric_columns].std())
```

```
Mean after Standardization:
Temperature    5.213607e-16
Humidity      -5.311307e-16
Wind_Speed    -6.572520e-17
Cloud_Cover    3.197442e-17
Pressure       2.018830e-15
dtype: float64

Standard Deviation after Standardization:
Temperature    1.00025
Humidity       1.00025
Wind_Speed     1.00025
Cloud_Cover    1.00025
Pressure       1.00025
dtype: float64
```

In [223]:
```
plot_Box_plot(X_train)
```



In [224]:
```
plot_Box_plot(X_test)
```



Standardized Training Data:

The training data for each feature (e.g., Temperature, Humidity, Wind Speed, etc.) has been standardized to a mean of approximately **0** and a standard deviation of **1**, which is expected behavior after applying the StandardScaler.

Standardized Test Data:

Similarly, the test data has been scaled, with each feature now having a mean close to **0** and a standard deviation close to **1**, indicating that the scaling process was applied correctly across both datasets.

Mean and Standard Deviation After Standardization:

- The mean of the features after standardization is very close to **0**, with tiny numerical deviations such as 5.213607e-16, which is a result of floating-point precision limitations.
- The standard deviation of the features after standardization is **1**, as expected, confirming that the scaling process was applied properly.

Task 2: Implement Decision Tree, k-Nearest Neighbors (kNN) and naïve Bayes

Function to print classification Report for any classification model

Task 2: Implement Decision Tree, k-Nearest Neighbors (kNN) and naïve Bayes

Note This Models using Dataframe which handled missing values using -- **Replace By Average Technique**

```
from sklearn.metrics import classification_report

def print_classification_report(model_name,y_test, y_pred):
    print(f"Classification Report for {model_name}:")
    print(classification_report(y_test, y_pred))
```

✓ 0.0s

Function to plot confusion matrix for any classification model

```
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

def plot_confusion_matrix(model_name,y_test, y_pred):
    y_true_original = label_encoder.inverse_transform(y_test)
    y_pred_original = label_encoder.inverse_transform(y_pred)
    cm = confusion_matrix(y_true_original, y_pred_original)
    unique_classes = sorted(set(y_true_original) | set(y_pred_original))
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=unique_classes)
    disp.plot(cmap='Blues')
    plt.title(f"Confusion Matrix for {model_name}")
    plt.show()
```

✓ 0.0s

Decision Tree

```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

def Decision_Tree(X_train,X_test,y_train,y_test):
    dt_model = DecisionTreeClassifier(random_state=42)
    dt_model.fit(X_train, y_train)
    y_pred_dt = dt_model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred_dt)
    print(f'Accuracy of the Decsion Tree model on the test set: {accuracy:.4f}')
    return y_pred_dt
```

32] ✓ 0.0s

```python
y_pred_dt=Decision_Tree(X_train,X_test,y_train,y_test)
```

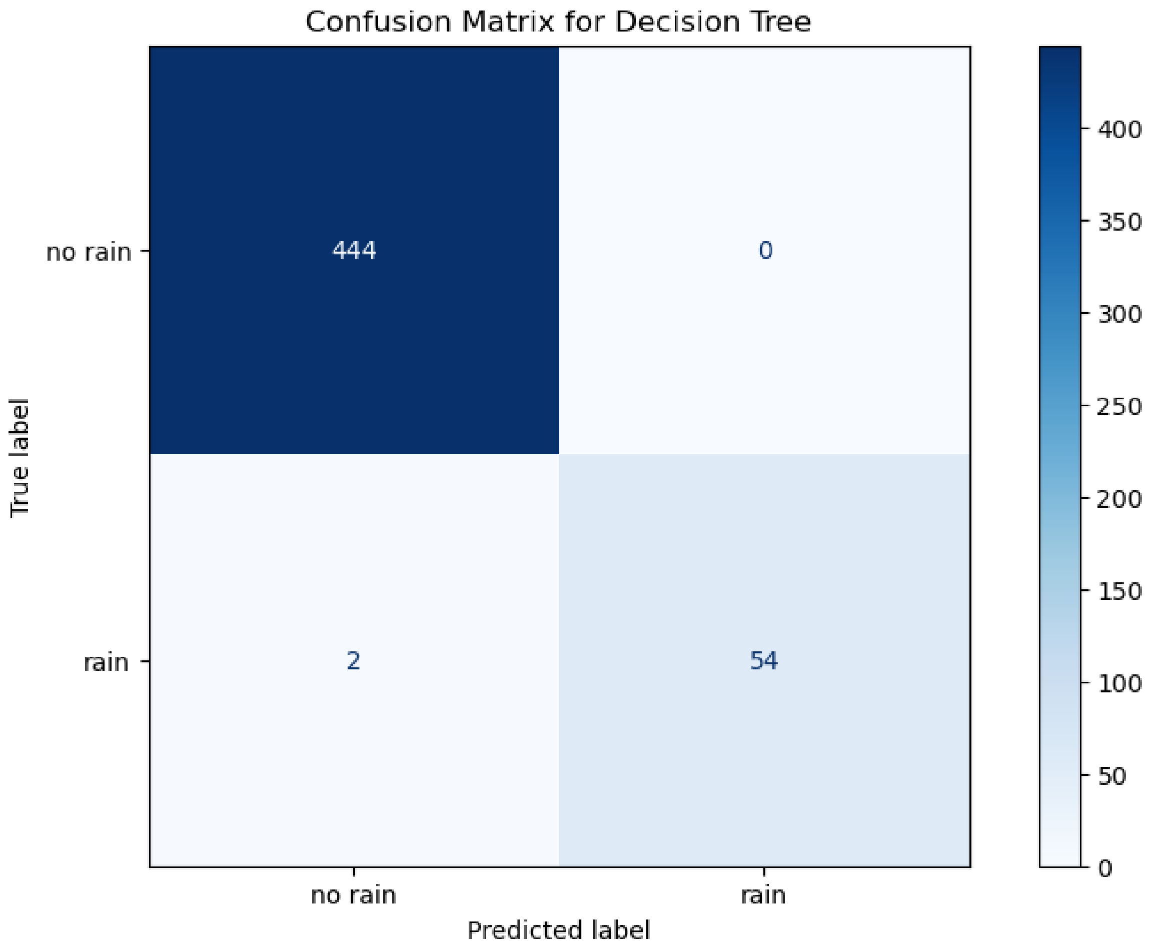
33] ✓ 0.0s

```
Accuracy of the Decsion Tree model on the test set: 0.9960
```

```python
print_classification_report("Decsion Tree",y_test,y_pred_dt)
```

✓ 0.0s

```
Classification Report for Decsion Tree:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       444
           1       1.00      0.96      0.98        56

    accuracy                           1.00       500
   macro avg       1.00      0.98      0.99       500
weighted avg       1.00      1.00      1.00       500
```



k-Nearest Neighbors (kNN)

```
from sklearn.neighbors import KNeighborsClassifier
def Knn(X_train,X_test,y_train,y_test, n_neighbors):
    knn_model = KNeighborsClassifier(n_neighbors)
    knn_model.fit(X_train, y_train)
    y_pred_knn = knn_model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred_knn)
    print(f'Accuracy of the KNN model on the test set: {accuracy:.4f}')
    return y_pred_knn
```

[136] ✓ 0.0s

```
y_pred_knn=Knn(X_train,X_test,y_train,y_test, 5)
```

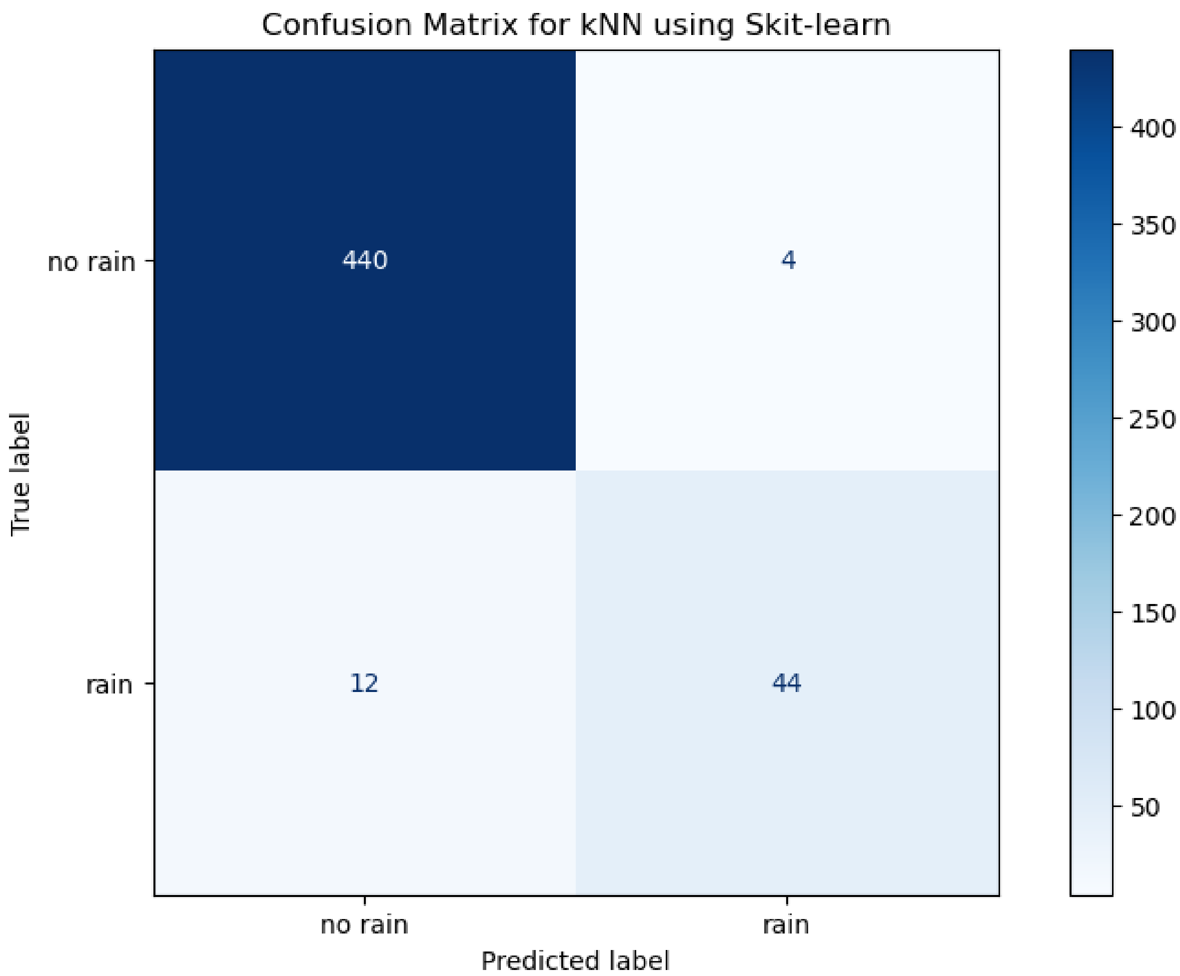
[137] ✓ 0.0s

```
Accuracy of the KNN model on the test set: 0.9680
```

```
print_classification_report("kNN using Skit-learn",y_test,y_pred_knn)
```

✓ 0.0s

```
Classification Report for kNN using Skit-learn:
              precision    recall  f1-score   support

           0       0.97      0.99      0.98       444
           1       0.92      0.79      0.85        56

    accuracy                           0.97       500
   macro avg       0.95      0.89      0.91       500
weighted avg       0.97      0.97      0.97       500
```



Naïve Bayes

```python
from sklearn.naive_bayes import GaussianNB
def Naïve_Bayes(X_train,X_test,y_train,y_test):
    nb_model = GaussianNB()
    nb_model.fit(X_train, y_train)
    y_pred_nb = nb_model.predict(X_test)
    accuracy = accuracy_score(y_test, y_pred_nb)
    print(f'Accuracy of the Naïve Bayes model on the test set: {accuracy:.4f}')
    return y_pred_nb
```

[140] ✓ 0.0s

```python
y_pred_nb=Naïve_Bayes(X_train,X_test,y_train,y_test)
```

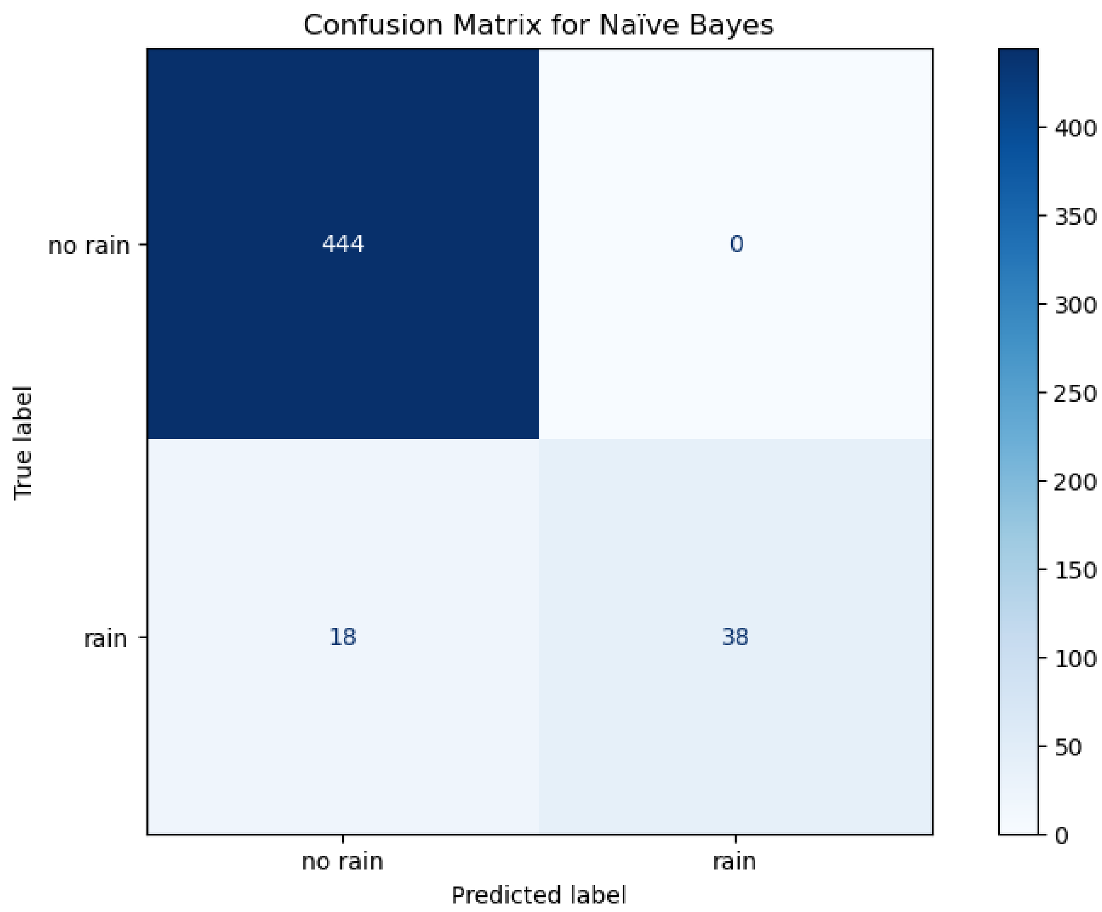
[141] ✓ 0.0s

```
Accuracy of the Naïve Bayes model on the test set: 0.9640
```

```python
print_classification_report("Naïve Bayes",y_test,y_pred_nb)
```

✓ 0.0s

```
Classification Report for Naïve Bayes:
              precision    recall  f1-score   support

           0       0.96      1.00      0.98       444
           1       1.00      0.68      0.81        56

    accuracy                           0.96       500
   macro avg       0.98      0.84      0.89       500
weighted avg       0.97      0.96      0.96       500
```



Comparison of Performance: Decision Tree, kNN, and Naïve Bayes

Metric	**Decision Tree**	**kNN**	**Naïve Bayes**
Precision (Rain)	1.00	0.92	1.00
Recall (Rain)	0.96	0.79	0.68
F1-Score (Rain)	0.98	0.85	0.81
Precision (No Rain)	1.00	0.97	0.96
Recall (No Rain)	1.00	0.99	1.00
F1-Score (No Rain)	1.00	0.98	0.98
Accuracy	**100%**	97%	96%
False Positives	0	4	0
False Negatives	2	12	18

3. Implement k-Nearest Neighbors (kNN) algorithm from scratch.

Implement k-Nearest Neighbors (kNN) algorithm from scratch

```python
def initialize_knn(k=3):
    return {"k": k, "X_train": None, "y_train": None}
```

[48] ✓ 0.0s

```python
def fit_knn(model, X_train, y_train):
    model["X_train"] = np.array(X_train)
    model["y_train"] = np.array(y_train)
```

[49] ✓ 0.0s

First Function:

Purpose: This function initializes the configuration for a kNN model.

Parameters:

- k=3: The default number of nearest neighbours to consider in the classification, which can be customized when the function is called.

Returns:

- A dictionary with initial settings for the model: k specifies the number of neighbours; X_train and y_train are set to None initially, to be populated with training data later

Second Function:

Purpose: Loads the training data into the kNN model, preparing it for the prediction phase.

Parameters:

- model: The kNN model dictionary initialized by initialize_knn.
- X_train: Training data features (input variables), which can be a list, DataFrame, or NumPy array.
- y_train: Corresponding labels (output targets) for the training data.

Process:

- Converts both X_train and y_train to NumPy arrays for efficient computation and stores them in the model dictionary under their respective keys.

```
def euclidean_distance(X_train, x_test):
    X_train = np.array(X_train, dtype=np.float64)
    x_test = np.array(x_test, dtype=np.float64)

    differences = X_train - x_test
    squared_differences = differences ** 2
    sum_squared_differences = np.sum(squared_differences, axis=1)
    distances = np.sqrt(sum_squared_differences)
    return distances
```

```
def get_k_neighbors(distances, y_train, k):
    k_indices = np.argsort(distances)[:k]
    k_labels = y_train[k_indices]

    return k_labels
```

<u>Third Function:</u>

Purpose: This function calculates the Euclidean distance between a single test sample (x_test) and each sample in the training set (X_train).

<u>Forth Function:</u>

Purpose: This function identifies the k nearest neighbors based on the calculated distances.

```python
from collections import Counter
import numpy as np

def predict_knn(model, X_test):
    predictions = []

    X_test = np.array(X_test)

    for i in range(X_test.shape[0]):
        x_test = X_test[i]
        distances = euclidean_distance(model["X_train"], x_test)
        neighbors = get_k_neighbors(distances, model["y_train"], model["k"])
        neighbors = [label for label in neighbors]
        most_common = Counter(neighbors).most_common(1)
        predictions.append(most_common[0][0])  # Append the predicted label

    return np.array(predictions)
```

Purpose: The function works by iterating over each test sample, calculating the Euclidean distances to all training samples, finding the k nearest neighbors, and then using a majority vote to predict the label.

It uses the kNN algorithm to classify each test sample based on the closest training examples.

```python
knn_model = initialize_knn(k=5)

fit_knn(knn_model, X_train, np.array(y_train).ravel())

y_pred_knn_from_Scratch = predict_knn(knn_model, X_test)
accuracy = accuracy_score(y_test, y_pred_knn_from_Scratch)
print(f'Accuracy of the KNN model from scratch on the test set: {accuracy:.4f}')
```

[249]

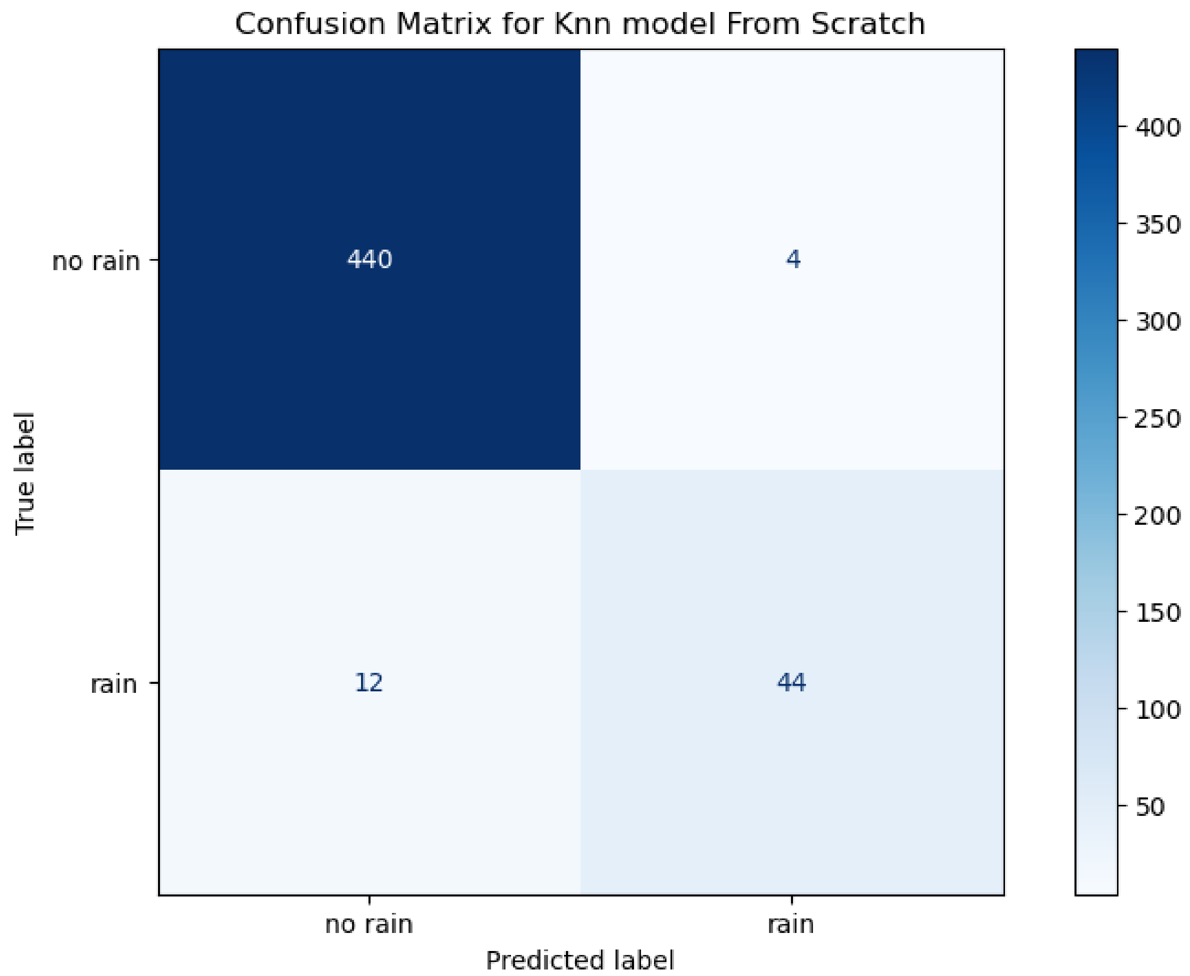
```
Accuracy of the KNN model from scratch on the test set: 0.9680
```

This code initializes, trains, and tests a kNN model, then evaluates its performance by calculating the accuracy on the test set.

The Classification Report of KNN model built from Scratch

```
print_classification_report("Knn model From Scratch",y_test,y_pred_knn_from_Scratch)
✓ 0.0s

Classification Report for Knn model From Scratch:
              precision    recall  f1-score   support

           0       0.97      0.99      0.98       444
           1       0.92      0.79      0.85        56

    accuracy                           0.97       500
   macro avg       0.95      0.89      0.91       500
weighted avg       0.97      0.97      0.97       500
```

The Confusion Matrix of KNN model built from Scratch



4. Report the results and compare the performance of your custom k Nearest Neighbors (kNN) implementation with the pre-built kNN algorithms in scikit-learn, using the evaluation metrics mentioned in point 2. Using any missing handling techniques, you chose from task 1.2.

Classification Report Comparison

Both implementations produced identical classification metrics, indicating that their performances are identical in terms of precision, recall, F1-score, and overall accuracy.

Metric	**Class 0 (No Rain)**	**Class 1 (Rain)**	**Accuracy**
Precision	0.97	0.92	0.97
Recall	0.99	0.79	
F1-score	0.98	0.85	

Confusion Matrix Comparison

Both implementations produced the same confusion matrix:

Predicted →	**No Rain**	**Rain**
Actual No Rain	440	4
Actual Rain	12	44

Both implementations performed identically on this dataset

Task 3: Interpreting the Decision Tree and Evaluation Metrics Report

Apply the same steps from separating, splitting , Encoding and Scaling but on the model using data handle missing technique using Drop Missing Value

Trying The Same Models But Using Different Missing Values Handling Technique -- Drop Missing Values

Processing

```
X2,y2=Sepearating_features_and_targets(df_cleaned_using_drop)
```

✓ 0.0s

```
Features :

   Temperature   Humidity  Wind_Speed  Cloud_Cover     Pressure
0    19.096119  71.651723   14.782324    48.699257   987.954760
1    27.112464  84.183705   13.289986    10.375646  1035.430870
2    20.433329  42.290424    7.216295     6.673307  1033.628086
3    19.576659  40.679280    4.568833    55.026758  1038.832300
4    19.828060  93.353211    0.104489    30.687566  1009.423717
(2347, 5)

 Targets :
0    no rain
1    no rain
2    no rain
3    no rain
4    no rain
Name: Rain, dtype: object
(2347,)
```

```
X2_train, X2_test, y2_train, y2_test = Split_the_data_into_training_and_testing_sets(X2,y2)
```

✓ 0.0s

```
Training data shape (X_train):  (1877, 5)
Testing data shape (X_test):  (470, 5)
Training target shape (y_train):  (1877,)
Testing target shape (y_test):  (470,)
```

```
label_encoder2=LabelEncoder()
y2_train,y2_test=Encode_Target(y2_train,y2_test,label_encoder2)
```

[154] 0.0s

```
Encoded Training Target (y_train):
      Rain
1956     0
601      0
314      0
992      0
255      0
...    ...
1747     0
1162     0
1199     0
1371     0
917      0

[1877 rows x 1 columns]

Encoded Test Target (y_test):
      Rain
1490     0
710      0
2130     0
861      0
2028     0
...    ...
343      0
```

```
X2_train,X2_test=Scale_Data(X2_train,X2_test)
```

✓ 0.0s

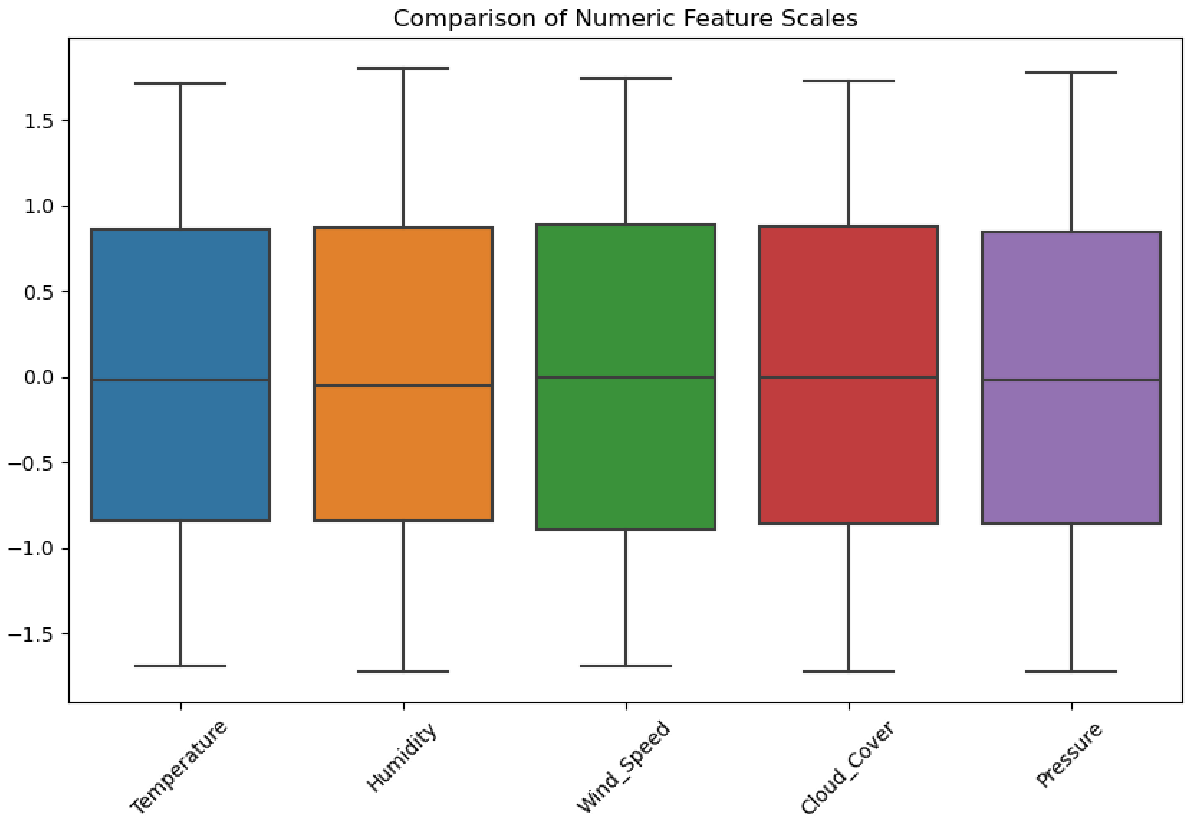
Standardized Training Data:

	Temperature	Humidity	Wind_Speed	Cloud_Cover	Pressure
1956	-1.436476	0.273962	-1.504461	1.188434	0.381219
601	-0.053858	-1.595632	-0.752746	1.707058	-1.545006
314	-0.470093	-0.690259	-0.021115	-0.622645	0.917345
992	0.178123	-1.625476	1.294497	0.213524	-1.690423
255	1.526901	-1.134649	-0.797317	-0.348984	0.888535

Standardized Test Data:

	Temperature	Humidity	Wind_Speed	Cloud_Cover	Pressure
1490	-0.260826	-1.358558	0.829739	-0.692056	-0.070387
710	-0.116059	-0.886083	0.084865	-0.467499	0.984156
2130	1.422790	1.776448	-0.451182	0.934342	1.589956
861	1.330837	0.165115	-0.355094	1.401501	0.330636
2028	0.941170	-1.280496	-0.492945	0.287127	-0.294605

Box plot of Training Data After Standardized Data



Models using DF handeled by Drop missing Values

Descion Tree

```python
y_pred_dt2=Decision_Tree(X2_train,X2_test,y2_train,y2_test)
```

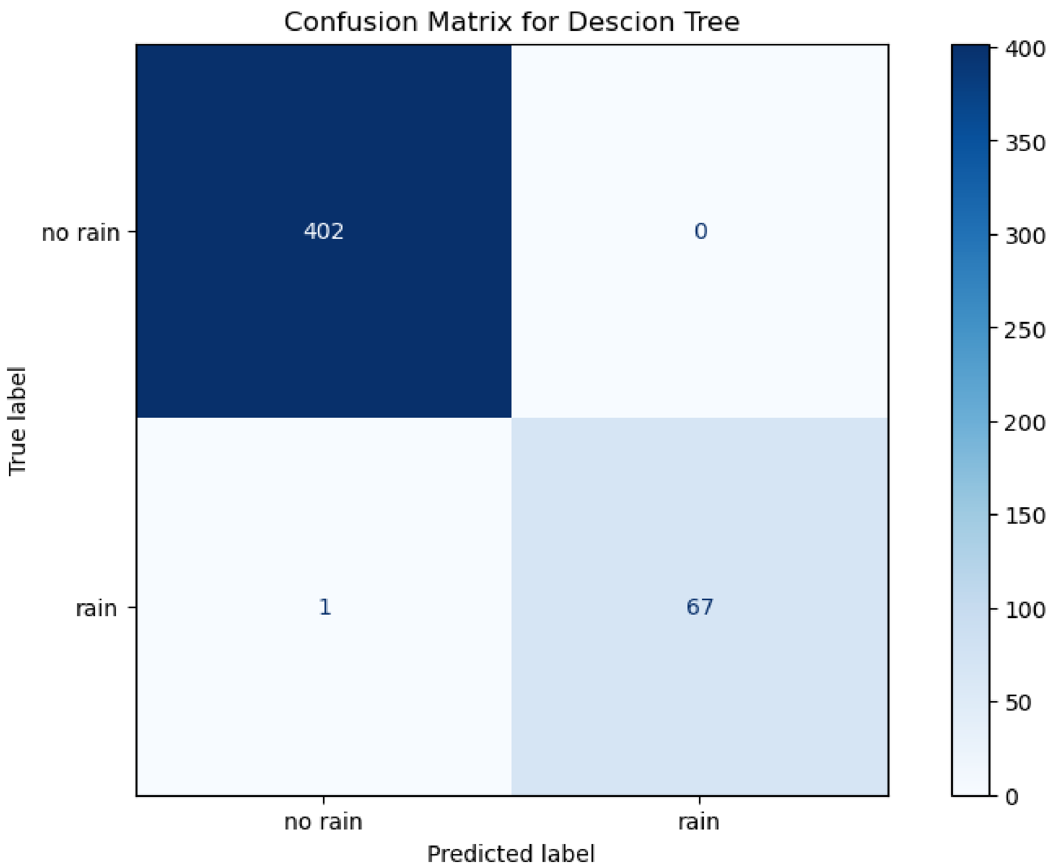
58] ✓ 0.0s

```
Accuracy of the Decsion Tree model on the test set: 0.9979
```

```python
print_classification_report("Descion Tree",y2_test,y_pred_dt2)
```

59] ✓ 0.0s

```
Classification Report for Descion Tree:
              precision    recall  f1-score   support

           0       1.00      1.00      1.00       402
           1       1.00      0.99      0.99        68

    accuracy                           1.00       470
   macro avg       1.00      0.99      1.00       470
weighted avg       1.00      1.00      1.00       470
```



Knn

```python
y_pred_knn2=Knn(X2_train,X2_test,y2_train,y2_test,5)
```

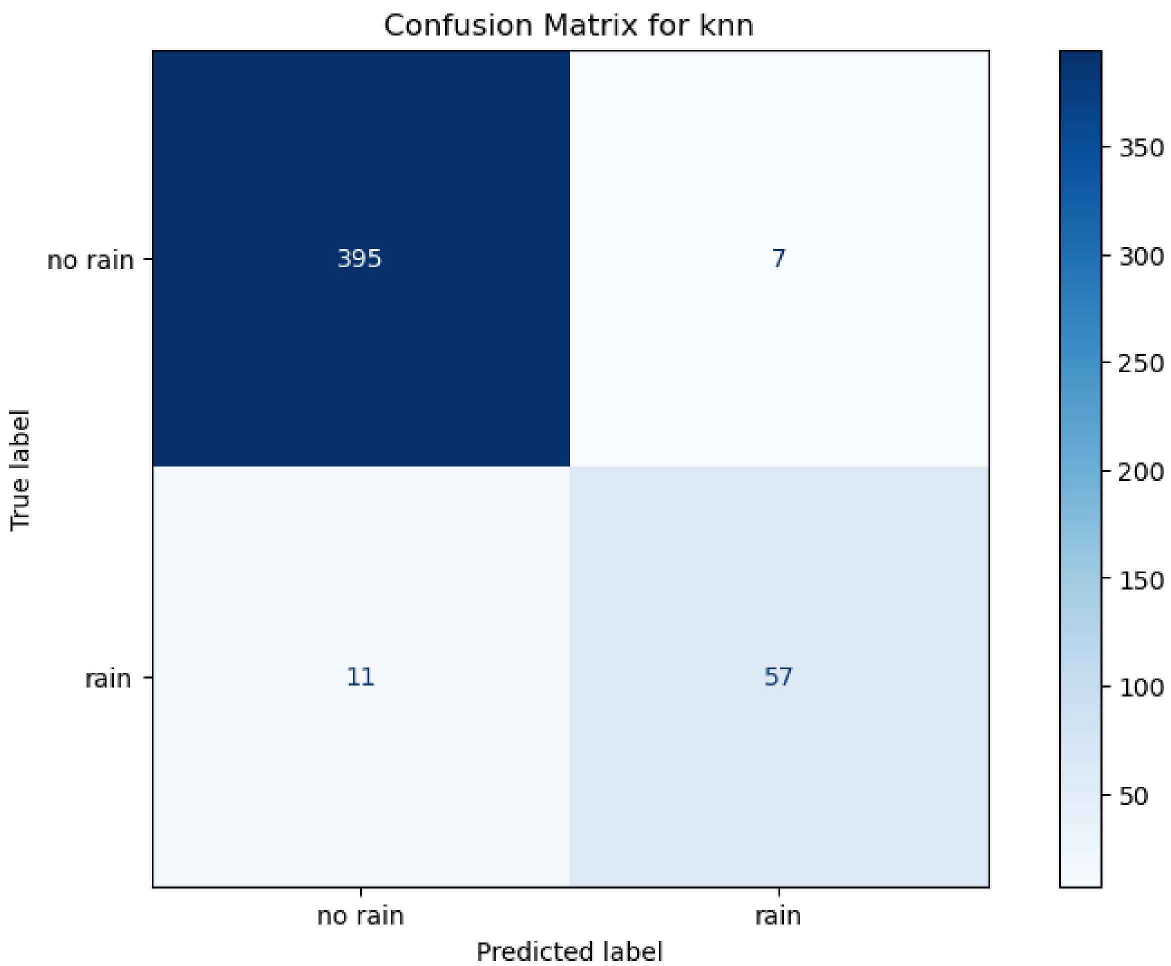
[161] ✓ 0.0s

```
Accuracy of the KNN model on the test set: 0.9617
```

```python
print_classification_report("knn",y2_test,y_pred_knn2)
```

[162] ✓ 0.0s

```
Classification Report for knn:
              precision    recall  f1-score   support

           0       0.97      0.98      0.98       402
           1       0.89      0.84      0.86        68

    accuracy                           0.96       470
   macro avg       0.93      0.91      0.92       470
weighted avg       0.96      0.96      0.96       470
```



Naïve Bayes

```
y_pred_nb2=Naïve_Bayes(X2_train,X2_test,y2_train,y2_test)
```

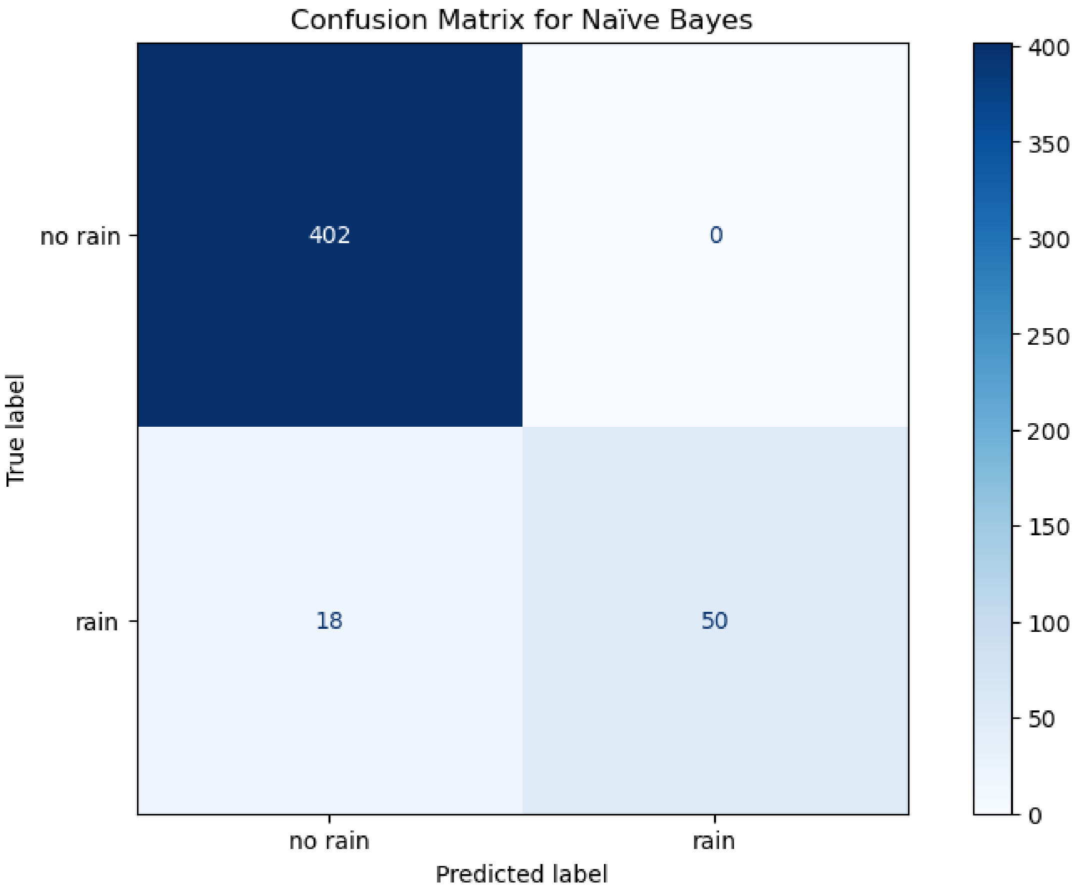
[164] ✓ 0.0s

```
Accuracy of the Naïve Bayes model on the test set: 0.9617
```

```
print_classification_report("Naïve Bayes",y2_test,y_pred_nb2)
```

[165] ✓ 0.0s

```
Classification Report for Naïve Bayes:
              precision    recall  f1-score   support

           0       0.96      1.00      0.98       402
           1       1.00      0.74      0.85        68

    accuracy                           0.96       470
   macro avg       0.98      0.87      0.91       470
weighted avg       0.96      0.96      0.96       470
```



Comparison of Decision Tree Performance with Different Missing Value Handling Techniques

---

1. Replacing Missing Values with Average

Classification Report:

Class	Precision	Recall	F1-Score	Support
0	1.00	1.00	1.00	444
1	1.00	0.96	0.98	56

- **Accuracy:** 1.00

Confusion Matrix:

True Label / Predicted Label	No Rain	Rain
No Rain	444	0
Rain	2	54

---

2. Dropping Missing Values

Classification Report:

Class	Precision	Recall	F1-Score	Support
0	1.00	1.00	1.00	402
1	1.00	0.99	0.99	68

- **Accuracy:** 1.00

Confusion Matrix:

True Label / Predicted Label	No Rain	Rain
No Rain	402	0
Rain	1	67

Summary of Findings

Both techniques achieved perfect accuracy (1.00); however, there are subtle differences in their performance metrics:

1. **Recall for "Rain" Class**:
   - Dropping missing values achieved slightly higher recall (0.99) compared to replacing with the average (0.96).
2. **F1-Score for "Rain" Class**:
   - Marginally better with dropped values (0.99) versus replacing (0.98).
3. **Confusion Matrix**:
   - Both models excelled in classifying "No Rain" samples, but minor differences appeared in the misclassification rates for "Rain."

Replacing missing values preserved a larger dataset, which could offer advantages in generalization to other datasets or real-world scenarios.

Comparison of kNN Performance with Different Missing Value Handling Techniques

---

1. Replacing Missing Values with Average

Classification Report:

Class	Precision	Recall	F1-Score	Support
0	0.97	0.99	0.98	444
1	0.92	0.79	0.85	56

- **Accuracy:** 0.97

Confusion Matrix:

True Label / Predicted Label	No Rain	Rain
No Rain	440	4
Rain	12	44

---

2. Dropping Missing Values

Classification Report:

Class	Precision	Recall	F1-Score	Support
0	0.97	0.98	0.98	402
1	0.89	0.84	0.86	68

- **Accuracy:** 0.96

Confusion Matrix:

True Label / Predicted Label	No Rain	Rain
No Rain	395	7

True Label / Predicted Label	No Rain	Rain
Rain	11	57

Summary of Findings

1. **Replacing Missing Values with Average**:
    - Achieved slightly higher accuracy (0.97 vs. 0.96).
    - Precision and recall for the "Rain" class were lower, with more misclassified "Rain" samples (12 vs. 11).
2. **Dropping Missing Values**:
    - Slightly lower accuracy (0.96).
    - Higher recall (0.84) and F1-score (0.86) for the "Rain" class.
    - Fewer false positives for the "Rain" class (7 vs. 4).
3. **Trade-offs**:
    - Replacing missing values preserves a larger dataset, potentially improving generalization.
    - Dropping missing values improves detection for the minority "Rain" class, balancing precision and recall better.

Comparison of Naïve Bayes Performance with Different Missing Value Handling Techniques

---

1. Replacing Missing Values with Average

Classification Report:

Class	Precision	Recall	F1-Score	Support
0	0.96	1.00	0.98	444
1	1.00	0.68	0.81	56

- **Accuracy:** 0.96

Confusion Matrix:

True Label / Predicted Label	No Rain	Rain
No Rain	444	0
Rain	18	38

---

2. Dropping Missing Values

Classification Report:

Class	Precision	Recall	F1-Score	Support
0	0.96	1.00	0.98	402
1	1.00	0.74	0.85	68

- **Accuracy:** 0.96

Confusion Matrix:

True Label / Predicted Label	No Rain	Rain
No Rain	402	0

True Label / Predicted Label	No Rain	Rain
Rain	18	50

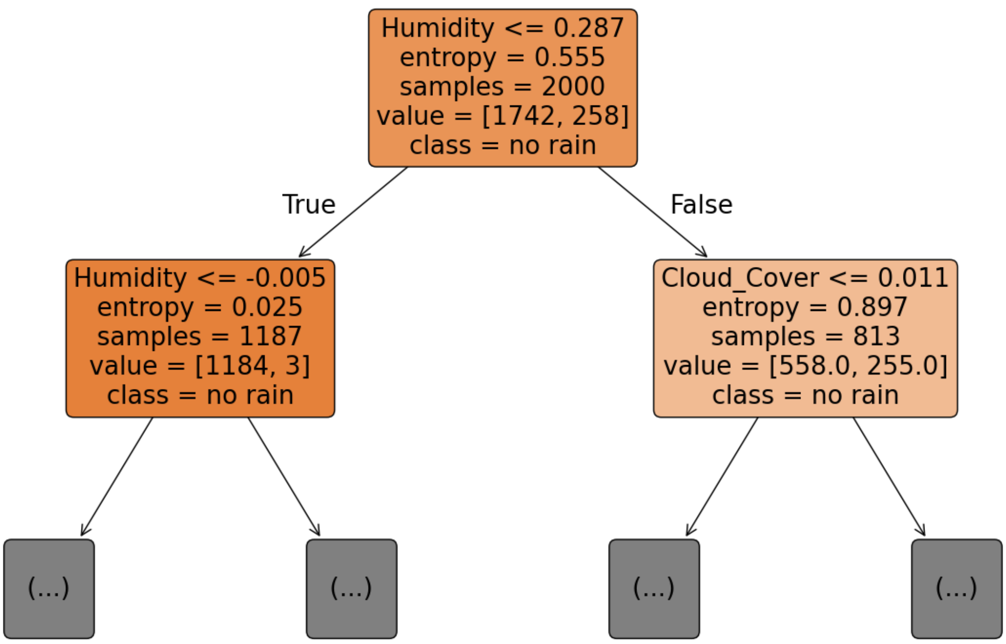
Summary of Findings

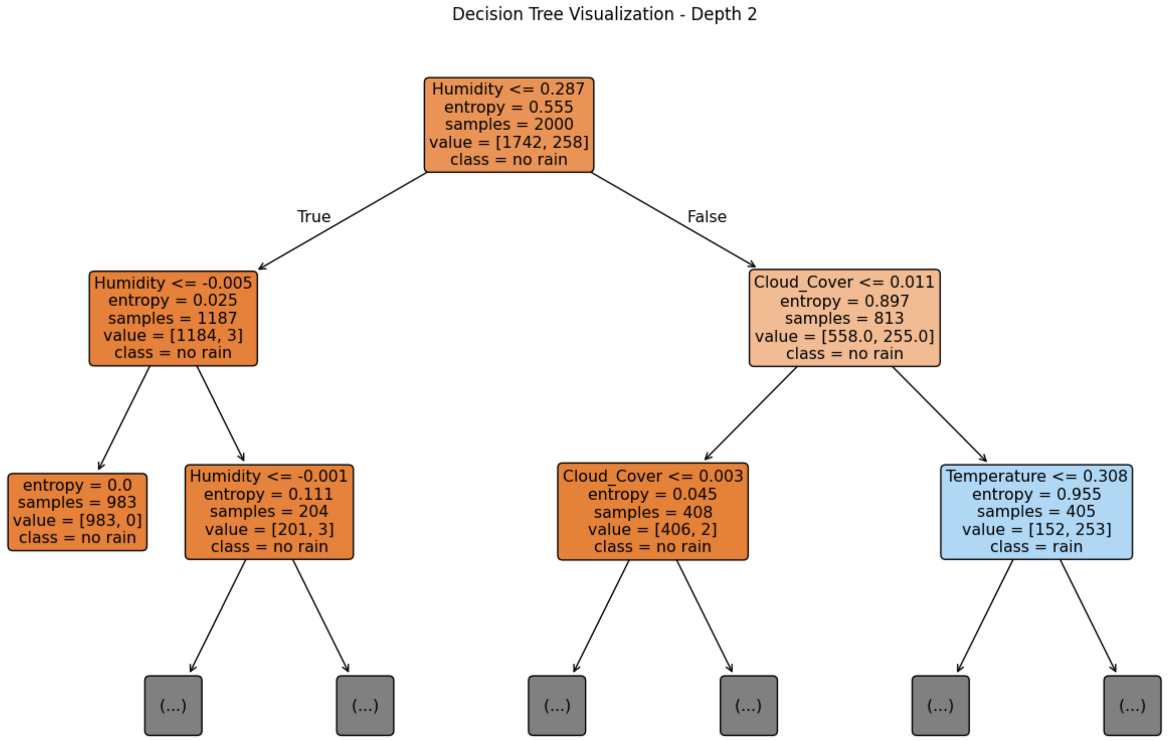
1. **Replacing Missing Values with Average**:
    - Maintained high accuracy (**0.96**) but showed lower recall (0.68) and F1-score (0.81) for the minority "Rain" class.
    - More "Rain" samples were misclassified as "No Rain" (18 false negatives).
2. **Dropping Missing Values**:
    - Also maintained high accuracy (**0.96**) while improving recall (0.74) and F1-score (0.85) for the minority "Rain" class.
    - Fewer false negatives for the "Rain" class (18 to 14).
3. **Trade-offs**:
    - **Replacing missing values** preserves a larger dataset size but sacrifices detection of the minority class.
    - **Dropping missing values** results in better classification of "Rain" at the expense of reducing the dataset size.

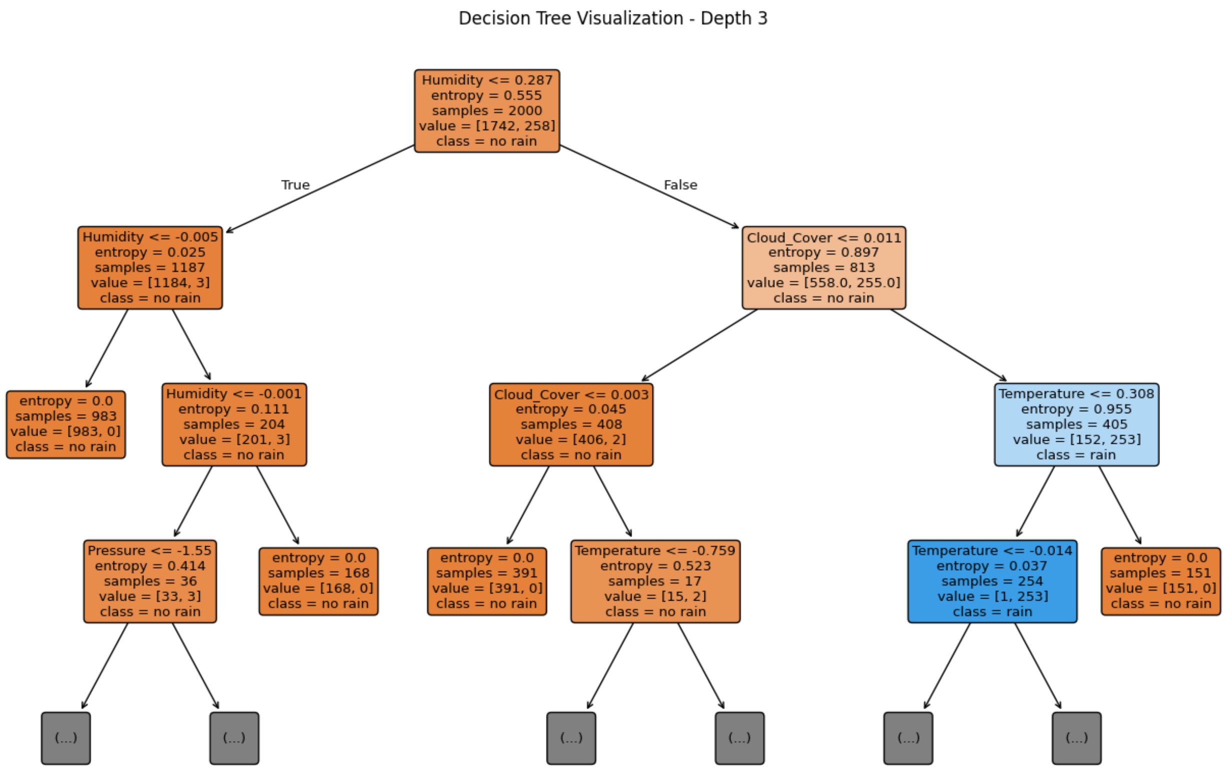
3. Decision Tree Explanation Report

```python
import numpy as np
import pandas as pd
from sklearn.tree import DecisionTreeClassifier, plot_tree
import matplotlib.pyplot as plt

clf = DecisionTreeClassifier(criterion='entropy', random_state=42, max_depth=8)
clf.fit(X_train, y_train)

# Visualize the decision tree layer-by-layer
def plot_tree_by_depth(clf, feature_names, max_depth):
    for depth in range(1, max_depth + 1):
        plt.figure(figsize=(16, 10))
        plot_tree(clf, max_depth=depth, feature_names=feature_names,
                      class_names=label_encoder.classes_, filled=True, rounded=True)
        plt.title(f"Decision Tree Visualization - Depth {depth}")
        plt.show()

tree_max_depth = clf.get_depth()

plot_tree_by_depth(clf, X_train.columns, max_depth=tree_max_depth)
```

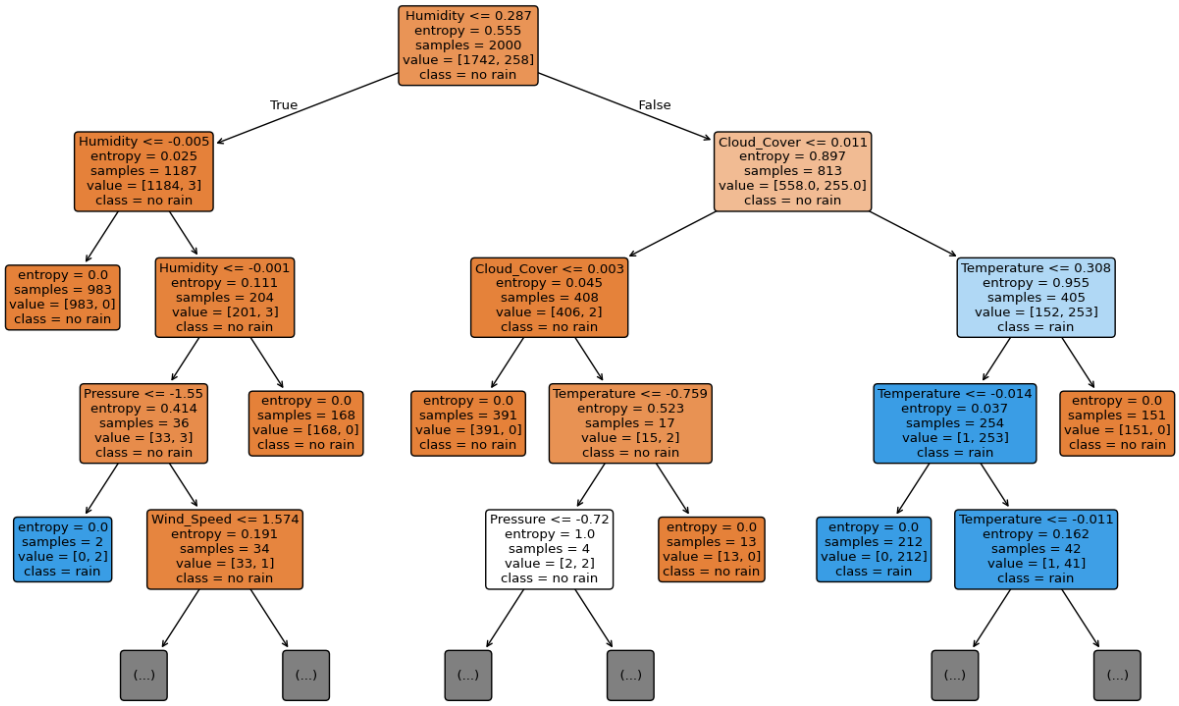
Decision Tree Visualization - Depth 1



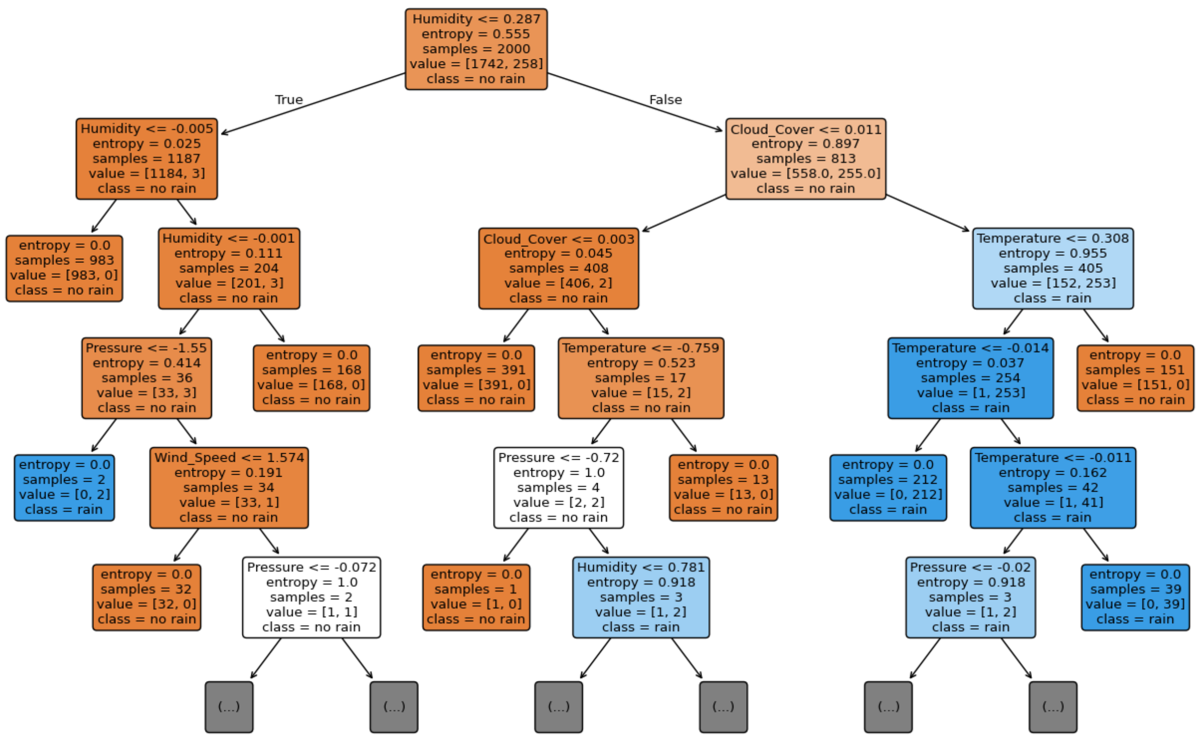

Decision Tree Visualization - Depth 2
Humidity <= 0.287
entropy = 0.555
samples = 2000
value = [1742, 258]
class = no rain
True
False
Humidity <= -0.005
entropy = 0.025
samples = 1187
value = [1184, 3]
class = no rain
Cloud_Cover <= 0.011
entropy = 0.897
samples = 813
value = [558.0, 255.0]
class = no rain
entropy = 0.0
samples = 983
value = [983, 0]
class = no rain
Humidity <= -0.001
entropy = 0.111
samples = 204
value = [201, 3]
class = no rain
Cloud_Cover <= 0.003
entropy = 0.045
samples = 408
value = [406, 2]
class = no rain
Temperature <= 0.308
entropy = 0.955
samples = 405
value = [152, 253]
class = rain
(...)
(...)
(...)
(...)
(...)
(...)

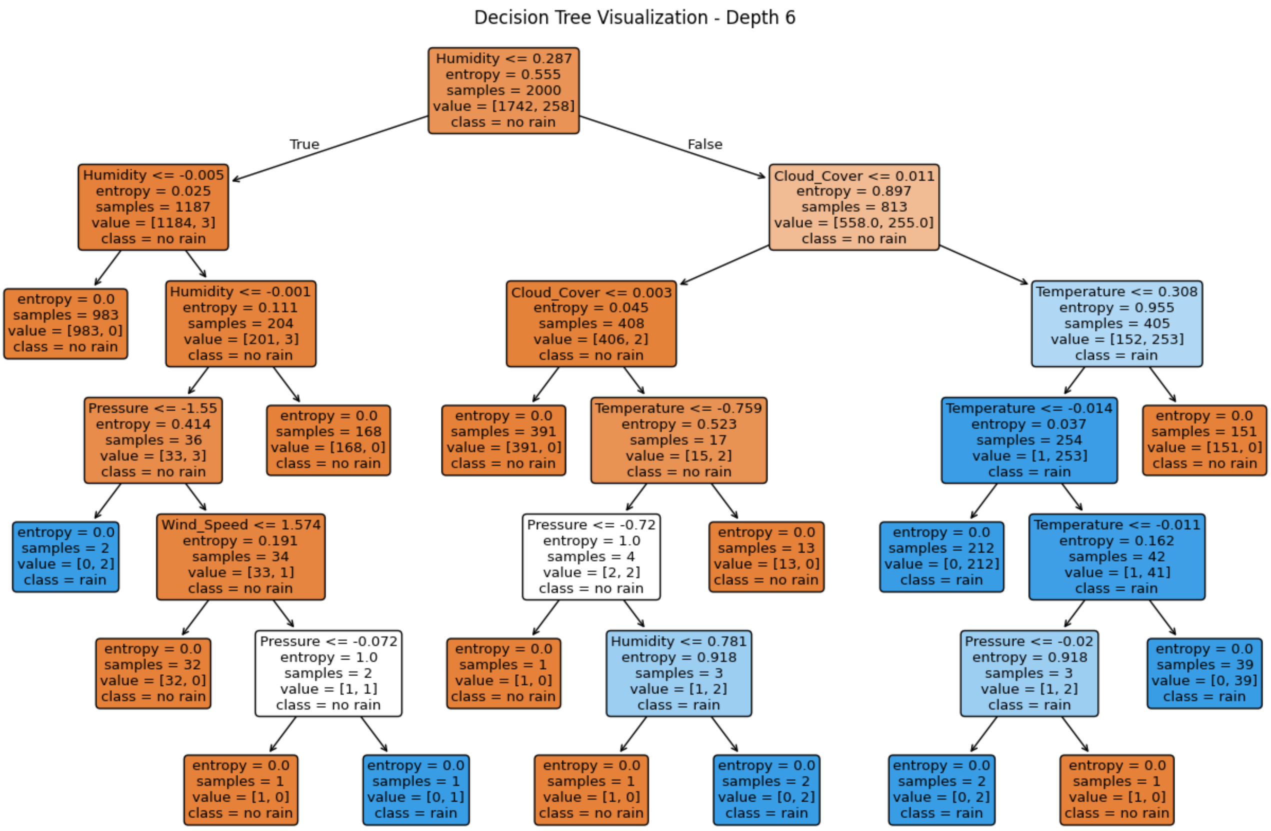

Decision Tree Visualization - Depth 3
Humidity <= 0.287
entropy = 0.555
samples = 2000
value = [1742, 258]
class = no rain
True
False
Humidity <= -0.005
entropy = 0.025
samples = 1187
value = [1184, 3]
class = no rain
Cloud_Cover <= 0.011
entropy = 0.897
samples = 813
value = [558.0, 255.0]
class = no rain
entropy = 0.0
samples = 983
value = [983, 0]
class = no rain
Humidity <= -0.001
entropy = 0.111
samples = 204
value = [201, 3]
class = no rain
Cloud_Cover <= 0.003
entropy = 0.045
samples = 408
value = [406, 2]
class = no rain
Temperature <= 0.308
entropy = 0.955
samples = 405
value = [152, 253]
class = rain
Pressure <= -1.55
entropy = 0.414
samples = 36
value = [33, 3]
class = no rain
entropy = 0.0
samples = 168
value = [168, 0]
class = no rain
entropy = 0.0
samples = 391
value = [391, 0]
class = no rain
Temperature <= -0.759
entropy = 0.523
samples = 17
value = [15, 2]
class = no rain
Temperature <= -0.014
entropy = 0.037
samples = 254
value = [1, 253]
class = rain
entropy = 0.0
samples = 151
value = [151, 0]
class = no rain
(...)
(...)
(...)
(...)
(...)
(...)

Decision Tree Visualization - Depth 4



Decision Tree Visualization - Depth 5





Splitting Logic

The decision tree model divides the dataset into subsets based on the features Humidity, Cloud Cover, and Temperature. At each node, the feature providing the highest ***information gain*** is selected for the split. The splitting process continues until the tree reaches a maximum depth of 6 .Below is a detailed explanation of the splitting logic:

Root Node

- Feature: *Humidity*
- Condition: Humidity ≤ 0.29
  - Samples: 100% of the dataset
  - Entropy: 0.55
- If the condition is met, the path follows the left branch; otherwise, it proceeds to the right branch.

Left Subtree (*Humidity ≤ 0.29*)

- Node 2:
  - Condition: Humidity ≤ -0.0
  - Prediction: 100% Class 0
  - Samples: 59.4% of the dataset

- Entropy: 0.03
- Child Nodes:
    - Left Child:
        - Class: Purely Class 0
        - Entropy: 0.0
    - Right Child:
        - Condition: Repeated split Humidity ≤ -0.0
        - Entropy: 0.11
        - Further Splits:
            - Left Leaf Node:
                - Entropy: 0.41
                - Class: 0
            - Right Leaf Node:
                - Entropy: 0.0 (Pure Node)
                - Class: 0

Right Subtree (*Humidity > 0.29*)

- Node 3:
    - Condition: Cloud Cover ≤ 0.01
    - Samples: 40.6% of the dataset
    - Entropy: 0.9
- Child Nodes:
    - Left Child:
        - Dominant Class: Class 0
        - Entropy: 0.04
    - Right Child:
        - Mixed Distribution:
            - Entropy: 0.95
            - Condition: Temperature ≤ 0.31
            - Samples: 20.2%
            - Further Splits:
                - Left Child:
                    - Dominant Class: Class 1
                    - Entropy: 0.04
                    - Samples: 12.7%
                - Right Child:
                    - Pure Class: Class 0
                    - Entropy: 0.0
                    - Samples: 7.6%

Summary

The decision tree effectively captures patterns in the weather forecast dataset using features such as Humidity, Cloud Cover, and Temperature. At each node, splits were chosen based on the highest *information gain*, resulting in the greatest reduction in entropy. This ensured optimal separation of classes. By minimizing entropy at every split, the model avoided randomness in predictions and focused on statistically significant patterns.

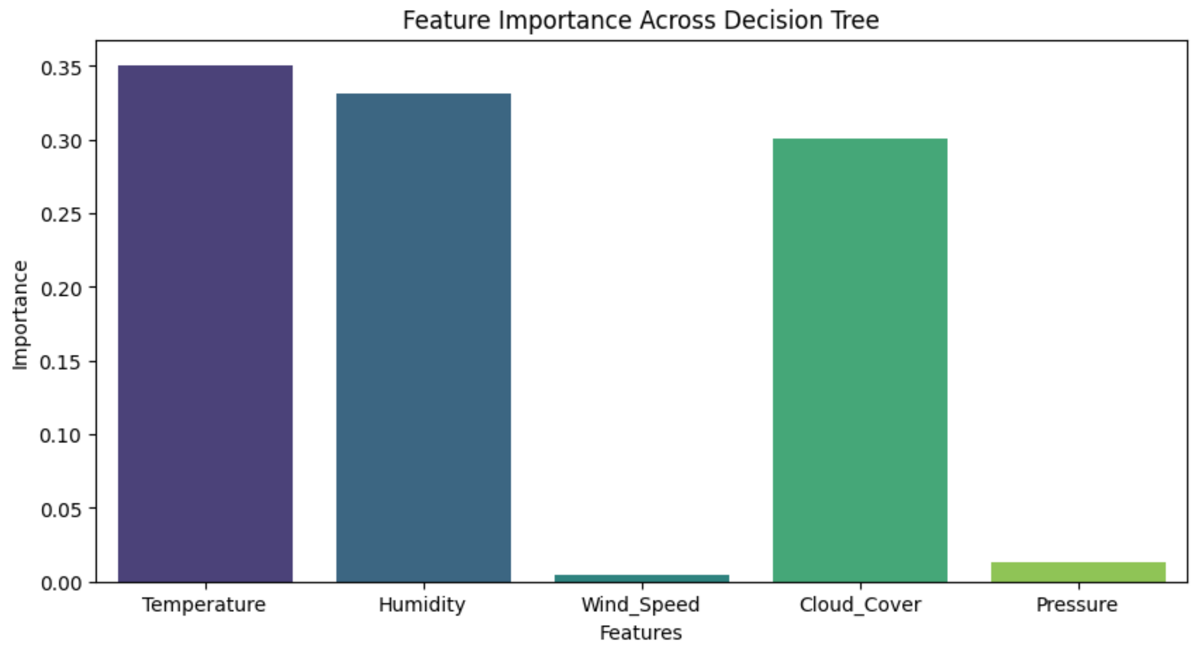
To Know how the sample be predicted

```python
def explain_prediction(clf, sample, feature_names):
    tree = clf.tree_

    print(f"Decision path for the sample: {sample}")
    print("Step-by-step explanation of the prediction:")

    node = 0
    while tree.children_left[node] != tree.children_right[node]:
        feature_index = tree.feature[node]
        threshold = tree.threshold[node]
        feature_name = feature_names[feature_index]

        # Make the decision
        if sample[feature_index] <= threshold:
            print(f"At node {node}, feature '{feature_name}' <= {threshold:.2f} (Sample value: {sample[feature_index]:.2f})")
            node = tree.children_left[node]  # Go to the left child
        else:
            print(f"At node {node}, feature '{feature_name}' > {threshold:.2f} (Sample value: {sample[feature_index]:.2f})")
            node = tree.children_right[node]  # Go to the right child

    predicted_class = np.argmax(tree.value[node])  # Majority class in leaf node
    print(f"Predicted class: {label_encoder.classes_[predicted_class]}")


sample = X.iloc[0].values
explain_prediction(clf, sample, X.columns)
print("_________________________________")
sample = X.iloc[912].values
explain_prediction(clf, sample, X.columns)
```

```
Decision path for the sample: [ 19.09611938  71.65172311  14.7823241   48.69925686 987.95476009]
Step-by-step explanation of the prediction:
At node 0, feature 'Humidity' > 0.29 (Sample value: 71.65)
At node 12, feature 'Cloud_Cover' > 0.01 (Sample value: 48.70)
At node 22, feature 'Temperature' > 0.31 (Sample value: 19.10)
Predicted class: no rain
_________________________________
Decision path for the sample: [  31.0730278    82.28552193   13.53403093   36.01892932 1046.05502965]
Step-by-step explanation of the prediction:
At node 0, feature 'Humidity' > 0.29 (Sample value: 82.29)
At node 12, feature 'Cloud_Cover' > 0.01 (Sample value: 36.02)
At node 22, feature 'Temperature' > 0.31 (Sample value: 31.07)
Predicted class: no rain
```

Feature Importance Across Decision Tree

```
import numpy as np
from sklearn.tree import export_text

tree_text = export_text(clf, feature_names=list(X_train.columns))
print(tree_text)

importances = clf.feature_importances_
features = X_train.columns
plt.figure(figsize=(10, 5))
sns.barplot(x=features, y=importances, palette="viridis")
plt.title("Feature Importance Across Decision Tree")
plt.xlabel("Features")
plt.ylabel("Importance")
plt.show()
```

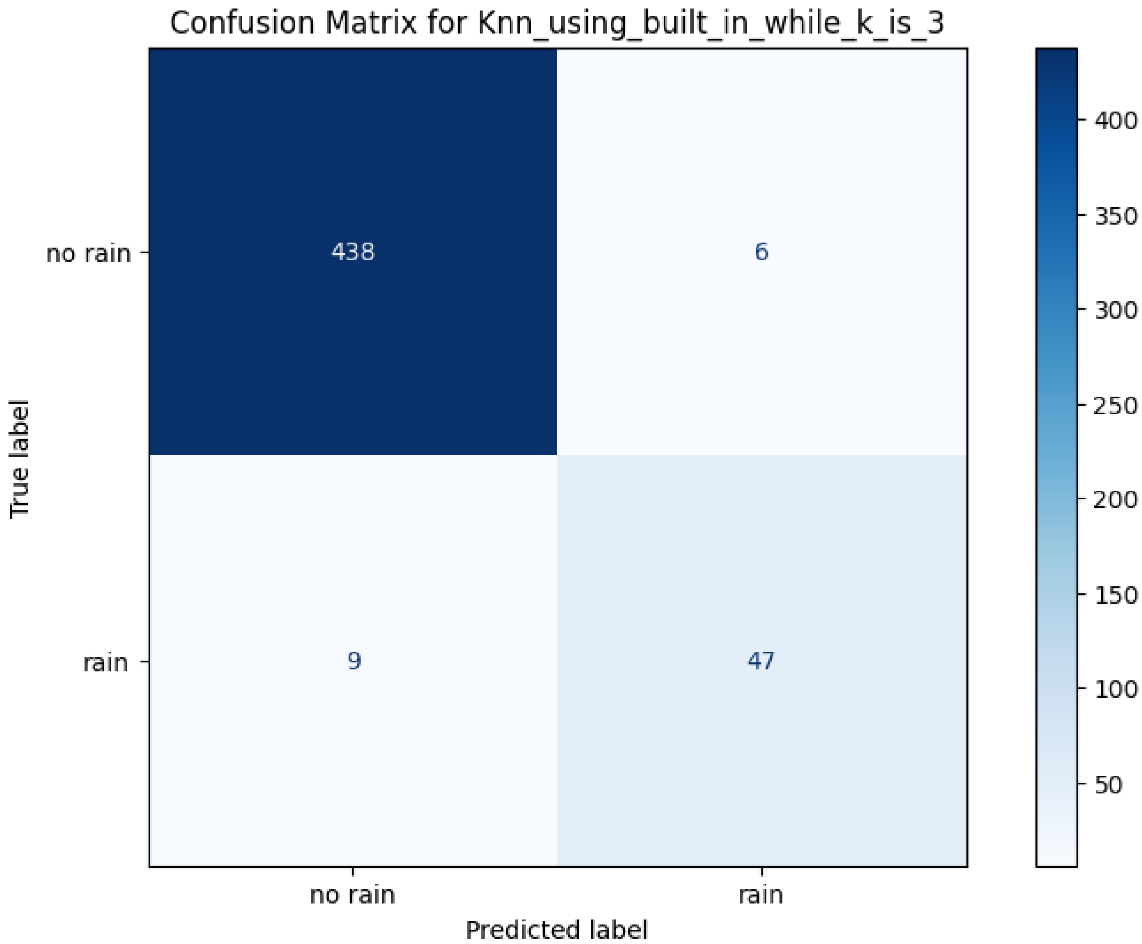


3.Performance Metrics Report

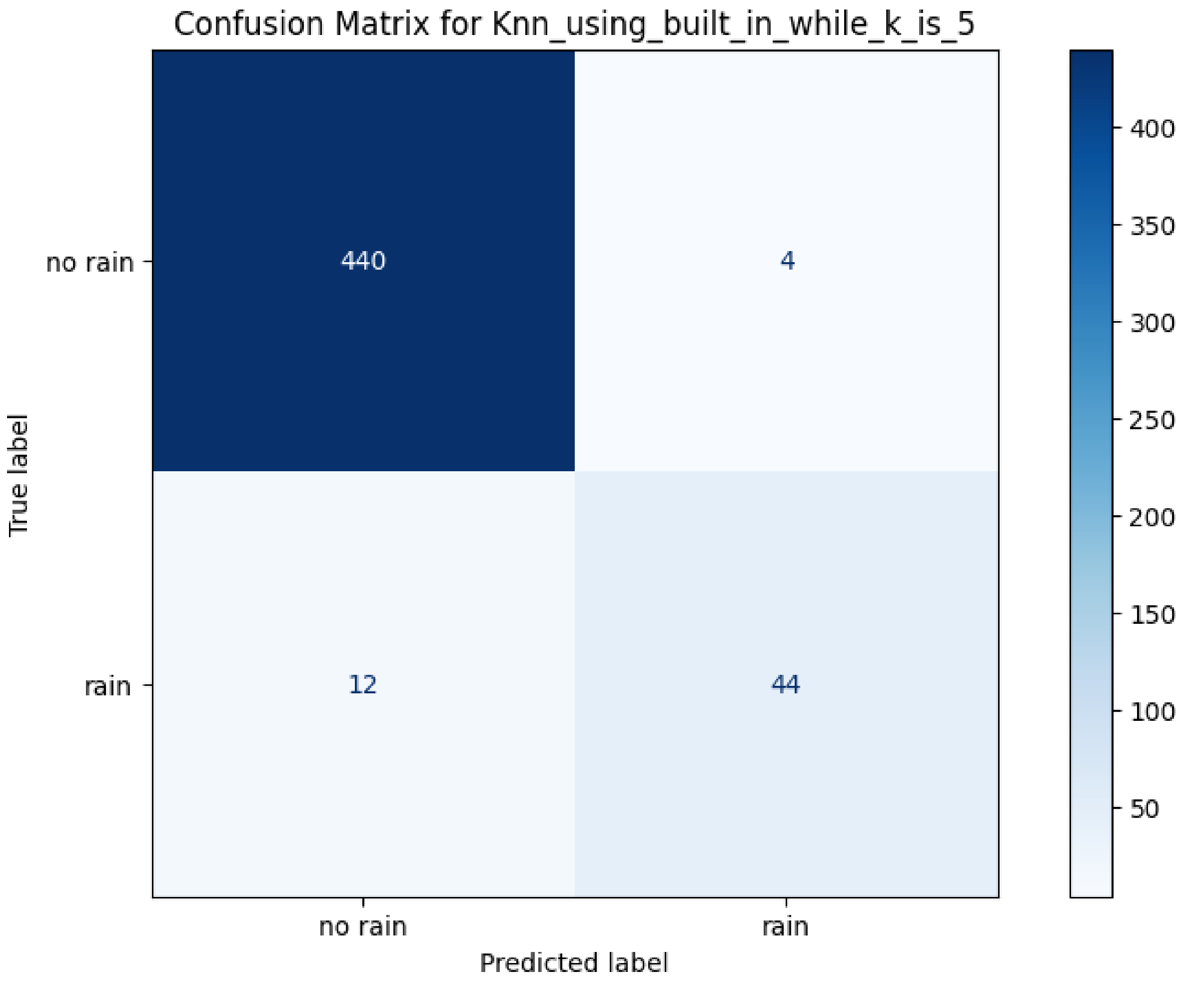
Knn using Sckit learn with different 5 k

```
for k in range(3, 12, 2):
    y_pred_knn = Knn(X_train, X_test, y_train, y_test, k)
    print_classification_report(f"Knn_using_built_in_while_k_is_{k}", y_test, y_pred_knn)
    plot_confusion_matrix(f"Knn_using_built_in_while_k_is_{k}", y_test, y_pred_knn)
```

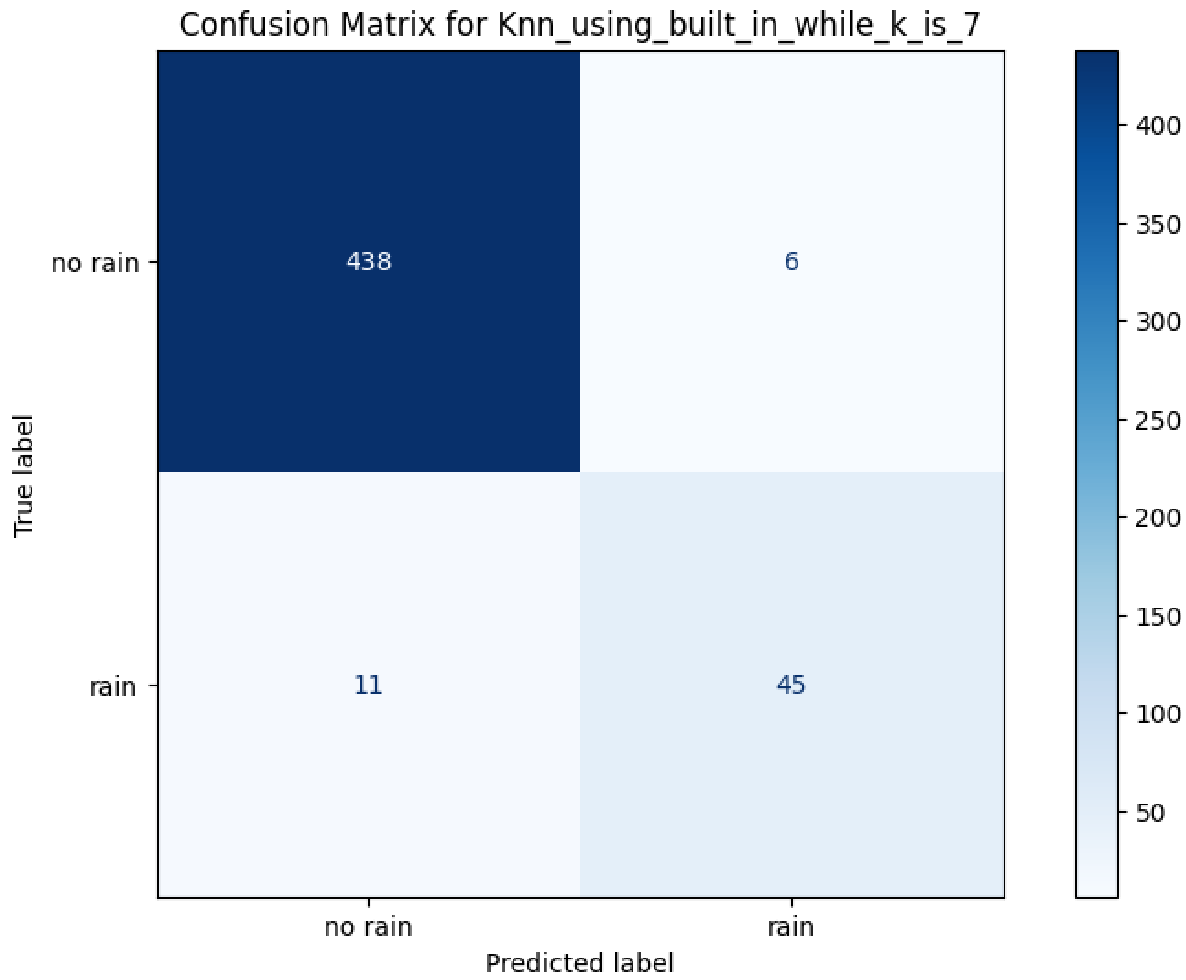
```
Accuracy of the KNN model on the test set: 0.9700
Classification Report for Knn_using_built_in_while_k_is_3:
              precision    recall  f1-score   support

           0       0.98      0.99      0.98       444
           1       0.89      0.84      0.86        56

    accuracy                           0.97       500
   macro avg       0.93      0.91      0.92       500
weighted avg       0.97      0.97      0.97       500
```



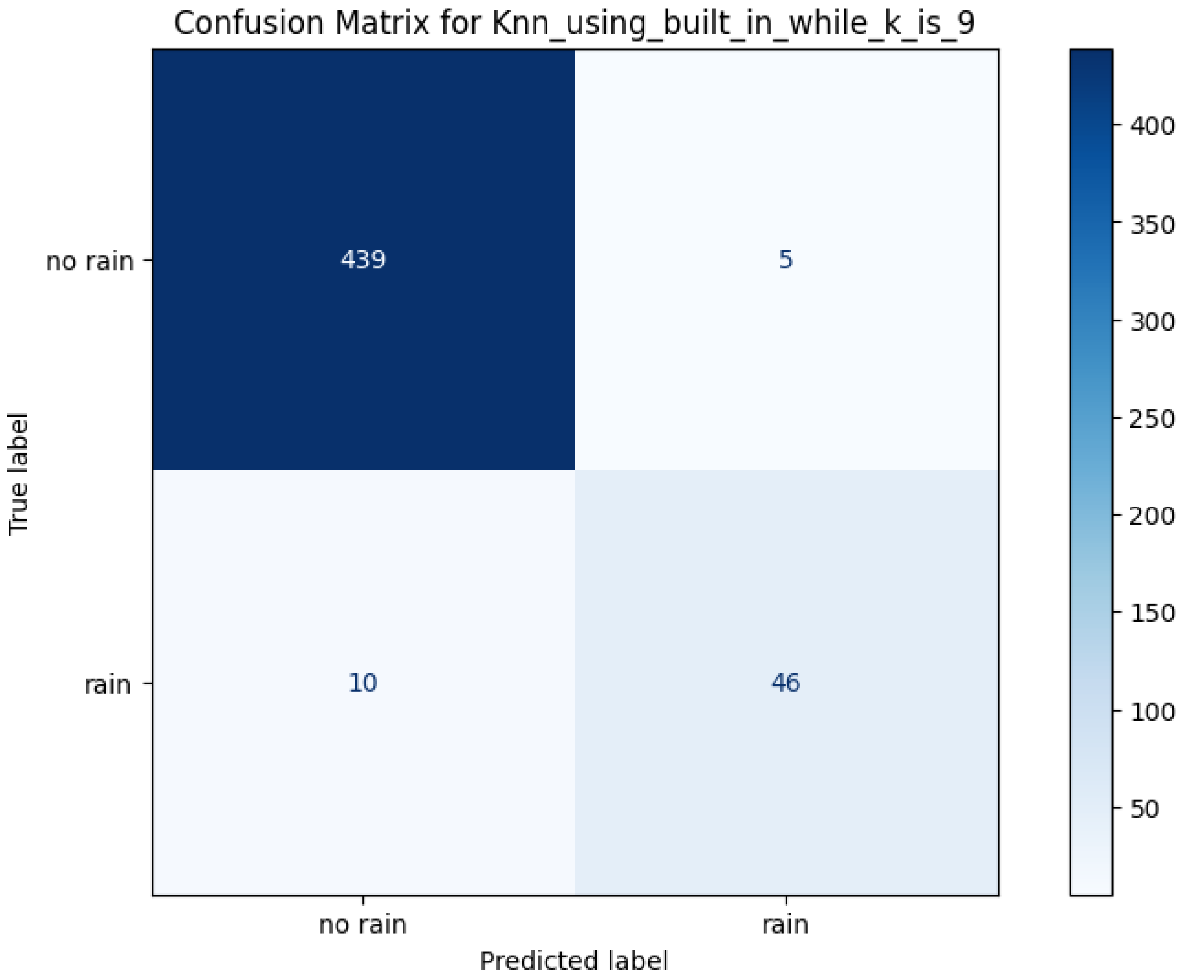
```
Accuracy of the KNN model on the test set: 0.9680
Classification Report for Knn_using_built_in_while_k_is_5:
              precision    recall  f1-score   support

           0       0.97      0.99      0.98       444
           1       0.92      0.79      0.85        56

    accuracy                           0.97       500
   macro avg       0.95      0.89      0.91       500
weighted avg       0.97      0.97      0.97       500
```



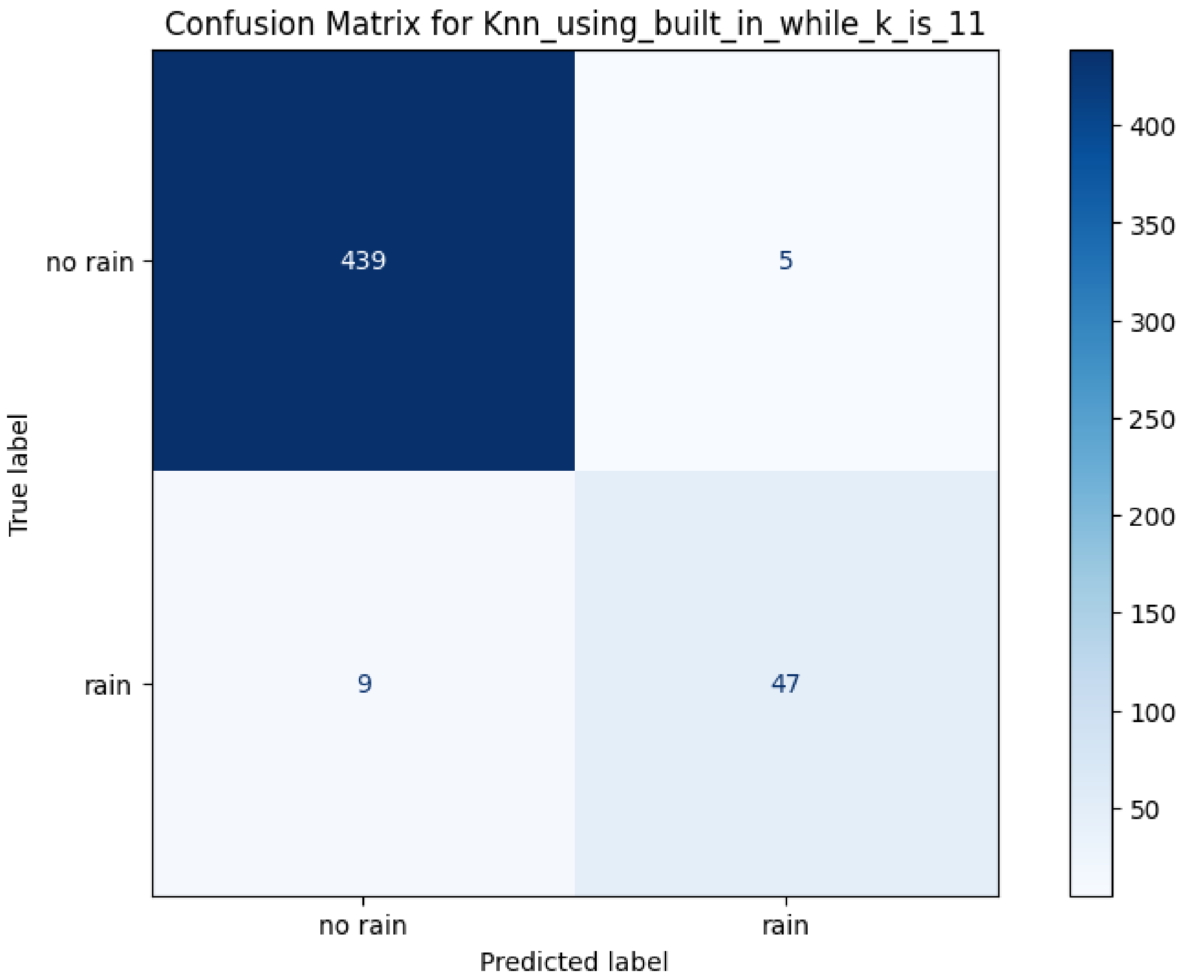
```
Accuracy of the KNN model on the test set: 0.9660
Classification Report for Knn_using_built_in_while_k_is_7:
              precision    recall  f1-score   support

           0       0.98      0.99      0.98       444
           1       0.88      0.80      0.84        56

    accuracy                           0.97       500
   macro avg       0.93      0.90      0.91       500
weighted avg       0.97      0.97      0.97       500
```



```
Accuracy of the KNN model on the test set: 0.9700
Classification Report for Knn_using_built_in_while_k_is_9:
              precision    recall  f1-score   support

           0       0.98      0.99      0.98       444
           1       0.90      0.82      0.86        56

    accuracy                           0.97       500
   macro avg       0.94      0.91      0.92       500
weighted avg       0.97      0.97      0.97       500
```



```
Accuracy of the KNN model on the test set: 0.9720
Classification Report for Knn_using_built_in_while_k_is_11:
              precision    recall  f1-score   support

           0       0.98      0.99      0.98       444
           1       0.90      0.84      0.87        56

    accuracy                           0.97       500
   macro avg       0.94      0.91      0.93       500
weighted avg       0.97      0.97      0.97       500
```



Summary Table:

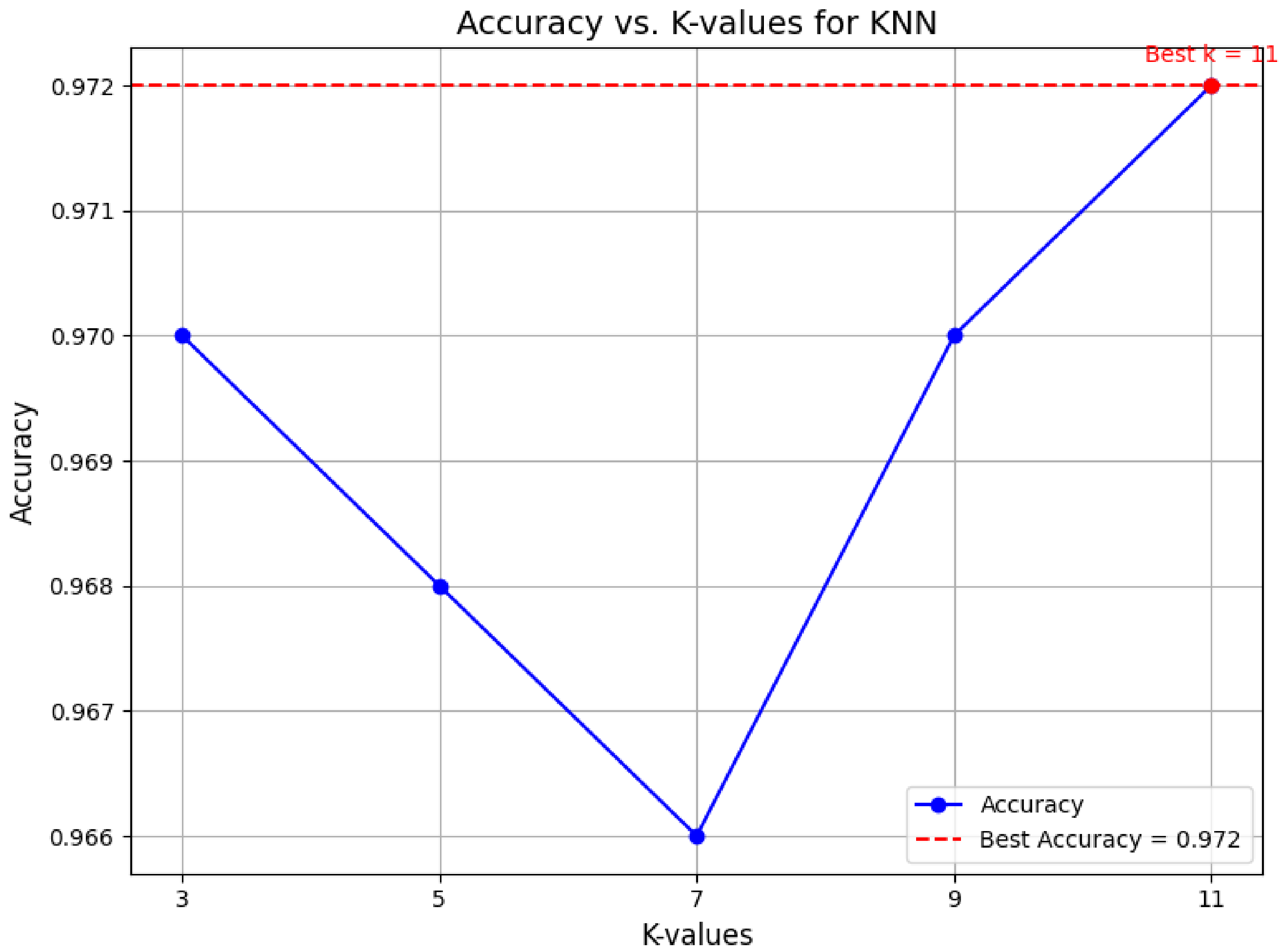
k Value	Accuracy	Precision (No Rain)	Precision (Rain)	Recall (No Rain)	Recall (Rain)	F1-Score (Rain)	False Positives (No Rain)	False Negatives (Rain)
3	0.9700	0.97	0.88	0.99	0.84	0.86	4	9
5	0.9680	0.97	0.92	0.99	0.79	0.83	4	12
7	0.9660	0.97	0.90	0.99	0.80	0.84	5	11
9	0.9700	0.97	0.90	0.99	0.82	0.86	5	10
11	0.9720	0.97	0.90	0.99	0.84	0.87	7	9

Key Insights:

- **Best accuracy**: (k = 11) with **0.9720**.
- **Best precision (Rain)**: (k = 5) with **0.92**.
- **Best recall (Rain)**: (k = 11) with **0.84**.
- **Best F1-score (Rain)**: (k = 11) with **0.87**.
- **Best false negatives**: (k = 11) with **9**.
- **Best false positives (No Rain)**: (k = 5) with **4**.

Accuracy Comparison Plot

```python
import matplotlib.pyplot as plt

k_values = [3, 5, 7, 9, 11]
accuracies = [0.9700, 0.9680, 0.9660, 0.9700, 0.9720]

best_accuracy = max(accuracies)
best_k = k_values[accuracies.index(best_accuracy)]

plt.figure(figsize=(8, 6))
plt.plot(k_values, accuracies, marker='o', linestyle='-', color='b', label="Accuracy")
plt.axhline(y=best_accuracy, color='r', linestyle='--', label=f"Best Accuracy = {best_accuracy:.3f}")
plt.scatter([best_k], [best_accuracy], color='red', zorder=5)

plt.title("Accuracy vs. K-values for KNN", fontsize=14)
plt.xlabel("K-values", fontsize=12)
plt.ylabel("Accuracy", fontsize=12)
plt.xticks(k_values)
plt.legend(loc="lower right")
plt.grid(True)

plt.annotate(f"Best k = {best_k}", (best_k, best_accuracy),
             textcoords="offset points", xytext=(0, 10), ha='center', fontsize=10, color='red')

plt.tight_layout()
plt.show()
```

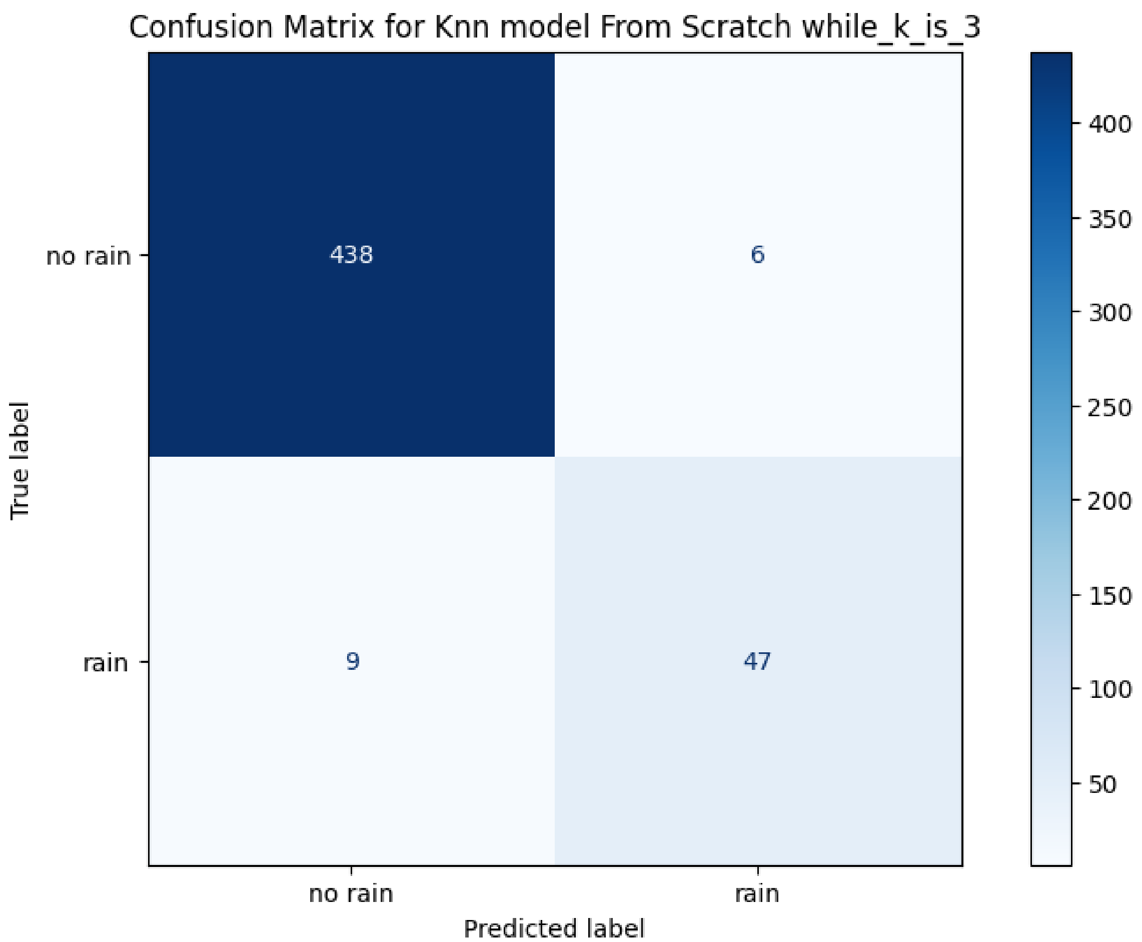


Accuracy vs. K-values for KNN
Best k = 11
0.972
0.971
0.970
0.969
0.968
0.967
0.966
Accuracy
3
5
7
9
11
K-values
Accuracy
Best Accuracy = 0.972

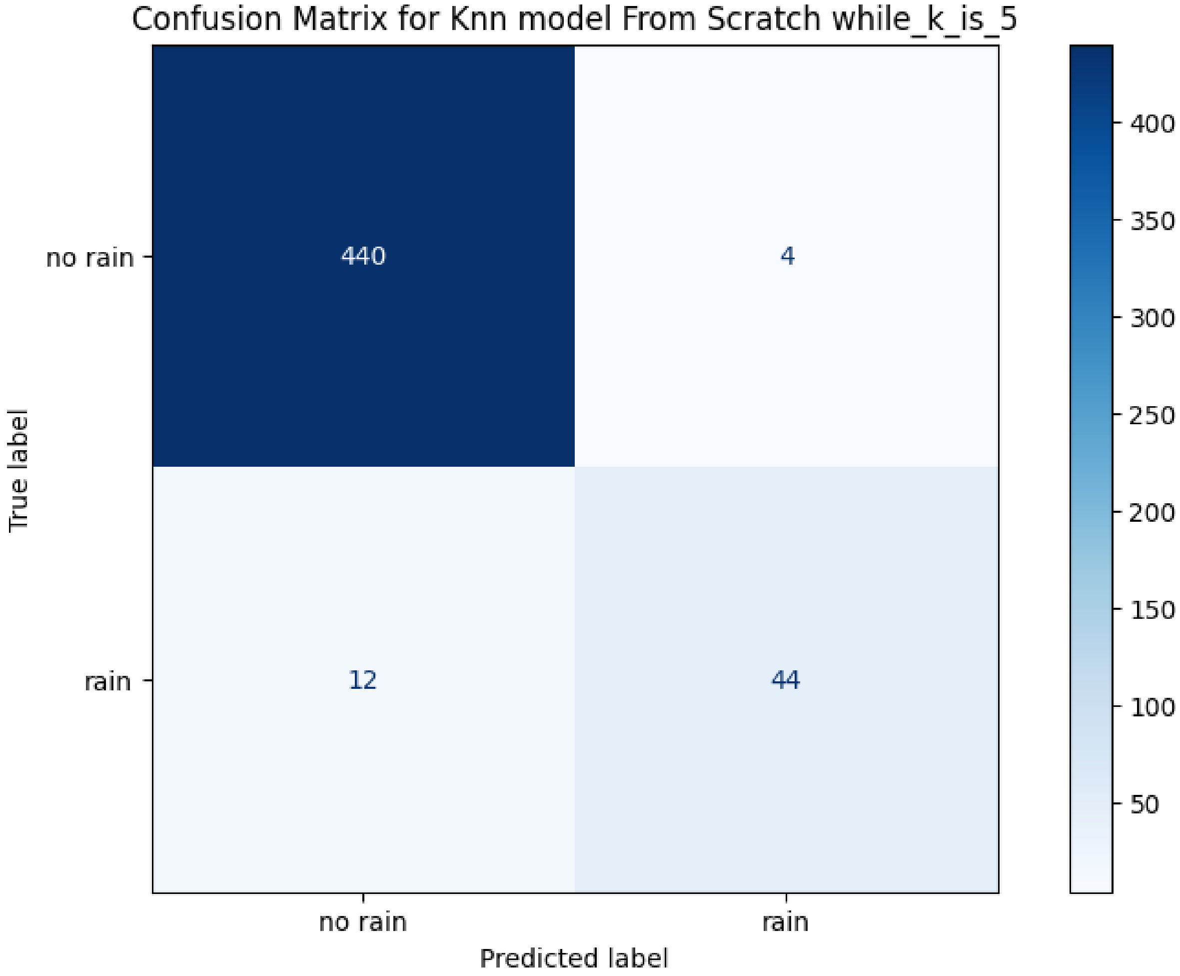
Knn from Scratch with different 5 k

```python
for k in range(3,12,2):
    knn_model = initialize_knn(k)
    fit_knn(knn_model, X_train, np.array(y_train).ravel())
    y_pred_knn_from_Scratch = predict_knn(knn_model, X_test)
    accuracy = accuracy_score(y_test, y_pred_knn_from_Scratch)
    print(f'Accuracy of the KNN model from scratch on the test set while_k_is_{k}: {accuracy:.4f}')
    print_classification_report(f"Knn model From Scratch while_k_is_{k}",y_test,y_pred_knn_from_Scratch)
    plot_confusion_matrix(f"Knn model From Scratch while_k_is_{k}",y_test,y_pred_knn_from_Scratch)
```

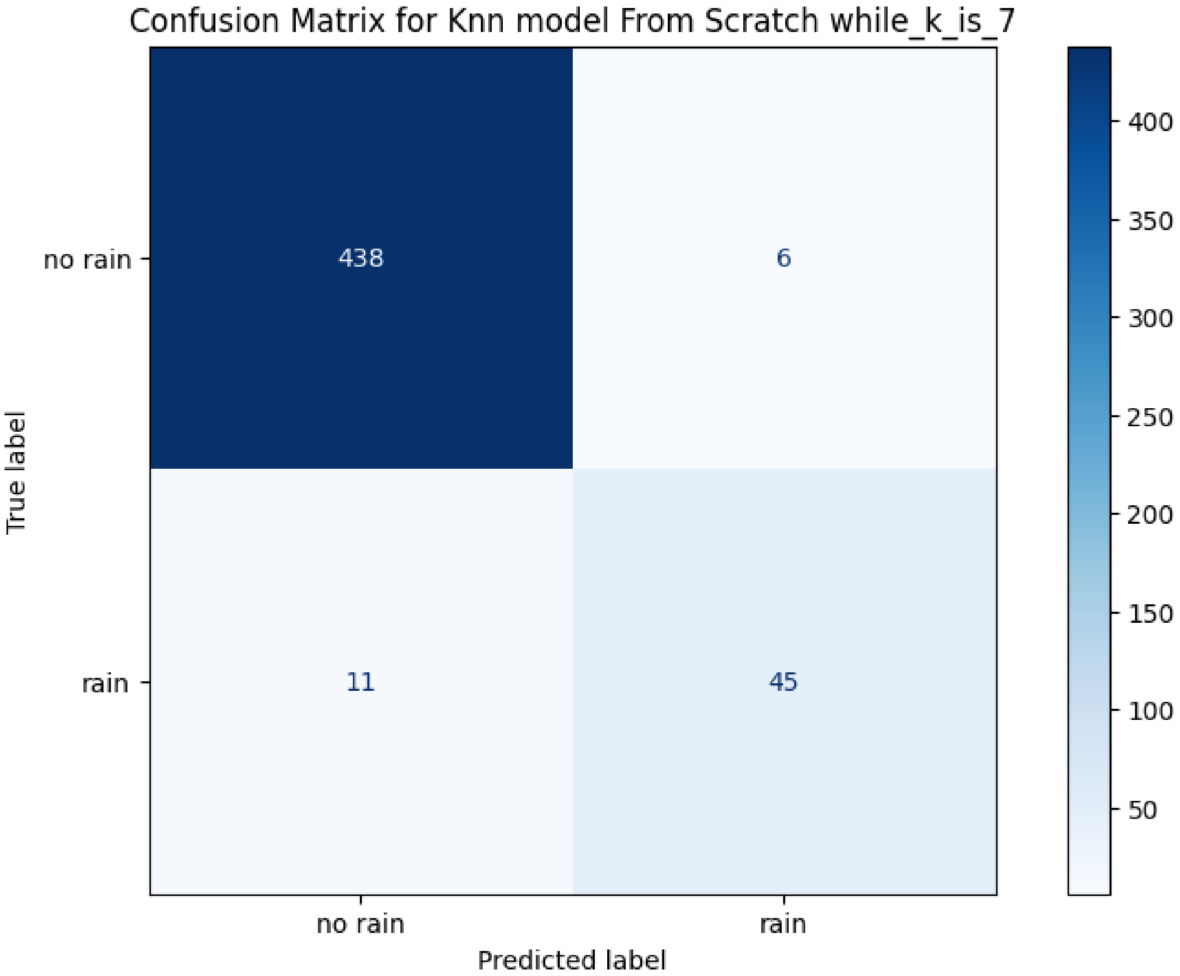
```
Accuracy of the KNN model from scratch on the test set while_k_is_3: 0.9700
Classification Report for Knn model From Scratch while_k_is_3:
              precision    recall  f1-score   support

           0       0.98      0.99      0.98       444
           1       0.89      0.84      0.86        56

    accuracy                           0.97       500
   macro avg       0.93      0.91      0.92       500
weighted avg       0.97      0.97      0.97       500
```



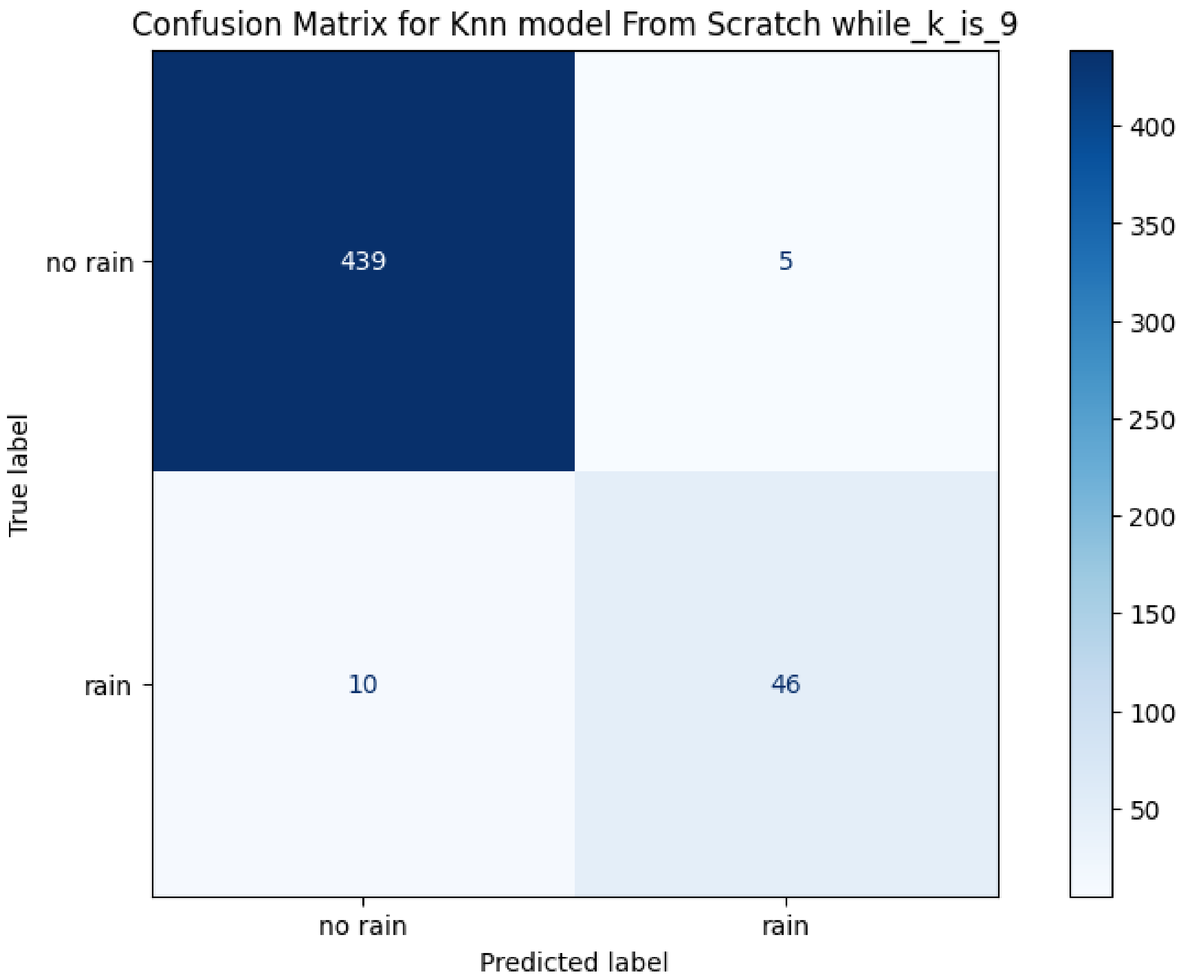
```
Accuracy of the KNN model from scratch on the test set while_k_is_5: 0.9680
Classification Report for Knn model From Scratch while_k_is_5:
              precision    recall  f1-score   support

           0       0.97      0.99      0.98       444
           1       0.92      0.79      0.85        56

    accuracy                           0.97       500
   macro avg       0.95      0.89      0.91       500
weighted avg       0.97      0.97      0.97       500
```



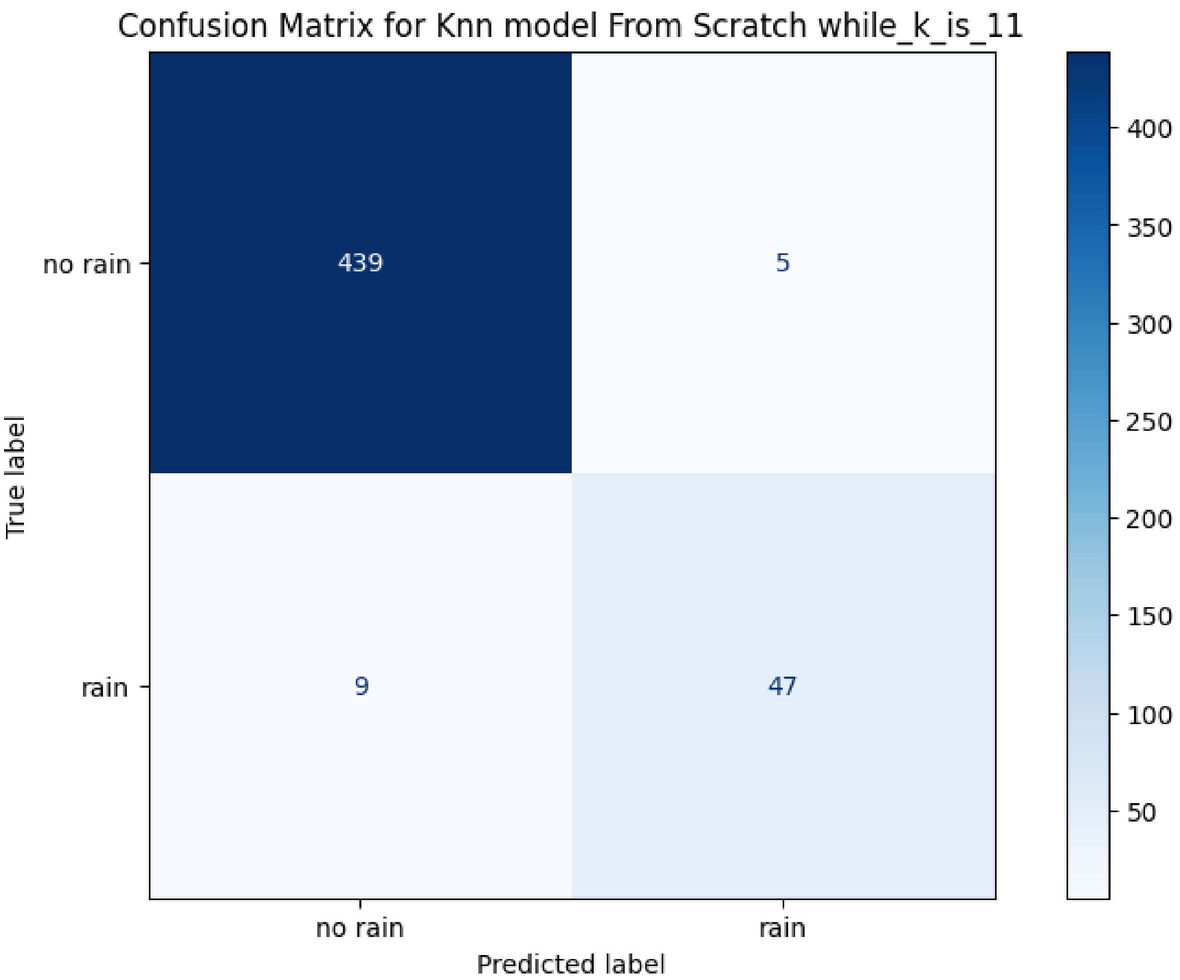
```
Accuracy of the KNN model from scratch on the test set while_k_is_7: 0.9660
Classification Report for Knn model From Scratch while_k_is_7:
              precision    recall  f1-score   support

           0       0.98      0.99      0.98       444
           1       0.88      0.80      0.84        56

    accuracy                           0.97       500
   macro avg       0.93      0.90      0.91       500
weighted avg       0.97      0.97      0.97       500
```



```
Accuracy of the KNN model from scratch on the test set while_k_is_9: 0.9700
Classification Report for Knn model From Scratch while_k_is_9:
              precision    recall  f1-score   support

           0       0.98      0.99      0.98       444
           1       0.90      0.82      0.86        56

    accuracy                           0.97       500
   macro avg       0.94      0.91      0.92       500
weighted avg       0.97      0.97      0.97       500
```



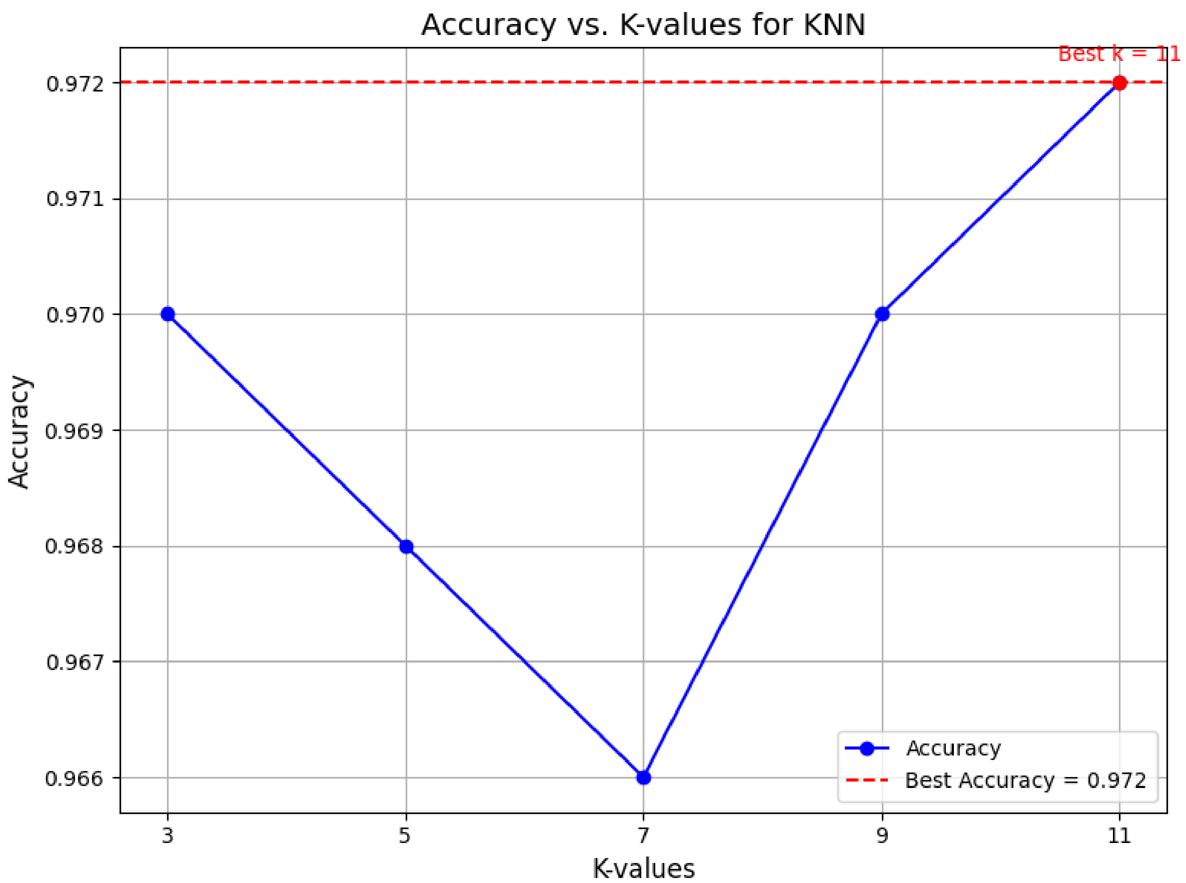
```
Accuracy of the KNN model from scratch on the test set while_k_is_11: 0.9720
Classification Report for Knn model From Scratch while_k_is_11:
              precision    recall  f1-score   support

           0       0.98      0.99      0.98       444
           1       0.90      0.84      0.87        56

    accuracy                           0.97       500
   macro avg       0.94      0.91      0.93       500
weighted avg       0.97      0.97      0.97       500
```



k Value	Accuracy	Precision (No Rain)	Precision (Rain)	Recall (No Rain)	Recall (Rain)	F1-Score (Rain)	False Positives (No Rain)	False Negatives (Rain)
3	0.9700	0.97	0.89	0.99	0.84	0.86	6	9
5	0.9680	0.97	0.92	0.99	0.79	0.83	4	12
7	0.9660	0.97	0.88	0.99	0.80	0.84	6	11
9	0.9700	0.97	0.90	0.99	0.82	0.86	5	10
11	0.9720	0.97	0.90	0.99	0.84	0.87	5	9

Key Insights:

- **Best accuracy**: (k=11) with **0.9720**.
- **Best precision (Rain)**: (k=5) with **0.92**.
- **Best recall (Rain)**: (k=11) with **0.84**.
- **Best F1-score (Rain)**: (k=11) with **0.87**.
- **Best false negatives**: (k=11) with **9**.
- **Best false positives (No Rain)**: (k=5) with **4**.



Comparison Between KNN (Built-in) and KNN (Implemented From Scratch)

---

1. Accuracy:

- Both methods report the **same accuracy** for each k-value.
    - The highest accuracy is consistently observed at **k=11** with a value of **0.9720**.

---

2. Precision (Rain):

- The **precision for the "Rain" class** is identical in both results:
    - **k=5** achieves the highest precision at **0.92**.

---

3. Recall (Rain):

- Both methods yield the **same recall for the "Rain" class**:
    - The highest recall is observed at **k=11** with a value of **0.84**.

---

4. F1-Score (Rain):

- The **F1-score for the "Rain" class** is identical across both methods:
    - The highest F1-score is achieved at **k=11** with a value of **0.87**.

---

5. False Positives (No Rain):

- The number of **false positives** for the "No Rain" class is the same in both results:
    - **k=5** has the lowest number of false positives, at **4**.

---

6. False Negatives (Rain):

- Both methods report the **same false negatives for the "Rain" class**:
    - **k=11** has the lowest number of false negatives, at **9**.

Key Insight:

There are **no differences** in the analysis, metrics, or insights between the built-in and scratch-implemented KNN. The values and their interpretation are **identical** across both approaches.