Faculty of Computer
and Artificial Intelligence

Cairo University

Assignment 2

Ahmed Yehia Abdelmoaty	20210049
Rana Essam Ibrahim Ibrahim	20210133
Mohannad Hisham Altahawy	20210413
Nour El din Ahmed Hussein Ismail	20210430
Merna Islam Mohamed	20210500

Task1: Preprocessing

1. Identify missing data

```python
print("null values count for each feature")
print(data.isnull().sum())
```

```
null values count for each feature
Temperature    25
Humidity       40
Wind_Speed     32
Cloud_Cover    33
Pressure       27
Rain            0
```

2. Preprocessing by filling missing values:

```
data = handel_missing_values_technique1(data)
print ("After handling missing values")
data.info()
```

[41] ✓ 0.0s Python

```
Before handling missing values
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2500 entries, 0 to 2499
Data columns (total 6 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Temperature  2475 non-null   float64
 1   Humidity     2460 non-null   float64
 2   Wind_Speed   2468 non-null   float64
 3   Cloud_Cover  2467 non-null   float64
 4   Pressure     2473 non-null   float64
 5   Rain         2500 non-null   object
dtypes: float64(5), object(1)
memory usage: 117.3+ KB
After handling missing values
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2500 entries, 0 to 2499
Data columns (total 6 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Temperature  2500 non-null   float64
 1   Humidity     2500 non-null   float64
 2   Wind_Speed   2500 non-null   float64
 3   Cloud_Cover  2500 non-null   float64
 4   Pressure     2500 non-null   float64
 5   Rain         2500 non-null   object
dtypes: float64(5), object(1)
memory usage: 117.3+ KB
```

3. Checking Data scale after handling missing data using filling missing values tech.

Not all the features have the same scale

```
data.describe()
```

[67] ✓ 0.0s Python

	Temperature	Humidity	Wind_Speed	Cloud_Cover	Pressure
count	2500.000000	2500.000000	2500.000000	2500.000000	2500.000000
mean	22.573777	64.366909	9.911826	49.808770	1014.409327
std	7.295628	19.813325	5.743575	28.869772	20.072933
min	10.001842	30.005071	0.009819	0.015038	980.014486
25%	16.417898	47.493987	4.829795	24.817296	997.190281
50%	22.573777	64.366909	9.911826	49.808770	1014.095390
75%	28.934369	81.445049	14.889660	74.989410	1031.606187
max	34.995214	99.997481	19.999132	99.997795	1049.985593

2 . Preprocessing by dropping nulls:

```
Before handling missing values
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2500 entries, 0 to 2499
Data columns (total 6 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Temperature  2475 non-null   float64
 1   Humidity     2460 non-null   float64
 2   Wind_Speed   2468 non-null   float64
 3   Cloud_Cover  2467 non-null   float64
 4   Pressure     2473 non-null   float64
 5   Rain         2500 non-null   object
dtypes: float64(5), object(1)
memory usage: 117.3+ KB
After handling missing values
<class 'pandas.core.frame.DataFrame'>
Index: 2347 entries, 0 to 2499
Data columns (total 6 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   Temperature  2347 non-null   float64
 1   Humidity     2347 non-null   float64
 2   Wind_Speed   2347 non-null   float64
 3   Cloud_Cover  2347 non-null   float64
 4   Pressure     2347 non-null   float64
 5   Rain         2347 non-null   object
dtypes: float64(5), object(1)
memory usage: 128.4+ KB
```

3. Checking the Data scale after handling missing data using dropping nulls

Not all the features have the same scale

```
data.describe()
```

[80] ✓ 0.0s

	Temperature	Humidity	Wind_Speed	Cloud_Cover	Pressure
count	2347.000000	2347.000000	2347.000000	2347.000000	2347.000000
mean	22.586674	64.313486	9.936976	49.826460	1014.362428
std	7.325814	19.969574	5.778717	29.163519	20.157864
min	10.001842	30.005071	0.009819	0.015038	980.014486
25%	16.423651	47.124078	4.786505	24.119752	997.010203
50%	22.533110	64.044753	9.999957	49.735062	1013.591009
75%	28.967040	81.607683	14.955263	75.496921	1031.683526
max	34.995214	99.997481	19.999132	99.997795	1049.985593

4. Splitting and Scaling

```
# 1.2 Scaling and Encoding
scaler = StandardScaler()
labelEncoder = LabelEncoder()

def scale_data(data, isTestData):
    if not isTestData:
        data = scaler.fit_transform(data)
    else:
        data = scaler.transform(data)
    return pandas.DataFrame(data, columns=numeric_features)

def encode_target(data):
    data= labelEncoder.fit_transform(data[target])
    return pandas.DataFrame(data, columns=[target])

x = data[numeric_features]
y = encode_target(data)[target]


x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=0)

x_train = scale_data(x_train, False)
x_test = scale_data(x_test, True)
```

Task 2:

KNN using Scikit by filling missing values:

```
from sklearn.neighbors import KNeighborsClassifier

sklearn_knn = KNeighborsClassifier(n_neighbors=9)
start_time = time.time()
sklearn_knn.fit(x_train, y_train)
predictions = sklearn_knn.predict(x_test)
end_time = time.time()

Metrics(y_test, predictions, end_time -start_time).show()
```

[225] ✓ 0.0s

```
time:  0.022060394287109375
Accuracy:  0.9744680851063829
Precision:  0.9361702127659575
Recall:  0.8301886792452831
=====================================
```

KNN from Scratch by filling missing values:

```python
class KNN:
    def __init__(self, k):
        self.k = k

    def fit(self, X, y):
        self.X = X
        self.y = y
    def get_nearest_neighbors(self, crnt_X):
        distances = np.sqrt(((self.X - crnt_X) ** 2).sum(axis=1)) # Euclidean distance
        nearest_neighbors_indices = distances.argsort()[:self.k]
        return nearest_neighbors_indices

    def predict(self, X):
        predictions = []
        for crnt_X in X:
            nearest_neighbors_indices = self.get_nearest_neighbors(crnt_X)
            cnt = np.sum(self.y[nearest_neighbors_indices])
            if cnt > self.k/2:
                prediction = 1
            else:
                prediction = 0
            predictions.append(prediction)
        return predictions
```

```python
# Scratch KNN
k_values = [1, 3, 5, 9, 100]

for k in k_values:
    knn = KNN(k)
    start_time = time.time()
    knn.fit(x_train.values, y_train.values)
    predictions = knn.predict(x_test.values)
    end_time = time.time()
    print("K = ", k)
    Metrics(y_test, predictions, end_time - start_time).show(False)
```

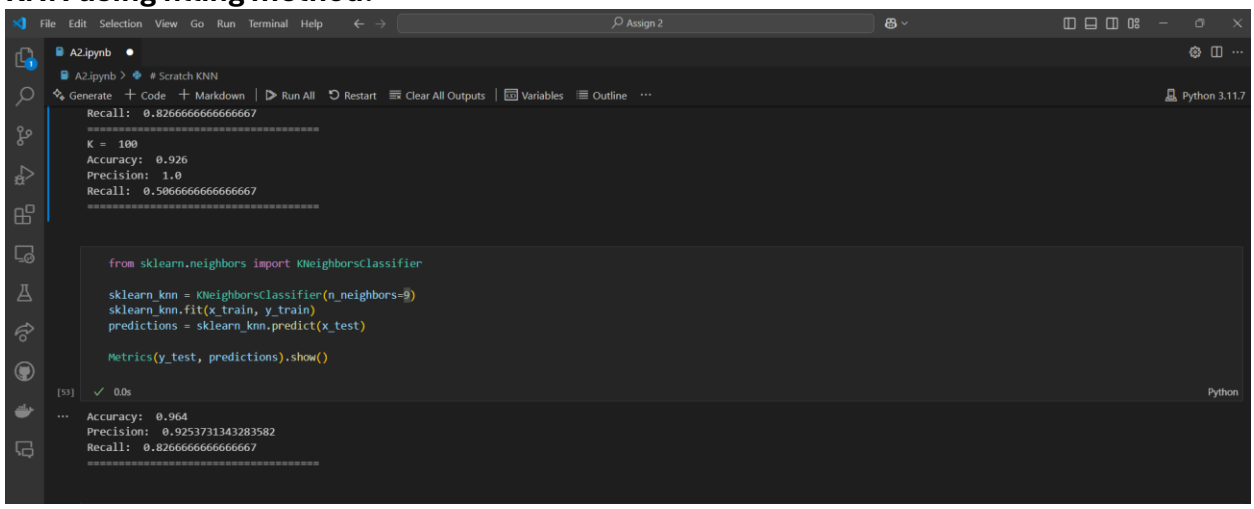
✓ 0.1s

```
✓ 0.1s
K =  1
time:  0.0322723388671875
Accuracy:  0.9574468085106383
Precision:  0.8235294117647058
Recall:  0.7924528301886793
K =  3
time:  0.028674602508544922
Accuracy:  0.9702127659574468
Precision:  0.9148936170212766
Recall:  0.8113207547169812
K =  5
time:  0.027711153030395508
Accuracy:  0.9744680851063829
Precision:  0.9555555555555556
Recall:  0.8113207547169812
K =  9
time:  0.02788090705871582
Accuracy:  0.9744680851063829
Precision:  0.9361702127659575
Recall:  0.8301886792452831
K =  100
time:  0.027899742126464844
Accuracy:  0.9531914893617022
Precision:  1.0
Recall:  0.5849056603773585
```

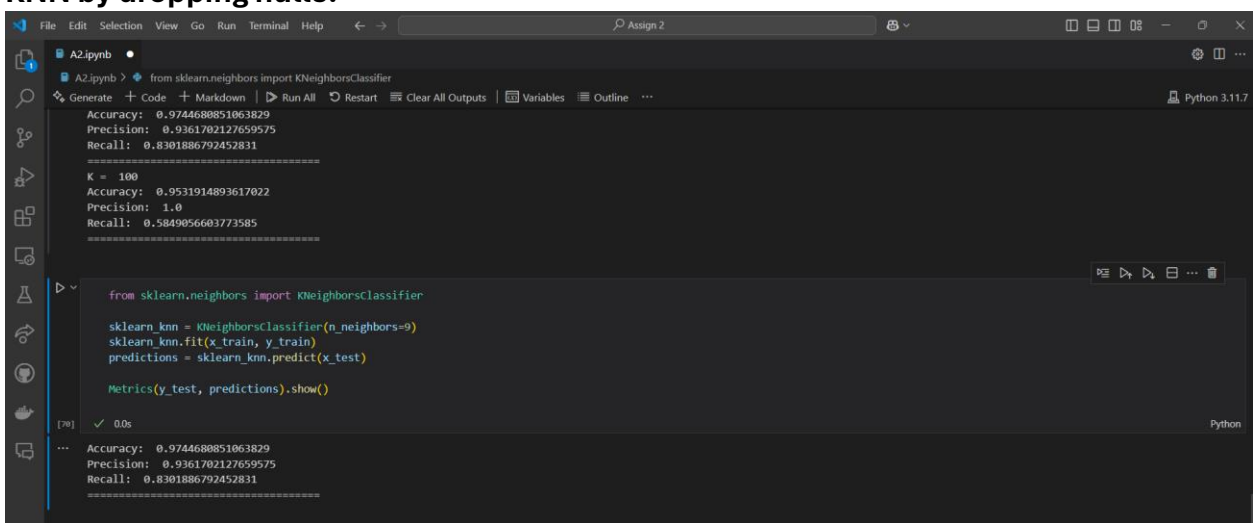
1- **Accuracy**: Both implementations achieve similar accuracy for K = 9, with no significant difference observed.
2- **Precision**: Custom implementation and scikit-learn yield the same precision for K = 9 (0.925), demonstrating comparable performance.
3- **Recall**: Recall values for K = 9 are also identical (0.826)
4- **Time**: Scikit-learn' s KNN is faster than the custom implementation for K = 9, with a difference of 0.0092 seconds

Task 3.1, Detailed report evaluating the performance of Scikit-Learn:

KNN using filling method:



KNN by dropping nulls:



Accuracy: Dropping null values slightly improved accuracy by ~1%.
Precision: Both methods yielded similar precision, with a marginal advantage (~1%) for the dropping nulls approach.
Recall: Recall is slightly higher when nulls are dropped, but the difference is minimal (~0.33%).
Dropping nulls: Yields slightly better metrics across all measures. However, this may not generalize well in practice as it reduces the training data size and potentially biases the dataset. <u>If the dataset is large and the percentage of null values is minimal</u>, *dropping nulls* might be more appropriate.
Filling with averages: Maintains all data, which could be crucial for smaller datasets or datasets with missing values in key variables. Results in marginally lower performance but may generalize better in real-world scenarios. <u>If the dataset size is small or missing values are significant</u>, prefer *filling with averages* to avoid losing information.

Naïve Bayes using filling method:

```
A2.ipynb > from sklearn.neighbors import KNeighborsClassifier
[53] 0.0s
Accuracy:  0.964
Precision:  0.9253731343283582
Recall:  0.8266666666666667
==================================

from sklearn.naive_bayes import GaussianNB

naive_bayes =  GaussianNB()
naive_bayes.fit(x_train, y_train)
predictions = naive_bayes.predict(x_test)

Metrics(y_test, predictions).show()
[54] 0.0s
Accuracy:  0.952
Precision:  1.0
Recall:  0.68
==================================
```

Naïve Bayes by dropping nulls:

```
A2.ipynb > from sklearn.tree import plot_tree

from sklearn.naive_bayes import GaussianNB

naive_bayes =  GaussianNB()
naive_bayes.fit(x_train, y_train)
predictions = naive_bayes.predict(x_test)

Metrics(y_test, predictions).show()
[71] 0.0s
Accuracy:  0.9702127659574468
Precision:  1.0
Recall:  0.7358490566037735
==================================
```

Accuracy: Dropping nulls achieved higher accuracy (97.02%) compared to filling with averages (95.2%).

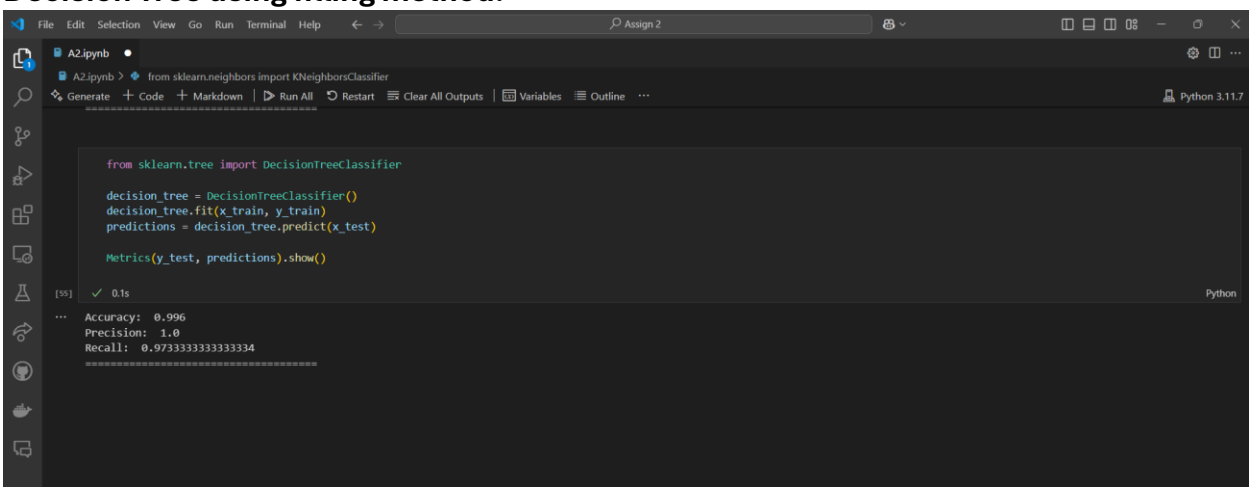
Precision: Precision is perfect (1.0 or 100%) in both cases, indicating that the classifier did not produce any false positives (all predicted positives were correct).

Recall: Recall is significantly lower than precision in both cases, with better performance when null values are dropped (73.58% vs. 68%).

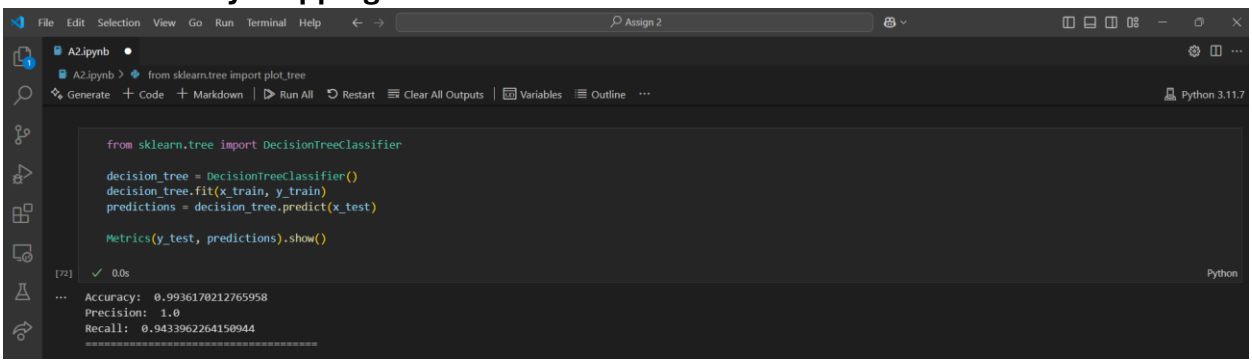
Filling Missing Values: Lower accuracy and recall compared to dropping nulls, despite retaining all data. Indicates that replacing missing values with averages might introduce bias or reduce the variance in the dataset, affecting model performance. *Filling missing* values is <u>less effective for this dataset and may degrade recall, though it retains all data.</u>

Dropping Nulls: Higher accuracy and recall demonstrate improved overall performance. Removing rows with null values may eliminate noise or irrelevant data, enhancing the model's ability to generalize. *Dropping null* values <u>provides better performance overall, with higher accuracy and recall.</u> If high recall is critical (e.g., in applications where missing positives can have severe consequences), dropping null values is preferred for this dataset.

Decision Tree using filling method:



Decision Tree by dropping nulls:



Accuracy: Both methods achieved very high accuracy, with filling missing values performing slightly better (99.6% vs. 99.36%).

Precision: Precision is perfect (1.0 or 100%) in both cases, indicating the classifier did not produce any false positives.

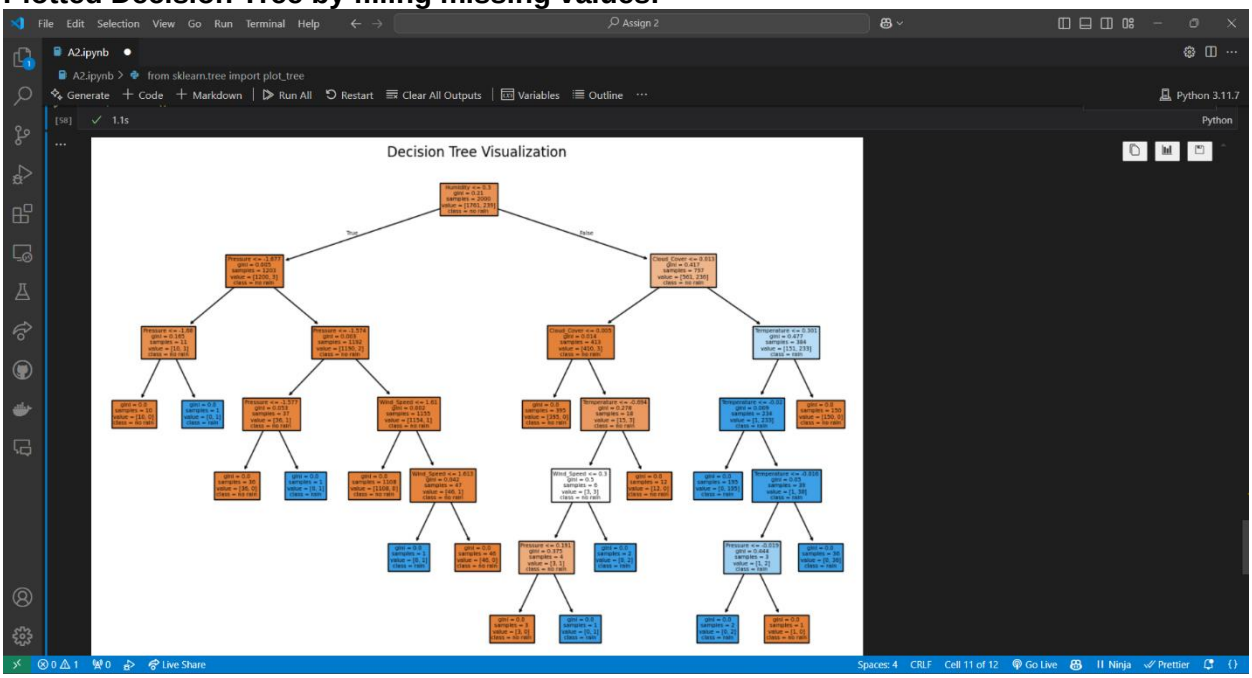
Recall: Recall is higher when missing values are filled (97.33%) compared to dropping nulls (94.34%).

Filling Missing Values: Retaining all data by filling missing values with the means leads to slightly higher performance across all metrics. *Filling missing values* offers slightly better performance, particularly in terms of recall, without compromising accuracy or precision. Use *filling missing values* <u>if your dataset is small or if recall is crucial for your application.</u>

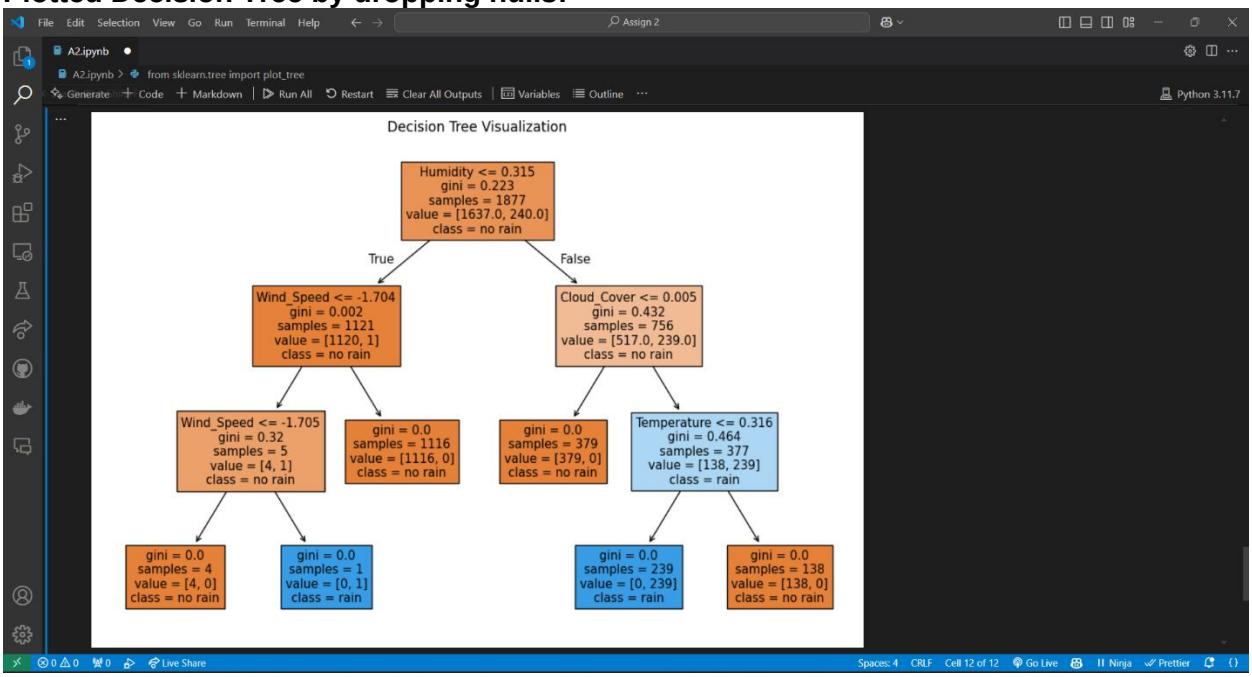
Dropping Nulls: While still highly accurate (99.36%), dropping null values slightly reduces recall (94.34%). Removing rows with missing data reduces the dataset size, which could limit the model's ability to generalize. *Dropping null values* also performs exceptionally well but has a slight disadvantage in identifying true positives due to a smaller dataset. Consider *dropping null values* <u>if missing data is sparse or the dataset is large enough to handle the reduction.</u>

Task 3.2, Create a well-formatted report that includes a plot of the decision tree:

Plotted Decision Tree by filling missing values:



Plotted Decision Tree by dropping nulls:



Tree with Missing Values Replaced by Averages: Contains more splits due to increased data size and greater variance introduced by imputation.
Tree with Dropped Null Values: Contains fewer splits because the dataset was reduced in size, resulting in simpler decision boundaries.

Decision-Making Process of the Decision Tree:

A decision tree predicts an outcome "rain" or "no rain" by sequentially splitting the data based on feature values. At each node, the following occur:

1. Feature Selection: The tree chooses the feature that best separates the data based on a criterion such as Gini Index or Entropy.
2. Splitting Logic: Based on a threshold value, the data is divided into two groups, one proceeding to the left child node (True) and the other to the right (False).

Splitting Criteria

- Gini Index: Measures the impurity of a node. A Gini value of 0 means the node is pure (all samples belong to a single class), while higher values indicate greater impurity.
- Samples: Indicates the number of data points at each node.
- Values: Shows the distribution of samples for each class at the node (e.g., [Class 0, Class 1]).
- Class: Denotes the dominant class at the node.

Tree Explanation

1. **Tree Root (Top Node)**
    - The root node splits on *Humidity*. If Humidity <= 0.315, the decision path proceeds to the left child node; otherwise, to the right. This split has a moderate Gini Index, suggesting some **impurity.**
2. **Intermediate Splits**
    - Nodes are further split based on features such as *Wind Speed, Temperature,* and *Cloud Cover*. For instance:
        - In the "no rain" subtree, *Wind Speed* is used to refine the prediction further.
        - In the "rain" subtree, *Temperature* helps distinguish between rain and no rain.
3. **Leaf Nodes**
    - Leaf nodes represent final predictions with no further splits. A Gini Index of 0 at these nodes indicates pure classifications (all samples belong to one class).

In conclusion, *filling missing* values results in more complex decision boundaries, which may capture subtle patterns or overfit the data. *Dropping missing* values simplifies the tree, potentially reducing overfitting but at the cost of losing information. The tree with *filling null values* is more complex due to retained sample size and broader variability, while the tree with *dropped null values* is simpler, as fewer samples and feature variability lead to fewer splits.

Task 3.3

Scikit-Learn KNN

```
K =  1
time:  0.028007030487060547
Accuracy:  0.9574468085106383
Precision:  0.8235294117647058
Recall:  0.7924528301886793
K =  3
time:  0.017821311950683594
Accuracy:  0.9702127659574468
Precision:  0.9148936170212766
Recall:  0.8113207547169812
K =  5
time:  0.019525766372680664
Accuracy:  0.9744680851063829
Precision:  0.9555555555555556
Recall:  0.8113207547169812
K =  9
time:  0.019391775131225586
Accuracy:  0.9744680851063829
Precision:  0.9361702127659575
Recall:  0.8301886792452831
K =  100
time:  0.034650325775146484
Accuracy:  0.9531914893617022
Precision:  1.0
Recall:  0.5849056603773585
```

Scratch KNN

```
K =  1
time:  0.034642696380615234
Accuracy:  0.9574468085106383
Precision:  0.8235294117647058
Recall:  0.7924528301886793
K =  3
time:  0.030536890029907227
Accuracy:  0.9702127659574468
Precision:  0.9148936170212766
Recall:  0.8113207547169812
K =  5
time:  0.02744436264038086
Accuracy:  0.9744680851063829
Precision:  0.9555555555555556
Recall:  0.8113207547169812
K =  9
time:  0.027953147888183594
Accuracy:  0.9744680851063829
Precision:  0.9361702127659575
Recall:  0.8301886792452831
K =  100
time:  0.027619600296020508
Accuracy:  0.9531914893617022
Precision:  1.0
Recall:  0.5849056603773585
```

Conclusion:

- Execution Time: Scikit-learn' s implementation is consistently faster than the custom implementation, with a time difference ranging from +0.0066 to +0.0097 seconds, depending on the value of K.
- Accuracy, Precision, and Recall: Both implementations achieve identical results for all performance metrics across all values of K.