Name: Ayush Maurya   Class: D15C/ Batch B   Roll No: 34

Experiment No 6

Aim: Classification modelling

a. Choose a classifier for a classification problem.

b. Evaluate the performance of the classifier.

- K-Nearest Neighbors (KNN)
- Decision Tree

Theory:

Classification Modeling: Theory & Techniques

Classification modeling is a type of supervised learning in machine learning where the goal is to predict the category or class of a given data point based on input features. The model is trained using labeled data (i.e., data where the output class is known).

Classification problems can be:

- **Binary Classification:** Two classes (e.g., spam vs. not spam).
- **Multiclass Classification:** More than two classes (e.g., classifying types of flowers).
- **Multi-label Classification:** Each sample can belong to multiple classes.

1. K-Nearest Neighbors (KNN)

K-Nearest Neighbors (KNN) is a simple, non-parametric classification algorithm based on proximity to labeled examples.

Working Principle:

1. Choose a value for K (number of neighbors).
2. Compute the distance between the new data point and all training samples.
3. Select the K nearest neighbors.
4. Assign the majority class among the K neighbors as the predicted class.

Common Distance Metrics:

- **Euclidean Distance:** $d=(\sum(xi-yi)^2)^{½}$ (Most commonly used)
- **Manhattan Distance:** $d=\sum|xi-yi|$
- **Minkowski Distance:** A generalization of Euclidean and Manhattan distances.

2. Naïve Bayes (NB)

Naïve Bayes is a probabilistic classifier based on **Bayes' Theorem**, assuming independence between predictors.

Bayes' Theorem:

$$P(A|B) = \frac{P(B|A)P(A)}{P(B)}$$

Where:

- P(A|B) = Probability of class A given data B
- P(B|A) = Probability of data B given class A
- P(A) = Prior probability of class A
- P(B) = Prior probability of data B

Types of Naïve Bayes Classifiers:

1. **Gaussian Naïve Bayes:** Assumes normal distribution of features.
2. **Multinomial Naïve Bayes:** Used for text classification (e.g., spam detection).
3. **Bernoulli Naïve Bayes:** Used when features are binary (e.g., word presence in spam detection).

3. Support Vector Machines (SVMs)

SVM is a powerful classification algorithm that finds the optimal hyperplane to separate data points into different classes.

Working Principle:

1. **Hyperplane:** A decision boundary that maximizes the margin between two classes.
2. **Support Vectors:** Data points that lie closest to the hyperplane and influence its position.
3. **Kernel Trick:** SVM can handle non-linearly separable data using kernel functions to transform the input space.

Common Kernel Functions:

Linear Kernel: $K(x, y) = x^T y$

Polynomial Kernel: $K(x, y) = (x^T y + c)^d$

Radial Basis Function (RBF) Kernel: $K(x, y) = e^{-\gamma||x-y||^2}$

Sigmoid Kernel: $K(x, y) = \tanh(\alpha x^T y + c)$

4. Decision Tree

A Decision Tree is a flowchart-like structure where internal nodes represent features, branches represent decisions, and leaves represent class labels.

Working Principle:

1. **Splitting Criteria:** Choose the best feature to split the data.
    - **Gini Index:** Measures impurity (Gini=$1-\sum pi^2$).
    - **Entropy (Information Gain):** Measures information gained from a split.
2. **Recursive Splitting:** Continue splitting nodes until a stopping criterion is met.
3. **Pruning:** Reduces overfitting by trimming branches.

Types of Decision Trees:

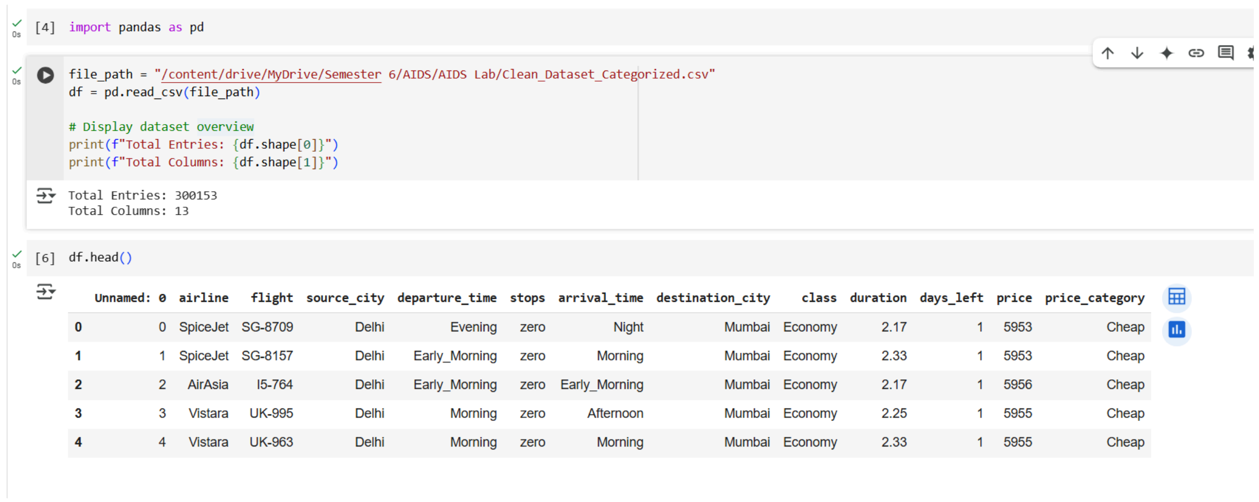
- **ID3 (Iterative Dichotomiser 3)** – Uses entropy for splitting.
- **C4.5 & C5.0** – Improvement over ID3 (handles continuous data).
- **CART (Classification and Regression Trees)** – Uses Gini Index.

Name: Ayush Maurya Class: D15C/ Batch B Roll No: 34

Steps:

Step 1: Load the Dataset

The dataset is loaded from a CSV file using pandas and First 5 entries in the Dataset is shown using df.head() and Total rows and columns are printed using df.shape[n].



```
[4] import pandas as pd
```

```
file_path = "/content/drive/MyDrive/Semester 6/AIDS/AIDS Lab/Clean_Dataset_Categorized.csv"
df = pd.read_csv(file_path)

# Display dataset overview
print(f"Total Entries: {df.shape[0]}")
print(f"Total Columns: {df.shape[1]}")
```

```
Total Entries: 300153
Total Columns: 13
```

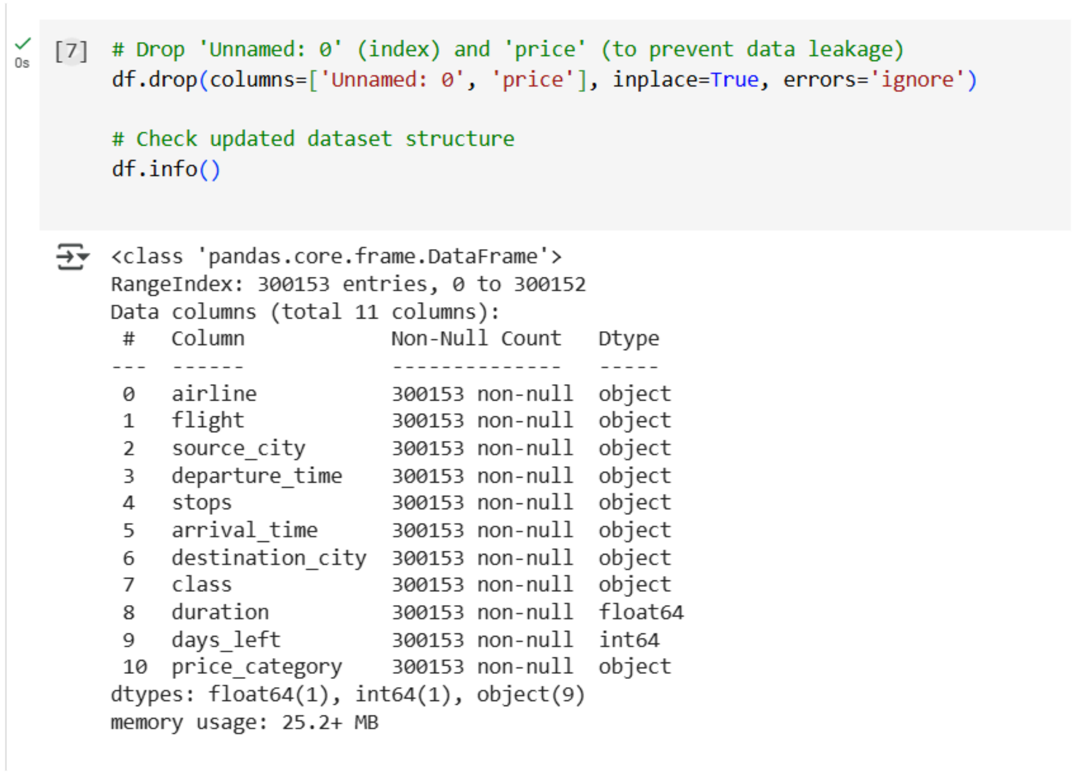
```
[6] df.head()
```

	Unnamed: 0	airline	flight	source_city	departure_time	stops	arrival_time	destination_city	class	duration	days_left	price	price_category
0	0	SpiceJet	SG-8709	Delhi	Evening	zero	Night	Mumbai	Economy	2.17	1	5953	Cheap
1	1	SpiceJet	SG-8157	Delhi	Early_Morning	zero	Morning	Mumbai	Economy	2.33	1	5953	Cheap
2	2	AirAsia	I5-764	Delhi	Early_Morning	zero	Early_Morning	Mumbai	Economy	2.17	1	5956	Cheap
3	3	Vistara	UK-995	Delhi	Morning	zero	Afternoon	Mumbai	Economy	2.25	1	5955	Cheap
4	4	Vistara	UK-963	Delhi	Morning	zero	Morning	Mumbai	Economy	2.33	1	5955	Cheap

Step 2: Data Preprocessing

1) Drop Unnecessary Columns

For the following experiment, the columns such as Unnamed and price are not required for classification. So to start preprocessing, we drop such columns using **df.drop(columns=[])** command.

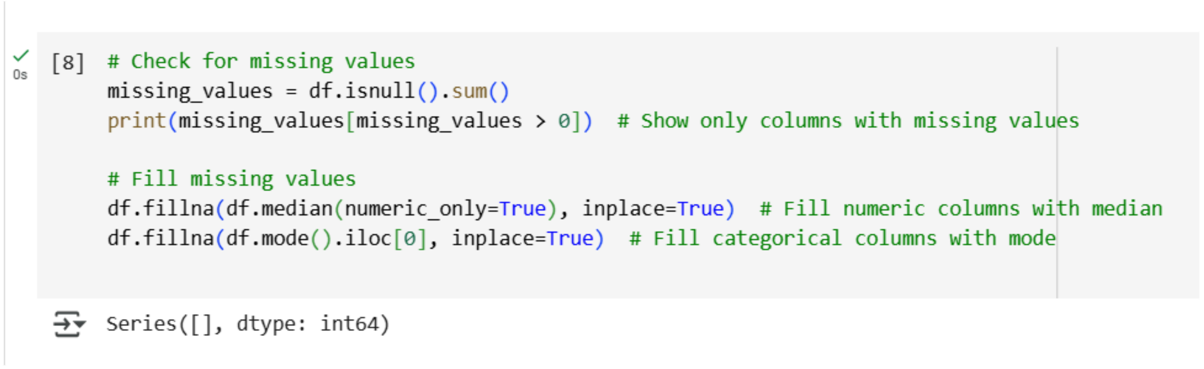


```
[7] # Drop 'Unnamed: 0' (index) and 'price' (to prevent data leakage)
    df.drop(columns=['Unnamed: 0', 'price'], inplace=True, errors='ignore')

    # Check updated dataset structure
    df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 300153 entries, 0 to 300152
Data columns (total 11 columns):
 #   Column            Non-Null Count   Dtype
---  ------            --------------   -----
 0   airline           300153 non-null  object
 1   flight            300153 non-null  object
 2   source_city       300153 non-null  object
 3   departure_time    300153 non-null  object
 4   stops             300153 non-null  object
 5   arrival_time      300153 non-null  object
 6   destination_city  300153 non-null  object
 7   class             300153 non-null  object
 8   duration          300153 non-null  float64
 9   days_left         300153 non-null  int64
 10  price_category    300153 non-null  object
dtypes: float64(1), int64(1), object(9)
memory usage: 25.2+ MB
```

2) Handling Missing Values

We fill up the missing values such that the numeric columns are filled with the median values, and the categorical columns are filled with mode of that column.

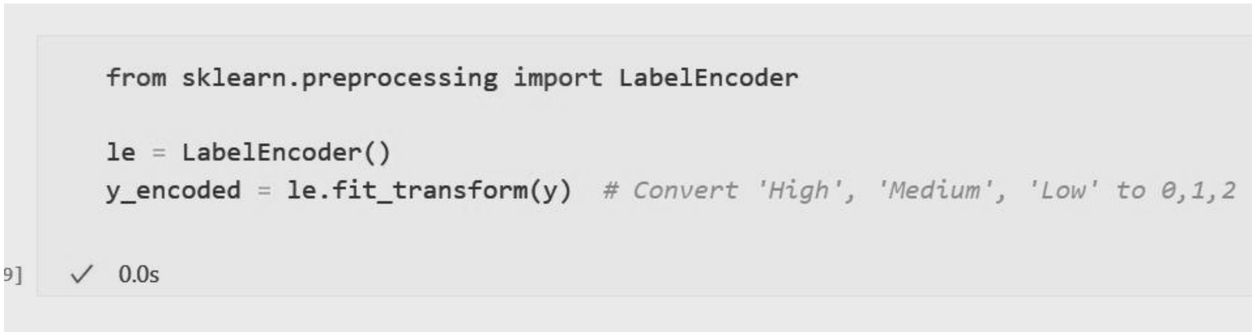


```
[8] # Check for missing values
    missing_values = df.isnull().sum()
    print(missing_values[missing_values > 0])  # Show only columns with missing values

    # Fill missing values
    df.fillna(df.median(numeric_only=True), inplace=True)  # Fill numeric columns with median
    df.fillna(df.mode().iloc[0], inplace=True)  # Fill categorical columns with mode
```

```
Series([], dtype: int64)
```

3) Encode Categorical Variables

The categorical columns are encoded so that it becomes suitable for the algorithm to make it easier for the algorithm to make the classification.

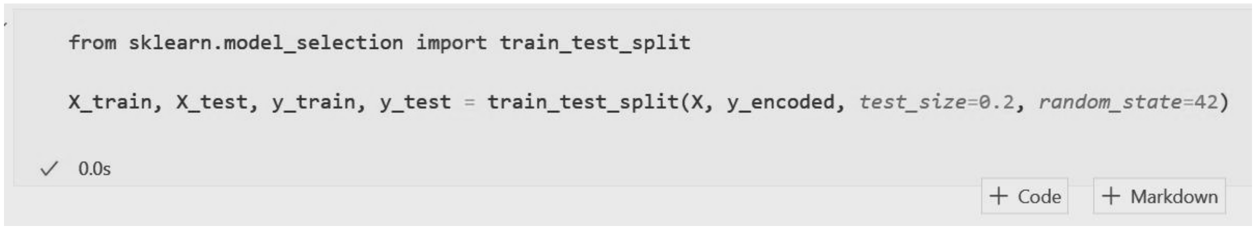


```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
y_encoded = le.fit_transform(y)  # Convert 'High', 'Medium', 'Low' to 0,1,2
```

```
✓ 0.0s
```

Step 3: Split Into Train and Test

The dataset it then split into training and testing such that the models are trained on 80% of the dataset and 20% is used to test the models for their accuracy.



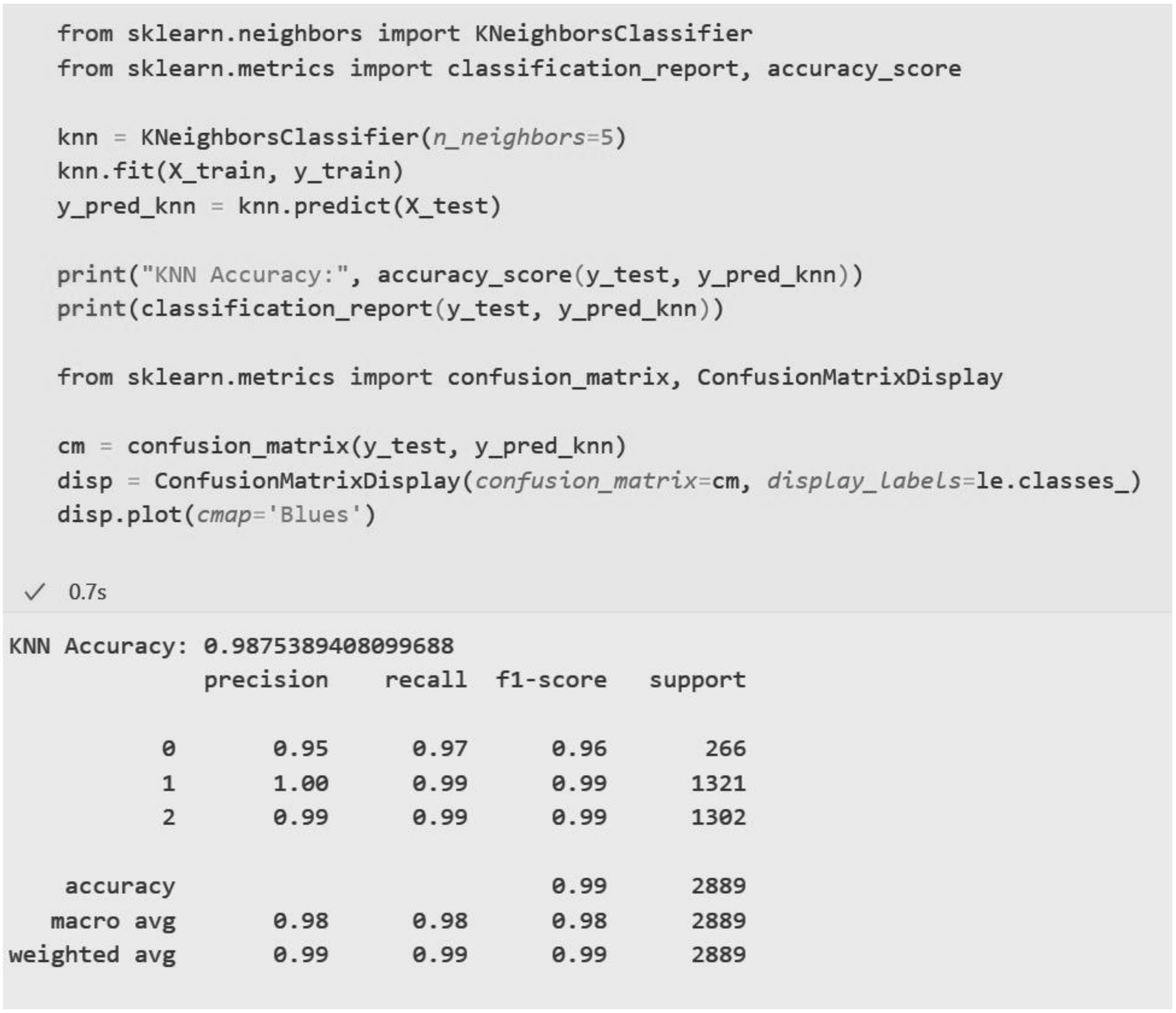
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y_encoded, test_size=0.2, random_state=42)
```

```
✓ 0.0s
```

Step 4: Train & Evaluate Classifiers

1)K-Nearest Neighbors (KNN)

From sklearn.neighbors library, we import the KNeighboursClassifier. We call this function and pass a parameter for the number of neighbours to be used. Here, we are passing 5 neighbours. After that, we fit the model with our training datasets (X and y) and create a variable to store the predicted values.



```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import classification_report, accuracy_score

knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(X_train, y_train)
y_pred_knn = knn.predict(X_test)

print("KNN Accuracy:", accuracy_score(y_test, y_pred_knn))
print(classification_report(y_test, y_pred_knn))

from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay

cm = confusion_matrix(y_test, y_pred_knn)
disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=le.classes_)
disp.plot(cmap='Blues')
```

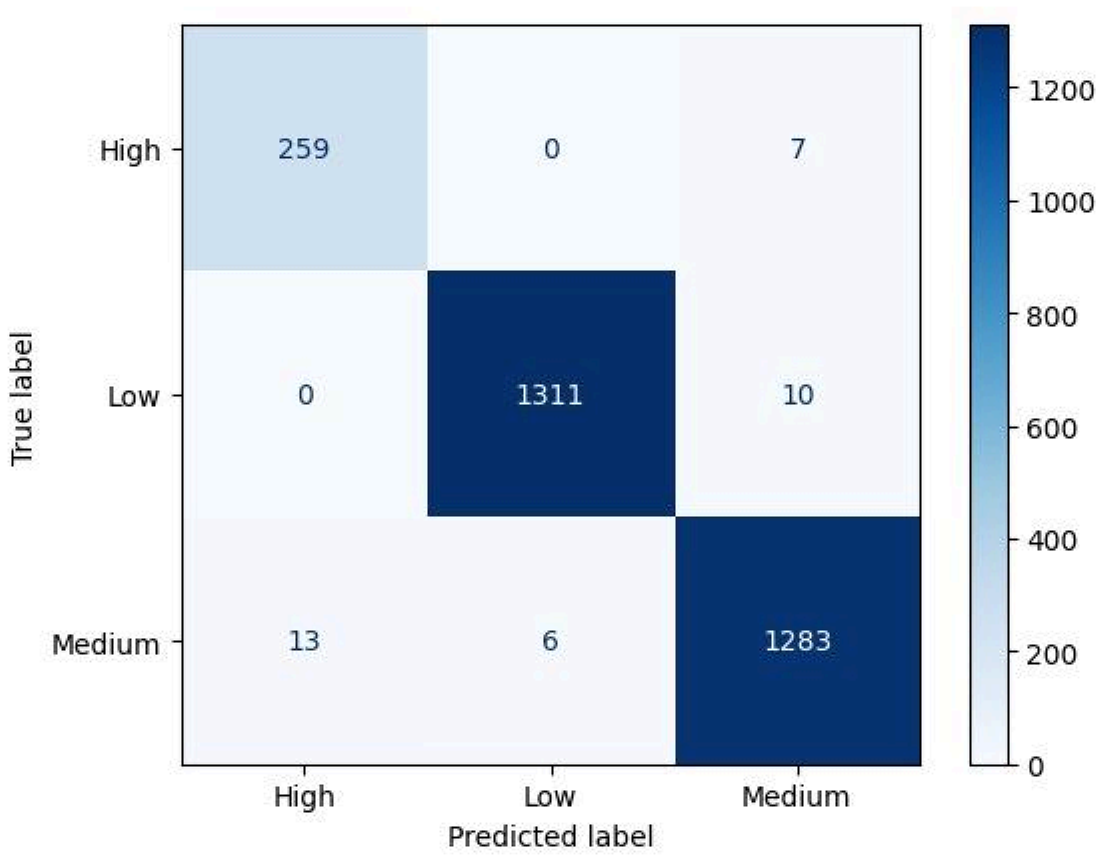
✓ 0.7s

```
KNN Accuracy: 0.9875389408099688
              precision    recall  f1-score   support

           0       0.95      0.97      0.96       266
           1       1.00      0.99      0.99      1321
           2       0.99      0.99      0.99      1302

    accuracy                           0.99      2889
   macro avg       0.98      0.98      0.98      2889
weighted avg       0.99      0.99      0.99      2889
```

I) Classification Report

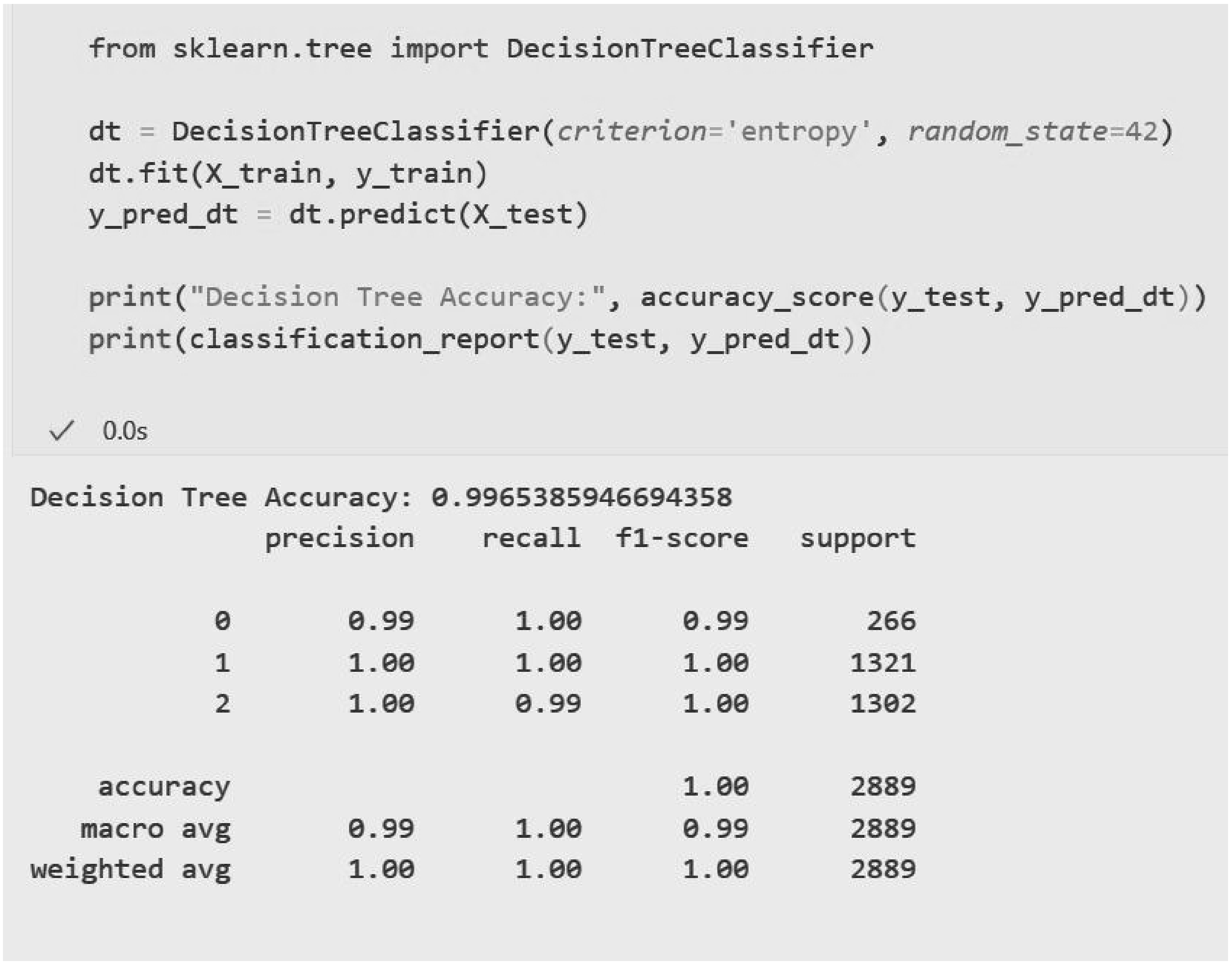
K-Nearest Neighbors (KNN) classification model performed exceptionally well, achieving an overall accuracy of 98.75%. The confusion matrix shows that most predictions are correct, with very few misclassifications—mainly between the High and Medium classes, and slightly between Medium and Low. Precision, recall, and F1-scores across all three classes (High, Low, Medium) are consistently high, especially for the Low class where the model achieved nearly perfect scores. This indicates that the feature space is well-separated, and the KNN algorithm is effectively capturing the underlying structure of the data. The low number of off-diagonal values in the matrix shows that there are minimal overlaps between classes, making KNN a good fit for this dataset. The macro and weighted averages being close also suggest the model handles class imbalance well. Overall, the model demonstrates high reliability and robustness in classifying battery RUL categories.

II) Confusion Matrix


High
Low
Medium
True label
259
0
7
0
1311
10
13
6
1283
High
Low
Medium
Predicted label
1200
1000
800
600
400
200
0

2)Decision Tree

Before pre-pruning

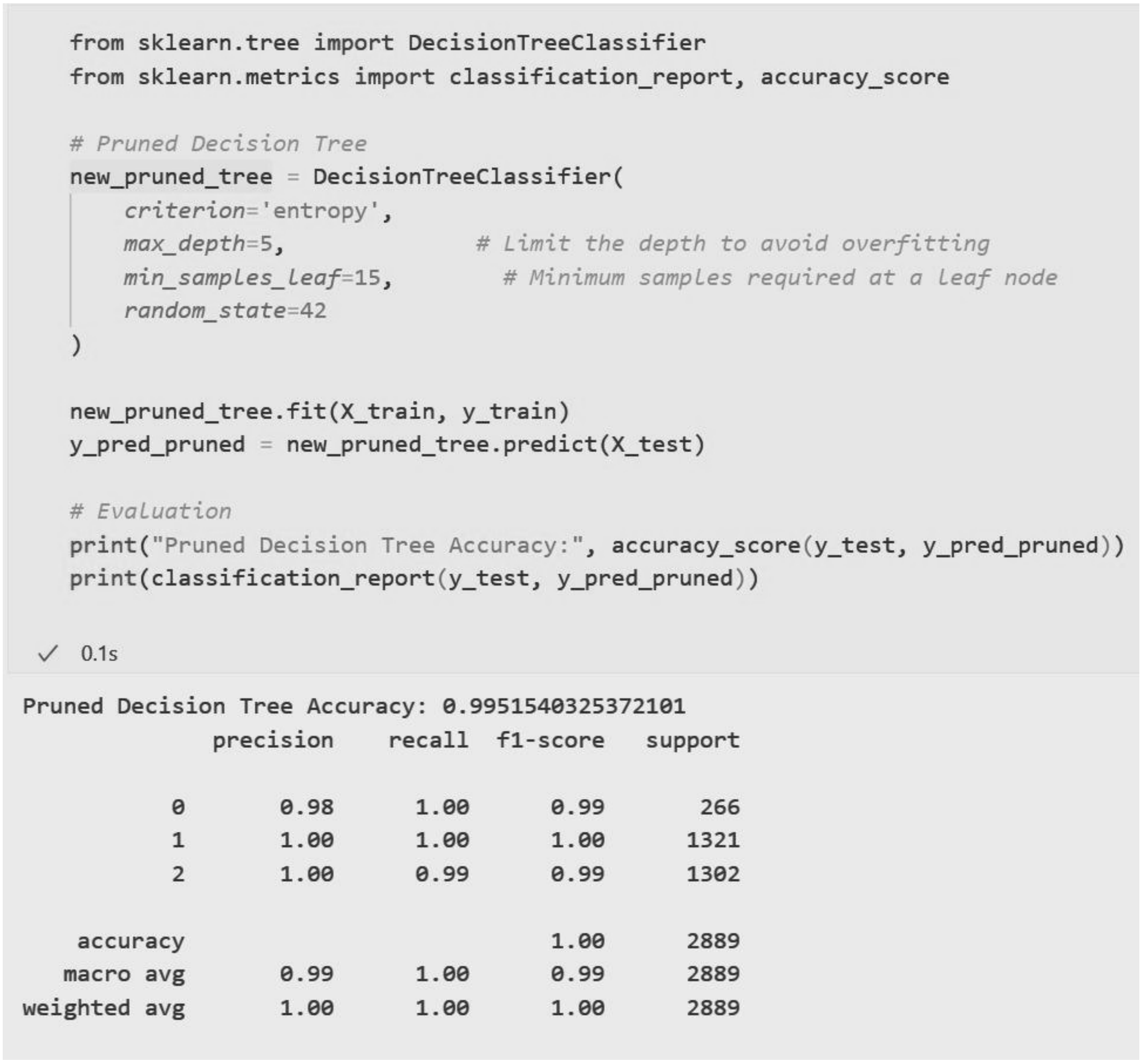


```
from sklearn.tree import DecisionTreeClassifier

dt = DecisionTreeClassifier(criterion='entropy', random_state=42)
dt.fit(X_train, y_train)
y_pred_dt = dt.predict(X_test)

print("Decision Tree Accuracy:", accuracy_score(y_test, y_pred_dt))
print(classification_report(y_test, y_pred_dt))
```

✓ 0.0s

```
Decision Tree Accuracy: 0.9965385946694358
              precision    recall  f1-score   support

           0       0.99      1.00      0.99       266
           1       1.00      1.00      1.00      1321
           2       1.00      0.99      1.00      1302

    accuracy                           1.00      2889
   macro avg       0.99      1.00      0.99      2889
weighted avg       1.00      1.00      1.00      2889
```

After pre-pruning

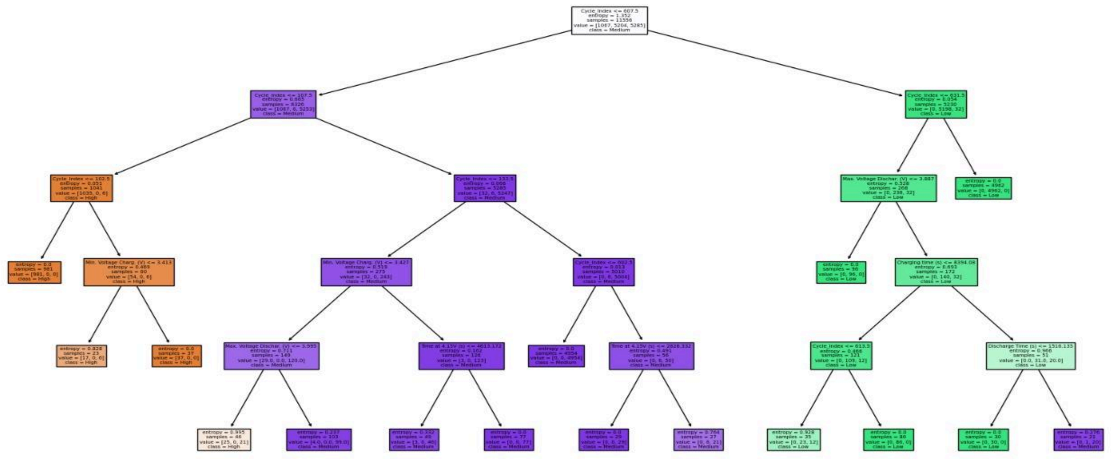


```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report, accuracy_score

# Pruned Decision Tree
new_pruned_tree = DecisionTreeClassifier(
    criterion='entropy',
    max_depth=5,             # Limit the depth to avoid overfitting
    min_samples_leaf=15,       # Minimum samples required at a leaf node
    random_state=42
)

new_pruned_tree.fit(X_train, y_train)
y_pred_pruned = new_pruned_tree.predict(X_test)

# Evaluation
print("Pruned Decision Tree Accuracy:", accuracy_score(y_test, y_pred_pruned))
print(classification_report(y_test, y_pred_pruned))
```

✓ 0.1s

```
Pruned Decision Tree Accuracy: 0.9951540325372101
              precision    recall  f1-score   support

           0       0.98      1.00      0.99       266
           1       1.00      1.00      1.00      1321
           2       1.00      0.99      0.99      1302

    accuracy                           1.00      2889
   macro avg       0.99      1.00      0.99      2889
weighted avg       1.00      1.00      1.00      2889
```

The comparison between the original and pruned decision tree highlights the importance of pruning in machine learning. While the unpruned decision tree achieves a slightly higher accuracy of 99.65%, it risks overfitting, meaning it may perform very well on training data but fail to generalize to new unseen data. On the other hand, the pruned decision tree, with an accuracy of 99.51%, introduces a small drop in accuracy but enhances the model's generalization and interpretability. Pruning restricts the tree’s growth (e.g., by setting max_depth=5 and min_samples_leaf=15), which reduces model complexity, avoids learning noise, and helps the model stay focused on meaningful patterns. In practical applications, especially with real-world, noisy data, a slightly less accurate but pruned model is usually more robust and reliable than a perfectly accurate yet overly complex model.



This is pruned decision tree

Conclusion:

Both the K-Nearest Neighbors (KNN) and Decision Tree (with and without pruning) models show strong classification performance on the battery RUL dataset. However, the key insight lies not just in accuracy but in model behavior, generalization, and practical usability. The unpruned decision tree offers near-perfect accuracy but risks overfitting, making it less reliable on unseen data. By applying pruning techniques, we slightly reduced the accuracy but gained a model that is more interpretable and generalizable—crucial for real-world deployment. In contrast, the KNN model, with its non-parametric nature and high precision/recall scores, proved to be both simple and powerful, especially when classes are well separated.

Through this experiment, I learned the importance of balancing accuracy with generalization. Pruning helps simplify complex models, preventing them from capturing noise, while models like KNN show how effectively simplicity and distance-based logic can perform when the feature space is clean and structured. Overall, this experiment reinforced the value of evaluating models not just by accuracy but also by their ability to generalize, their interpretability, and robustness to variation—key aspects in building reliable predictive systems for battery health and RUL classification.