Implementation:

1- Sina is a computer engineering student. He has decided to take a break on Fridays and not go to work or school. Sina usually either stays at home or does one of the activities such as shopping, going to a café, or hiking. It should be noted that the surrounding conditions also affect the activity he chooses and performs. For example, if Sina's bank card has no balance, he will not go to the café, or if the weather is snowy or rainy, he cannot go hiking.

Kian, who is one of Sina's close friends, decides to take notes on Sina's activities and a set of conditions that are effective in his decision-making to predict his activities in the coming weeks. The results of Kian's notes for 10 consecutive weeks can be seen in the table below:

Sina's decision	Exam	Account balance	Weather condition	Week
Shopping	False	True	Sunny	1
Coffee shop	True	True	Sunny	2
Coffee shop	False	True	Snowy	3
Climbing	True	True	Sunny	4
Stay at home	True	True	Rainy	5
Climbing	False	False	Sunny	6
Climbing	False	True	Sunny	7
Coffee shop	True	True	Rainy	8
Shopping	True	True	Snowy	9
Stay at home	True	False	Snowy	10

1-1 Kian intends to implement a simple decision tree to help him predict Sina's status on future Fridays.

1-2 Report the results in a rule-based format.

Description of implemented functions:

1- Importing the necessary libraries:

```
import pandas as pd
from pprint import pprint
import json
```

We add the above libraries to the project environment for implementing the following functions. These libraries include Pandas, pprint library, and JSON library. The pprint library is used to print the decision tree in a beautiful format. The JSON library is also used to convert data sets to the JSON format.

2- Implementation of the Gini-index calculation function:

```
def gini_index(data: pd.DataFrame, feature, label, classes):
    total_gini = 0
    samples = len(data)
    for value in data[feature].unique():
        sub_tree_labels = data[data[feature] == value][label].value_counts()
        sub_tree_samples = len(data[data[feature] == value])
        sum_gini = 0
        for c in classes:
            pi = sub_tree_labels.get(c, 0) / sub_tree_samples
            sum_gini += pi ** 2
        sum_gini = 1 - sum_gini
        sum_gini = sum_gini * sub_tree_samples
        total_gini += sum_gini / samples

    return total_gini
```

This function calculates the Gini index for a specific feature in the data. Calculating this index plays an important role in the process of learning, generalizing branches, and breaking down appropriate features, along with the entropy criterion. The Gini index measures the impurity of a set of samples, where the value 0 indicates a

completely pure set (i.e., all samples have the same class) and the value 1 indicates a completely impure set (equal number of samples from each class).

This function takes four arguments:

•Data (as a Pandas DataFrame)

•The feature for calculating the Gini index

•The label column (which includes class labels for each sample)

•The list of classes.

3- Calculation of the best feature for Split:

```
def calculate_best_split(data: pd.DataFrame, label, classes):
    available_features = set(data.columns) - set([label])
    assert len(available_features) > 0

    best_gini = 1
    best_feature = None
    for f in available_features:
        gini = gini_index(data, f, label, classes)
        if gini < best_gini:
            best_gini = gini
            best_feature = f

    return best_feature
```

Using this function and calling the Gini index calculation function, at each step, we will select the best feature for extending and creating branches. This function is repeated on all existing features in the data, and when used for dividing the data, it selects the lowest Gini index.

This function takes three arguments:

•Data (as a Pandas DataFrame)

•The label column (which includes class labels for each sample)

•The list of classes.

4- Data Separator Function:

```python
def split_data(data: pd.DataFrame, feature):
    sub_datas = {}
    for value in data[feature].unique():
        sub_tree_data = data[data[feature] == value]
        sub_tree_data = sub_tree_data.drop(feature, axis=1)
        sub_datas[value] = sub_tree_data

    return sub_datas
```

After identifying the best feature for branching by the above functions, we need to divide the dataset into subsets based on that specific feature's unique values. This function divides the data into subsets based on the unique values of a feature.

This function takes two arguments:

•Data (as a Pandas DataFrame)

•The feature for splitting

5- Decision Tree Creation Function:

```python
def make_decision_tree(data: pd.DataFrame, label='label', classes=None, return_prob=True):
    available_features = set(data.columns) - set([label])
    if len(available_features) == 0 or len(data[label].unique()) == 1:
        if return_prob:
            return json.loads((data['label'].value_counts() / len(data)).to_json())
        else:
            return {'label': data[label].value_counts().idxmax()}

    split_feature = calculate_best_split(data, label, classes)
    sub_datas = split_data(data, split_feature)

    if len(sub_datas.keys()) == 1:
        if return_prob:
            return json.loads((data['label'].value_counts() / len(data)).to_json())
        else:
            return {'label': data[label].value_counts().idxmax()}

    results = (split_feature, {})
    for key in sub_datas.keys():
        results[1][key] = make_decision_tree(sub_datas[key], label, classes)

    return results
```

This function is implemented to build and form a decision tree. In calling this function, first, the calculate_best_split function is called to find the best feature to break down. Then, by calling the split_data function, we divide the current subtree into two different subtrees based on the selected feature, and this function recursively forms the decision tree on each one. This function will be recursively called until we reach the leaves, and the branches and subtrees will be formed recursively until reaching the leaves. When the features reach 0 in a subtree or the number of different labels in a node reaches 1, we are inside a leaf, and there is no need to form a subtree anymore.

By raising the return_prob flag, we can declare the probability of decision-making in each leaf.

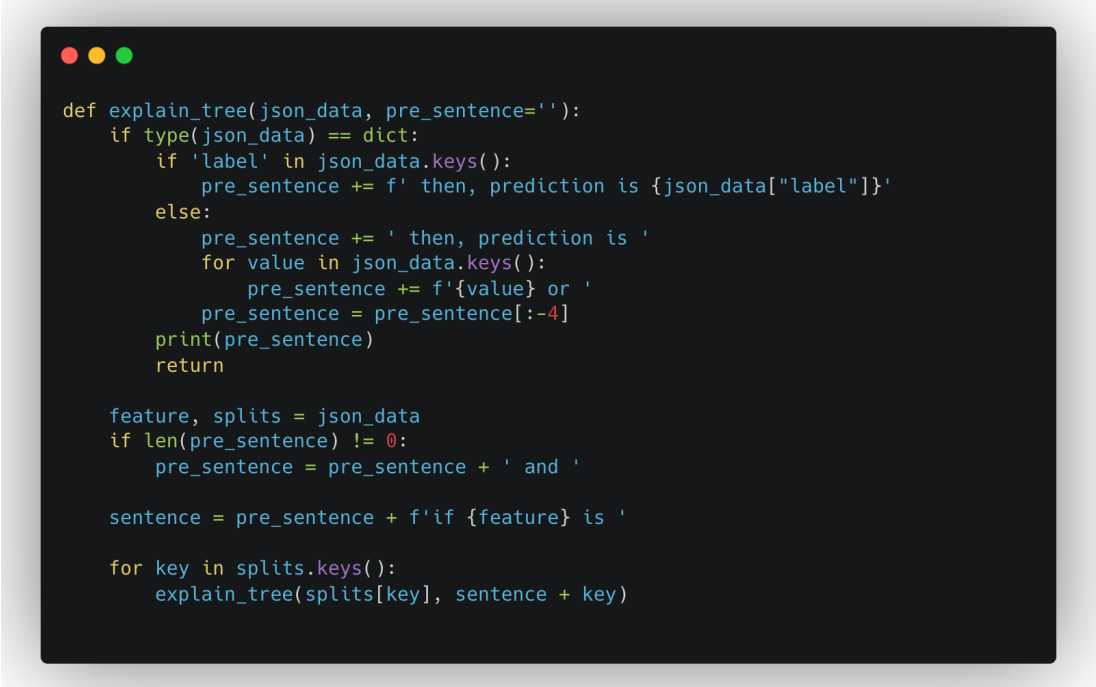
This function takes four arguments:

•Data (as a Pandas DataFrame)

•The label column (which includes class labels for each sample)

•The list of classes

6- Reporting the Results in a Rule-Based Format:



```
def explain_tree(json_data, pre_sentence=''):
    if type(json_data) == dict:
        if 'label' in json_data.keys():
            pre_sentence += f' then, prediction is {json_data["label"]}'
        else:
            pre_sentence += ' then, prediction is '
            for value in json_data.keys():
                pre_sentence += f'{value} or '
            pre_sentence = pre_sentence[:-4]
        print(pre_sentence)
        return

    feature, splits = json_data
    if len(pre_sentence) != 0:
        pre_sentence = pre_sentence + ' and '

    sentence = pre_sentence + f'if {feature} is '

    for key in splits.keys():
        explain_tree(splits[key], sentence + key)
```

As mentioned in the benefits of decision trees, the algorithm's decision-making is very close to human decision-making in the form of if-then rules. This function prints a human-readable explanation of the decision tree. The function recursively traverses the decision tree and adds a sentence to the text and the previous sentence for each branch and feature split, respectively, to print the modeled rule.

This function takes two arguments:

•The decision tree (as a JSON object)

•A string containing a piece of the sentence that is completed recursively at each step.

7- Creating the Dataset:

```
data = [
    {'weather': 'sun',  'cash': 'yes',  'exam': 'no',   'label': 'shop'},
    {'weather': 'sun',  'cash': 'yes',  'exam': 'yes',  'label': 'coffe'},
    {'weather': 'snow', 'cash': 'yes',  'exam': 'no',   'label': 'coffe'},
    {'weather': 'sun',  'cash': 'yes',  'exam': 'yes',  'label': 'hiking'},
    {'weather': 'rain', 'cash': 'yes',  'exam': 'yes',  'label': 'stay'},
    {'weather': 'sun',  'cash': 'no',   'exam': 'no',  'label': 'hiking'},
    {'weather': 'sun',  'cash': 'yes',  'exam': 'no',   'label': 'hiking'},
    {'weather': 'rain', 'cash': 'yes',  'exam': 'yes',  'label': 'coffe'},
    {'weather': 'snow', 'cash': 'yes',  'exam': 'yes',  'label': 'shop'},
    {'weather': 'snow', 'cash': 'no',   'exam': 'yes',  'label': 'stay'}
]

df = pd.DataFrame(data)
```

We store the datasets and reports written about Sina's decisions in a dictionary format and create a DataFrame from them.

8- Creating Decision Tree and Displaying Tree Output:

```
classes = df['label'].unique()
tree = make_decision_tree(df, 'label', classes)
pprint(tree)
explain_tree(tree)
```

Finally, we call the implemented functions by entering the dataset along with their labels to create a decision tree.

You can view the created decision tree and rule-based rules in Chapter 3 or the Results section.

2- In this section, we are going to implement a decision tree to estimate the quality of milk production in a factory. To do this, you can expand the functions from the previous section and complete them. The dataset for this question is accessible in the attachment, which includes 8 columns. The descriptions of each column are as follows:

- **PH:** This column defines PH of the milk which ranges from 3 to 9
- **Temperature:** This column defines temperature of the milk which ranges from 34'C to 90'C
- **Taste:** This column defines the taste of the milk which is categorical data: 0 (Bad) or 1 (Good)
- **Odor:** This column defines the odor of the milk which is categorical data: 0 (Bad) or 1 (Good)
- **Fat:** This column defines the fat level of the milk which is categorical data:0 (Low) or 1 (High)
- **Turbidity:** This column defines the turbidity of the milk which is categorical data: 0 (Low) or 1 (High)
- **Color:** This column defines the color of the milk which ranges from 240 to 255
- **Grade:** This column defines the grade (target) of the milk which is categorical data: Low (Bad) or Medium (Moderate) High (Good)

2-1. Divide the dataset into two parts: training and testing data (90% training - 10% testing).

2-2. Perform necessary pre-processing.

2-3. Use the following two criteria to select the best feature when building the tree. Does using different criteria affect the model's performance? Choose the best criterion.

• Entropy
• Gini-index

One way to prevent overfitting in decision trees is to use pruning techniques such as pre-pruning or post-pruning. These techniques can prevent the model from becoming overly complex. Apply the following two techniques at this stage.

2-4. Pre-pruning: Perform pruning by considering the following:
• Determine the maximum depth of the tree, and do not continue building the tree after reaching the maximum depth.
• Do not continue splitting a branch if the number of data points entering that branch is insignificant. Does using this technique affect the performance of the model? Report the results of your experiments.

2-5. Post-pruning: After building the tree completely, try to remove some of the subtrees. Does removing these subtrees affect the performance of the model? Report the results of your experiments.

2-6. Model Evaluation: Using the following criteria, evaluate the best model you have found on the test data and analyze your results.
• Macro accuracy - micro accuracy
• Macro precision - micro precision
• Macro recall - micro recall
• F1-score

.2-7After performing the above steps, in this section, train a decision tree on the same dataset using the scikit-learn library. Evaluate the model's performance on the test dataset and report whether the results of the model differ when using the pre-built library from the results obtained in the previous sections. Analyze the reasons.

Description of Implemented Functions:

1- Importing Required Libraries:

To implement the functions we need, we add the following libraries to the project environment:

Pandas library: a popular library for working with data frames and tables in Python.

NumPy library: for working with numerical data in Python.

JSON library: for converting a dictionary to variables of the JSON type or vice versa.

Matplotlib library: a popular library for creating visualizations and graphs in Python.

The deepcopy function is a function that creates a deep copy of an object, meaning it creates a completely new object with its own memory space. It is usually used for sampling variables that are pass by reference.

We import the accuracy_score and classification_report functions from the Sklearn library to calculate the required evaluation metrics.

We import the DecisionTreeClassifier class from the scikit-learn library. We will use the functions inside this class to train the model for comparison with the implemented model.

2- Implementation of the Gini-index Calculation Function:

The implementation of this function is generally similar to the function described in the previous question, with the difference that in the implementation of this function, another input is considered as a threshold for calculating the best split point for numerical variables.

To calculate the best split point for each numerical feature, all possible split scenarios are calculated and the best one is selected.

3- Implementation of the Entropy Calculation Function:

This function is very similar to the Gini calculation function, with the difference that for each feature, the entropy measure is calculated instead of Gini using the formula for calculating entropy, and it is used to generalize and split the best categorized or numerical feature.

The formula for calculating the average entropy of branches:

The defined entropy function receives multiple parameters as input:

- Data: a Pandas DataFrame containing the data set
- Feature: the name of the feature (column) on which we want to branch

- Label: the labels of the data
- Classes: a list of all labels inside the dataset
- Threshold: the threshold for dividing nodes and generalizing branches.

Description of Implemented Functions:

1- Importing Required Libraries:

```
import pandas as pd
import numpy as np
import json
import matplotlib.pyplot as plt
from copy import deepcopy
from sklearn.metrics import accuracy_score, classification_report
from sklearn.tree import DecisionTreeClassifier
```

To implement the functions we need, we add the following libraries to the project environment:

Pandas library: a popular library for working with data frames and tables in Python.

NumPy library: for working with numerical data in Python.

JSON library: for converting a dictionary to variables of the JSON type or vice versa.

Matplotlib library: a popular library for creating visualizations and graphs in Python.

The deepcopy function is a function that creates a deep copy of an object, meaning it creates a completely new object with its own memory space. It is usually used for sampling variables that are pass by reference.

We import the accuracy_score and classification_report functions from the Sklearn library to calculate the required evaluation metrics.

We import the DecisionTreeClassifier class from the scikit-learn library. We will use the functions inside this class to train the model for comparison with the implemented model.

2- Implementation of the Gini-index Calculation Function:

```python
def gini_index(data: pd.DataFrame, feature, label, classes, threshold):
    total_gini = 0
    samples = len(data)

    sub_tree_labels_high = data[data[feature] >= threshold][label].value_counts()
    sub_tree_samples_high = len(data[data[feature] >= threshold])
    sum_gini_high = 0
    for c in classes:
        pi = sub_tree_labels_high.get(c, 0) / sub_tree_samples_high
        sum_gini_high += pi ** 2
    sum_gini_high = 1 - sum_gini_high
    sum_gini_high = sum_gini_high * sub_tree_samples_high
    total_gini += sum_gini_high / samples

    sub_tree_labels_low = data[data[feature] < threshold][label].value_counts()
    sub_tree_samples_low = len(data[data[feature] < threshold])
    sum_gini_low = 0
    for c in classes:
        pi = sub_tree_labels_low.get(c, 0) / sub_tree_samples_low
        sum_gini_low += pi ** 2
    sum_gini_low = 1 - sum_gini_low
    sum_gini_low = sum_gini_low * sub_tree_samples_low
    total_gini += sum_gini_low / samples

    return total_gini
```

The implementation of this function is generally similar to the function described in the previous question, with the difference that in the implementation of this function, another input is considered as a threshold for calculating the best split point for numerical variables.

To calculate the best split point for each numerical feature, all possible split scenarios are calculated and the best one is selected.

3- Implementation of the Entropy Calculation Function:

```python
def entropy(data: pd.DataFrame, feature, label, classes, threshold):
    total_entropy = 0
    samples = len(data)

    sub_tree_labels_high = data[data[feature] >= threshold][label].value_counts()
    sub_tree_samples_high = len(data[data[feature] >= threshold])
    sum_entropy_high = 0
    for c in classes:
        pi = sub_tree_labels_high.get(c, 0) / sub_tree_samples_high
        if pi != 0:
            sum_entropy_high -= pi * np.log(pi)
        # else: pi * np.log(pi) is 0 and there is no need to compute
    sum_entropy_high = sum_entropy_high * sub_tree_samples_high
    total_entropy += sum_entropy_high / samples

    sub_tree_labels_low = data[data[feature] < threshold][label].value_counts()
    sub_tree_samples_low = len(data[data[feature] < threshold])
    sum_entropy_low = 0
    for c in classes:
        pi = sub_tree_labels_low.get(c, 0) / sub_tree_samples_high
        if pi != 0:
            sum_entropy_low -= pi * np.log(pi)
        # else: pi * np.log(pi) is 0 and there is no need to compute
    sum_entropy_low = sum_entropy_low * sub_tree_samples_low
    total_entropy += sum_entropy_low / samples

    return total_entropy
```

This function is very similar to the Gini calculation function, with the difference that for each feature, the entropy measure is calculated instead of Gini using the formula for calculating entropy, and it is used to generalize and split the best categorized or numerical feature.

The formula for calculating the average entropy of branches:

$$I(S,A)=\sum_i \frac{|S_i|}{|S|}\cdot E(S_i)$$

The defined entropy function receives multiple parameters as input:

- Data: a Pandas DataFrame containing the data set
- Feature: the name of the feature (column) on which we want to branch

- Label: the labels of the data
- Classes: a list of all labels inside the dataset
- Threshold: the threshold for dividing nodes and generalizing branches.

4 - Function for calculating the best numerical split point:

```python
def calculate_best_threshold(data: pd.DataFrame, feature, label, classes, purity_function):
    values = sorted(data[feature].unique())
    thresholds = [(values[i + 1] + values[i]) / 2 for i in range(len(values) - 1)]

    best_purity = 1
    best_th = 0
    for threshold in thresholds:
        purity = purity_function(data, feature, label, classes, threshold)
        if purity < best_purity:
            best_purity = purity
            best_th = threshold
    return best_purity, best_th
```

This function is used to calculate the best split point for numerical features. At each iteration, it takes the average of two adjacent values as the threshold and returns the best split point. Initially, it sorts all the values in the feature and then iterates over all the values along with the entropy or Gini coefficient functions (as specified in the problem statement) to find the value that has the best and lowest entropy or Gini coefficient.

The function loops over all threshold values and calculates the split criterion, which is calculated based on the "purity_function" variable. If the current split purity is lower than the previous best purity, the "best_purity" and "best_th" variables are updated with the current values, and the learning process continues with that value.

5 - Function for calculating the best feature for Split:

```python
def calculate_best_split(data: pd.DataFrame, label, classes, purity_function):
    available_features = set(data.columns) - set([label])
    assert len(available_features) > 0

    best_purity = 1
    best_feature = None
    best_th = None
    for f in available_features:
        purity, th = calculate_best_threshold(data, f, label, classes, purity_function)
        if purity < best_purity:
            best_purity = purity
            best_feature = f
            best_th = th

    return best_feature, best_th
```

This function is implemented to calculate the best feature for splitting. First, it creates a set called "available_features," which contains all the column names in the data except for the label column. Then, it examines to ensure that at least one feature is available for splitting.

Next, by calling the above functions, it checks all the features available in the sub-tree and returns the best categorical or numerical feature for splitting. Finally, the function returns a tuple containing the best feature and threshold value for the overall best split.

6 - Function for separating data:

```python
def split_data(data: pd.DataFrame, feature, threshold):
    sub_datas = {f">= {threshold}": data[data[feature] >= threshold],
                 f"< {threshold}": data[data[feature] < threshold]}

    return sub_datas
```

This function is implemented to separate a subtree based on the feature that was declared as the best feature for splitting in the previous functions. It separates all values less than and greater than the threshold into two different subtrees. Finally, it returns "sub_datas," which contains two subsets of data.

7 - Function for creating a decision tree:

```
def make_decision_tree(data: pd.DataFrame,
                       label='label',
                       classes=None,
                       return_prob=True,
                       purity_function=gini_index,
                       min_sample_split=0,
                       max_depth=np.inf):
    available_features = set(data.columns) - set([label])
    if len(available_features) == 0 or \
            len(data[label].unique()) == 1 or \
            len(data) < min_sample_split or \
            max_depth == 0:
        if return_prob:
            return json.loads((data[f'{label}'].value_counts() / len(data)).to_json())
        else:
            return {f'{label}': data[label].value_counts().idxmax()}

    split_feature, split_th = calculate_best_split(data, label, classes, purity_function)
    sub_datas = split_data(data, split_feature, split_th)

    if len(sub_datas.keys()) == 1:
        if return_prob:
            return json.loads((data[f'{label}'].value_counts() / len(data)).to_json())
        else:
            return {f'{label}': data[label].value_counts().idxmax()}

    results = (split_feature, {})
    for key in sub_datas.keys():
        results[1][key] = make_decision_tree(sub_datas[key],
                                             label,
                                             classes,
                                             return_prob=return_prob,
                                             purity_function=purity_function,
                                             min_sample_split=min_sample_split,
                                             max_depth=max_depth - 1
                                             )

    return results
```

This function is implemented to create and form a decision tree. When this function is called, it first calls the "calculate_best_split" function to find the best feature for splitting. Then, by calling the "split_data" function, it divides the current subtree into two different subtrees based on the best feature and threshold value found. This function recursively calls the creation of the decision tree on each of the subtrees. The function will be recursively called until it reaches a leaf where either the number of features in the subtree reaches 0 or the number of different labels in a node reaches 1.

Additionally, two new conditions have been added to the function to establish pre-pruning:

1. Minimum number of samples required to split (Min sample split):

• If the number of data points available in each subtree does not reach a desirable threshold specified in the function's input, the process of splitting that branch for pre-pruning is not performed.

2. Maximum depth of the tree (Max_depth):

• After reaching the maximum allowable depth for the decision tree, the process of splitting that branch for pre-pruning is not performed. After setting the input maximum depth, this function is executed recursively, and in each subtree, the depth is reduced by one until that variable reaches 0. The zero value means that this subtree was the last subtree to be split, and we are at the last allowable depth in the decision tree.

.3If a numerical feature has only one unique value, we cannot examine threshold values for that feature. Therefore, we do not check that feature for tree splitting.

8- Implementation of the prediction and execution function in the decision tree:

```python
def predict_tree(tree_json, label, data):
    if type(tree_json) == dict:
        if label in tree_json.keys():
            return pd.DataFrame(tree_json[label], index=data.index, columns=['predict'])
        else:
            return None

    feature, splits = tree_json
    # split.keys()[0] = '>= threshold'
    threshold = float(list(splits.keys())[0].split(' ')[1])
    sub_datas = split_data(data, feature, threshold)

    predictions = [predict_tree(splits[key], label, sub_datas[key]) for key in splits.keys()]
    predictions = predictions[0].append(predictions[1])
    predictions = predictions.sort_index()

    return predictions
```

This function is implemented for predicting the data that will be entered into the decision tree. If the data enters a leaf node, it should take the label of that leaf.

However, if it enters a node other than a leaf, recursively, it moves under the branches according to the path selected for it in the tree until it reaches a leaf. In other words, if that node is not a leaf, we extract the threshold value for that data and recursively apply this function to each subtree until all the data entered into the tree is predicted. Then, we assign a label to each data and place them together and sort them to align with the order of data in the dataset.

9- Implementation of the function for evaluating the results of the decision tree:

```
def score_tree(tree_json, label, data):
    classes = df[label].unique()
    h = predict_tree(tree_json, label, data)
    report = classification_report(data[label], h, output_dict=True)
    report['weighted avg']['accuracy'] = report['accuracy']
    report['macro avg']['accuracy'] = sum([accuracy_score(data[label] == cls, h == cls) for cls in classes]) / len(classes)
    report = {'macro avg': report['macro avg'], 'micro avg': report['weighted avg']}
    return report
```

This function is implemented to evaluate the trained model. After training the tree, it predicts the labels of the data by calling the previous function, which is predict_tree, and shows the performance of the model based on the metrics requested in the question:

- macro accuracy - micro accuracy
- macro precision - micro precision
- macro recall - macro recall
- f1-score

10- Implementation of the Post-pruning function:

```python
def post_pruning(tree_json,
                 label,
                 valid_df):
    if type(tree_json) == dict or len(valid_df) == 0:
        return tree_json

    tree_json = deepcopy(tree_json)
    base_score = score_tree(tree_json, label, valid_df)['micro avg']['f1-score']

    pruned_subtree = {f'{label}': valid_df[label].value_counts().idxmax()}
    pruned_score = score_tree(pruned_subtree, label, valid_df)['micro avg']['f1-score']

    if pruned_score > base_score:
        print('pruned...')
        return pruned_subtree
    else:
        feature, splits = tree_json
        threshold = float(list(splits.keys())[0].split(' ')[1])
        sub_datas = split_data(valid_df, feature, threshold)

        # tree_json[1] = branches
        for key in tree_json[1].keys():
            tree_json[1][key] = post_pruning(tree_json[1][key], label, sub_datas[key])

    return tree_json
```

This function is implemented to perform Post-pruning and avoid overfitting challenges. Note that only branches that the validation data passes through are pruned. To prune, we first sample the tree and make changes to the sampled tree. Then, we consider a path from the validation data. We intend to replace a leaf with a subtree on that path. To do this, we select a subtree and replace a leaf with it. The label of that leaf is determined from the data inside that subtree. To do this, by entering the validation data into that path, we measure the accuracy of that path after replacing the leaf with the subtree. If the accuracy on the validation data increases, we will replace that leaf with the subtree.

Note that after doing this, it is obvious that the accuracy of the model on the training data will decrease because the learned paths with the training data will change.

You can see the results of applying Post-pruning in section 3 or in the results analysis.

11- Implementation of the decision tree evaluation function and calculation of evaluation metrics:

```python
def evaluate_model(data,
                   purity_function=gini_index,
                   min_sample_split=0,
                   max_depth=np.inf,
                   num_runs=10,
                   postpruning=False):
    classes = data['target'].unique()
    total_report = {'macro avg': {'precision': {'train': 0, 'valid': 0},
                                  'recall': {'train': 0, 'valid': 0},
                                  'f1-score': {'train': 0, 'valid': 0},
                                  'accuracy': {'train': 0, 'valid': 0}},
                    'micro avg': {'precision': {'train': 0, 'valid': 0},
                                  'recall': {'train': 0, 'valid': 0},
                                  'f1-score': {'train': 0, 'valid': 0},
                                  'accuracy': {'train': 0, 'valid': 0}}}
    for i in range(num_runs):
        # train test split without sklearn or converting to numpy
        msk = np.random.rand(len(data)) < 0.7
        train_df = data[msk]
        valid_df = data[~msk]

        tree = make_decision_tree(train_df, 'target', classes, return_prob=False,
                                  purity_function=purity_function,
                                  min_sample_split=min_sample_split,
                                  max_depth=max_depth)
        if postpruning:
            tree = post_pruning(tree, 'target', valid_df)

        report_train = score_tree(tree, 'target', train_df)
        report_valid = score_tree(tree, 'target', valid_df)
        for avg_type in total_report.keys():
            for metric in total_report[avg_type].keys():
                total_report[avg_type][metric]['train'] += report_train[avg_type][metric] / num_runs
                total_report[avg_type][metric]['valid'] += report_valid[avg_type][metric] / num_runs

    total_report['model config'] = {'purity_function': purity_function.__name__,
                                    'min_sample_split': min_sample_split,
                                    'max_depth': max_depth,
                                    'post-pruning': postpruning}
    return total_report
```

Outside this code block and during execution, the dataset is divided into 90% for training data and 10% for testing data. Then, inside this function, the training data is split into 70% for training data and 30% for validation data.

This function evaluates the decision tree in the specified conditions for pre-pruning and post-pruning. The model is evaluated by this function 10 times with 10 different training-validation data splits to make the accuracy we evaluate from the model more precise and the average of all of these models.

12- Implementation of the Experiments class for evaluating and preparing the best model:

```
class Experiments:
    def __init__(self, data, num_runs=10):
        msk = np.random.rand(len(data)) < 0.9
        self.train_df = data[msk]
        self.test_df = data[~msk]

    def purity_function_experiment(self):
        report_gini = evaluate_model(self.train_df, purity_function=gini_index)
        report_entropy = evaluate_model(self.train_df, purity_function=entropy)

          #Plotting

        if report_gini['micro avg']['f1-score']['valid'] > report_entropy['macro avg']['f1-score']['valid']:
            return gini_index
        else:
            return entropy


    def max_depth_experiment(self, purity_function):
        reports_train = {'macro avg': {'accuracy': [], 'precision': [], 'recall': [], 'f1-score': []},
                         'micro avg': {'accuracy': [], 'precision': [], 'recall': [], 'f1-score': []}}
        reports_valid = {'macro avg': {'accuracy': [], 'precision': [], 'recall': [], 'f1-score': []},
                         'micro avg': {'accuracy': [], 'precision': [], 'recall': [], 'f1-score': []}}

        explore_range = [0, 21, 2]
        for max_depth in range(*explore_range):
            report = evaluate_model(self.train_df,
                                    purity_function=purity_function,
                                    max_depth=max_depth)
            print(report)
            for avg_type in reports_valid.keys():
                for metric in reports_valid[avg_type].keys():
                    reports_valid[avg_type][metric].append(report[avg_type][metric]['valid'])
                    reports_train[avg_type][metric].append(report[avg_type][metric]['train'])

             #Plotting

        return list(range(*explore_range))[
            reports_valid['micro avg']['f1-score'].index(max(reports_valid['micro avg']['f1-score']))]
```

This class is actually a Helper Class and helps us to execute all the steps requested from us in order. At each step, we will use the criterion that produces the best result in the next step to ultimately achieve the best specifications and results for the model. Initially, the dataset is split into 90% for training data and 10% for test data.

purity_function_experiment function:

We first train the model with different Split criteria, gini and entropy, and evaluate the results. Then, we visualize the results obtained from them on a chart with the 6 evaluation criteria requested. We will then repeat the criterion that achieves the best accuracy in the next steps

max_depth_experiment function:

We train the model with different values from 0 to 21 for the maximum allowable depth variable of the tree and evaluate the results. Then, we visualize the results obtained from them on a chart with the 6 evaluation criteria requested. We will then repeat the criterion that achieves the best accuracy in the next steps.

```
def min_samples_split_experiment(self, purity_function, max_depth):
    reports_train = {'macro avg': {'accuracy': [], 'precision': [], 'recall': [], 'f1-score': []},
                     'micro avg': {'accuracy': [], 'precision': [], 'recall': [], 'f1-score': []}}
    reports_valid = {'macro avg': {'accuracy': [], 'precision': [], 'recall': [], 'f1-score': []},
                     'micro avg': {'accuracy': [], 'precision': [], 'recall': [], 'f1-score': []}}

    explore_range = [0, 500, 50]
    for min_sample_split in range(*explore_range):
        report = evaluate_model(self.train_df,
                                purity_function=purity_function,
                                max_depth=max_depth,
                                min_sample_split=min_sample_split)
        print(report)
        for avg_type in reports_valid.keys():
            for metric in reports_valid[avg_type].keys():
                reports_valid[avg_type][metric].append(report[avg_type][metric]['valid'])
                reports_train[avg_type][metric].append(report[avg_type][metric]['train'])

        #Plotting

    return list(range(*explore_range))[
        reports_valid['micro avg']['f1-score'].index(max(reports_valid['micro avg']['f1-score']))]


def post_pruning_experiment(self, purity_function, max_depth, min_sample_split):
    report_normal = evaluate_model(self.train_df,
                                   purity_function=purity_function,
                                   max_depth=max_depth,
                                   min_sample_split=min_sample_split,
                                   postpruning=False)
    report_prune = evaluate_model(self.train_df,
                                  purity_function=purity_function,
                                  max_depth=max_depth,
                                  min_sample_split=min_sample_split,
                                  postpruning=True)

        #Plotting

    if report_normal['micro avg']['f1-score']['valid'] > report_prune['macro avg']['f1-score']['valid']:
        return False
    else:
        return True
```

min_sample_experiment function:

We train the model with different values from 0 to 500 for the maximum required data variable for the split and evaluate the results. Then, we visualize the results obtained from them on a chart with the 6 evaluation criteria requested. We will then repeat the criterion that achieves the best accuracy in the next steps.

post_pruning_experiment function:

We train the model once with post-pruning and once without post-pruning and evaluate the results. Then, we visualize the results obtained from them on a chart with the 6 evaluation criteria requested. We will then repeat the criterion that achieves the best accuracy in the next steps.

```
def sklearn_experiment(self):
    dt = DecisionTreeClassifier(criterion='gini',
                                max_depth=18)

    dt.fit(self.train_df.drop('target', axis=1), self.train_df['target'])
    h = dt.predict(self.test_df.drop('target', axis=1))

    classes = self.train_df['target'].unique()
    report = classification_report(self.test_df['target'], h, output_dict=True)
    report['macro avg']['accuracy'] = report['accuracy']
    report['weighted avg']['accuracy'] = sum([accuracy_score(self.test_df['target'] == cls, h == cls) for cls in classes]) / len(classes)
    report = {'macro avg': report['macro avg'], 'micro avg': report['weighted avg']}
    print('sklearn score:')
    print(report)

def run_experiments(self):
    purity_function = self.purity_function_experiment()
    max_depth = self.max_depth_experiment(purity_function)
    min_sample_split = self.min_samples_split_experiment(purity_function, max_depth)
    postpruning = self.post_pruning_experiment(purity_function, max_depth, min_sample_split)

    if postpruning:
        msk = np.random.rand(len(self.train_df)) < 0.7
        train_df = self.train_df[msk]
        valid_df = self.train_df[~msk]

        tree = make_decision_tree(train_df, 'target', classes=self.train_df['target'].unique(),
                                  return_prob=False, purity_function=purity_function,
                                  max_depth=max_depth, min_sample_split=min_sample_split)
        tree = post_pruning(tree, 'target', valid_df)
    else:
        tree = make_decision_tree(self.train_df, 'target', classes=self.train_df['target'].unique(),
                                  return_prob=False, purity_function=purity_function,
                                  max_depth=max_depth, min_sample_split=min_sample_split)

    print('My score')
    print(score_tree(tree, 'target', self.test_df))

    self.sklearn_experiment()
```

sklearn_experiment function:

In this function, the data along with the labels are entered into the decision tree implemented inside the Sklearn library to create and train the decision tree using this library. Finally, we perform the necessary evaluations on that decision tree so that we can compare the results obtained from it with our own decision tree and report the results.

run_experiments function:

This function is implemented to call the above implementation functions in the order requested in the question. If post-pruning is needed, it divides the validation data and then enters them in the model training process. However, if post-pruning is not required, we use all the training data to train the model, as this way the model will see more data and use them in the learning process.

13- Importing datasets and running decision tree:

```
df = pd.read_csv('HW1-Dataset.csv')
df = df.rename(columns={'Grade (target)': 'target'})

exp = Experiments(df)
exp.run_experiments()
```

After implementing the above functions, we read the dataset using the pandas library and after performing the necessary preprocessing, we call it along with the Experiments function. The Experiments function's task is to execute and iterate through the steps requested from us in the question. At each step, we evaluate the best criterion and use that criterion in the next test to achieve the best model results. Finally, we compare the best results of our implemented model with the model trained by the Sklearn library.

You can see the results obtained at each step in Chapter 3 or Results Analysis.

Analysis of Obtained Results:

In this report, we first reviewed the concepts of the decision tree algorithm and then explained the implementation functions. Now we come to the results and discussion section, where we present and analyze the results of the implemented models.

In this section, we will recap the questions asked and examine the results of implementing each section.

Implementation Questions:

Question 1:

The results of Kian's note-taking over ten consecutive weeks can be seen in the table below:

Sina's decision	Exam	Account balance	Weather condition	Week
Shopping	False	True	Sunny	1
Coffee shop	True	True	Sunny	2
Coffee shop	False	True	Snowy	3
Climbing	True	True	Sunny	4
Stay at home	True	True	Rainy	5
Climbing	False	False	Sunny	6
Climbing	False	True	Sunny	7
Coffee shop	True	True	Rainy	8
Shopping	True	True	Snowy	9
Stay at home	True	False	Snowy	10

1-1: Kian intends to help himself predict Sina's status on future Saturdays by implementing a simple decision tree.

1-2: Report the results in rule-based form.

Decision Tree and Answer 1-1:

The ID3 decision tree algorithm was used to solve this question, with the gini index used as the splitting criterion. After implementing and executing the decision tree algorithm using the functions reported in the relevant section, the resulting decision tree will be as follows:

```
('weather',
 {'rain': {'coffe': 0.5, 'stay': 0.5},
  'snow': ('cash',
           {'no': {'stay': 1.0},
            'yes': ('exam', {'no': {'coffe': 1.0}, 'yes': {'shop': 1.0}})}),
  'sun': ('exam',
          {'no': ('cash',
                  {'no': {'hiking': 1.0}, 'yes': {'hiking': 0.5, 'shop': 0.5}}),
           'yes': {'coffe': 0.5, 'hiking': 0.5}})})
```

As you can see, the probability of each decision is mentioned in each leaf. By initializing the "return_prob" variable in the input of the decision tree creation function, you can configure to see a decision or the probabilities of different decision-making by Sina inside each leaf. In this execution, this variable was initialized as "return_prob=True".

Reporting the results in Rule-based form and answering question 2-1:

As mentioned in the benefits section of the decision tree, decision-making by the decision tree algorithm is very close to human decision-making. After learning and decision-making by the decision tree on the evaluation dataset, the results can be reported in rule-based form, as explained in the implementation section.

You can see the results of these reports on the dataset that Kian had collected in the image below:

```
if weather is sun and if exam is no and if cash is yes then, prediction is shop or hiking
if weather is sun and if exam is no and if cash is no then, prediction is hiking
if weather is sun and if exam is yes then, prediction is coffe or hiking
if weather is snow and if cash is yes and if exam is no then, prediction is coffe
if weather is snow and if cash is yes and if exam is yes then, prediction is shop
if weather is snow and if cash is no then, prediction is stay
if weather is rain then, prediction is stay or coffe
```

For example:

If the weather is sunny, Sina has no exams, and his account balance is sufficient, Sina either goes shopping or mountain climbing.

If the weather is sunny, Sina has no exams, and he has no balance, Sina only goes mountain climbing.

Implementation Questions:

Question 2:

Use the following two criteria to select the best feature when constructing the tree. Does using different criteria affect the performance of the model? Choose the best criterion.

• Entropy

• Gini-index

Answer:

The criterion used to select the best feature when constructing the decision tree affects the performance of the model. The best criterion can be selected by comparing the performance of the model using different criteria. In this implementation, the Gini-index was used as the splitting criterion, but entropy can also be used depending on the dataset and the problem being solved.

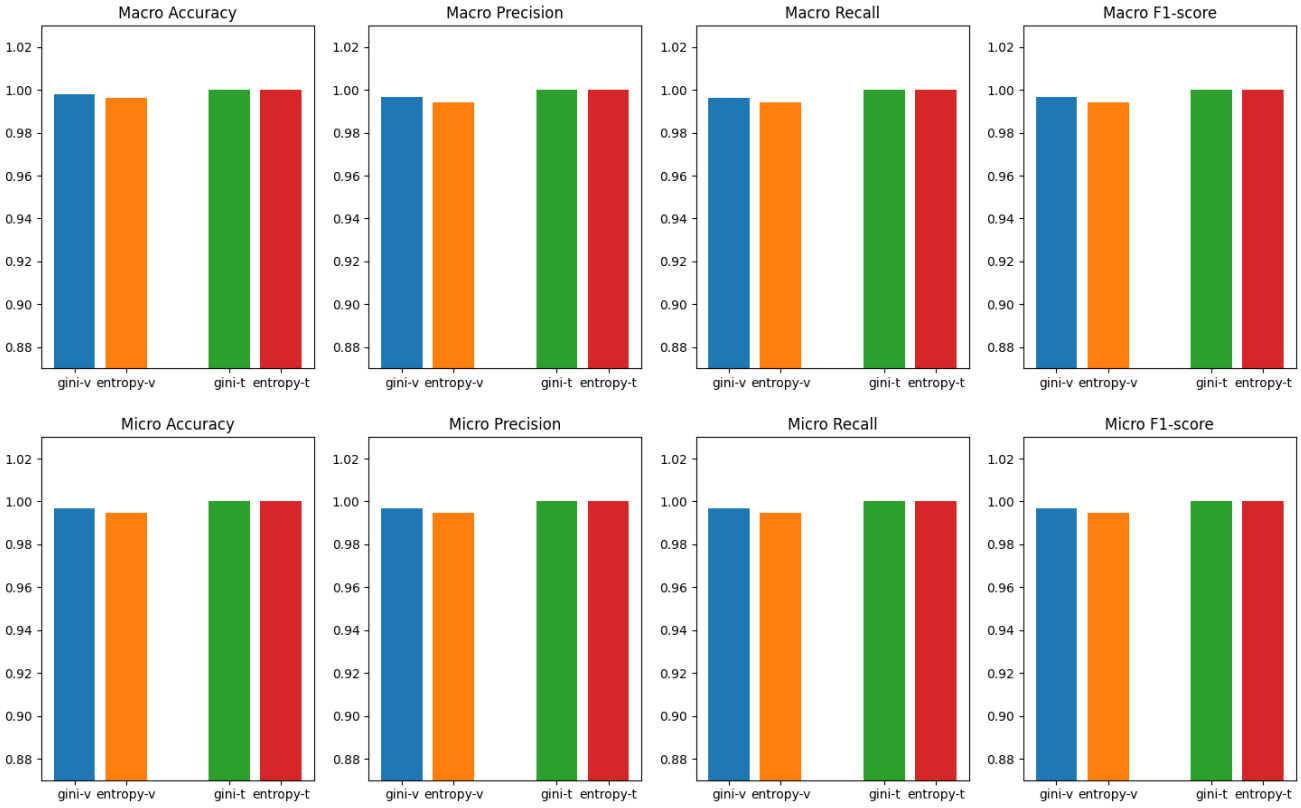
At the beginning of the report, we became familiar with the concepts of entropy and Gini criteria. The values of these criteria in the decision tree help with the method of branching or generalizing the branches. At each stage, the feature that has the lowest value of entropy or Gini compared to other examinable features is broken. A value of 0 for each feature means that by breaking this feature, we can completely separate the samples of each category after breaking the feature. Any feature that is broken at high stages or depths indicates that it had a high distinguishing capability.

$$I(S,A)=\sum_i \frac{|S_i|}{|S|}\cdot E(S_i) \quad Gini(S,A)=\sum_i \frac{|S_i|}{|S|}\cdot Gini(S_i)$$

To examine whether the use of different criteria has an impact on the model's performance, we call the model evaluation function with two different criteria, entropy and Gini, and print the results obtained on a chart.

```
report_gini = evaluate_model(self.train_df, purity_function=gini_index)
report_entropy = evaluate_model(self.train_df, purity_function=entropy)
```

The chart below shows the results obtained for the evaluation of these six criteria on the training and validation data separately:



As you can see in the above chart, choosing one of the two criteria, entropy or Gini, has no difference in the results obtained from the evaluation of these six criteria on the training data. However, the results on the validation data are variable. Using the Gini criterion on the validation data has resulted in better results. Therefore, to achieve the best results in the next stages, we will use the Gini criterion for decision tree learning.

Pre-pruning: By considering the following cases, perform pruning.

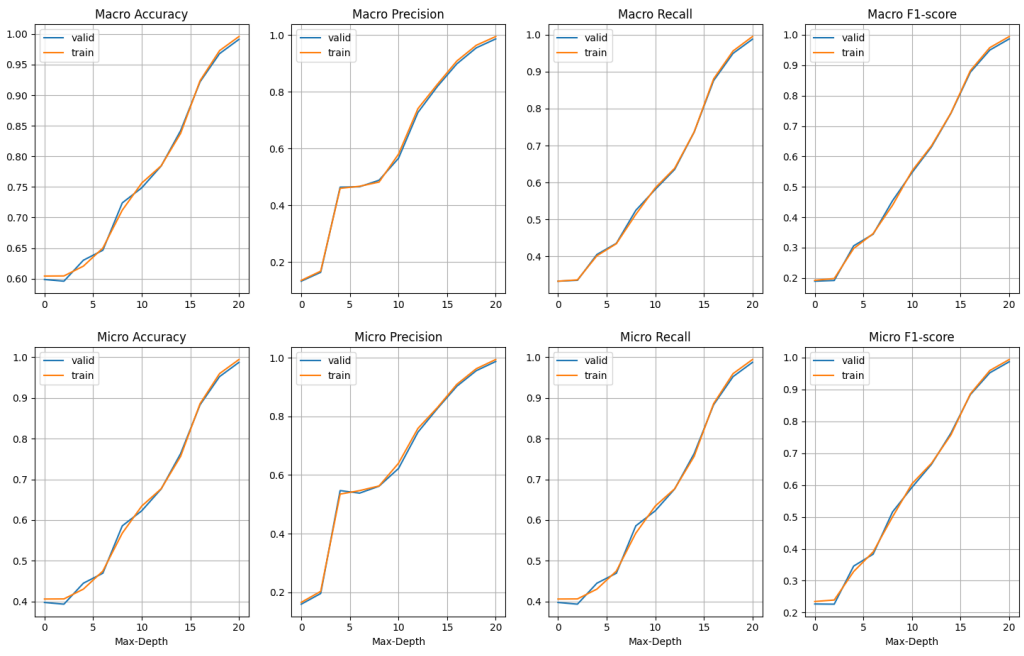
• Determine the maximum depth of the tree and do not continue building the tree after reaching the maximum depth.

• If the number of data entering a branch is negligible, do not break that branch. Does this technique have any effect on the model's performance? Report your experiment results.

```
explore_range = [0, 21, 2]
for max_depth in range(*explore_range):
    report = evaluate_model(self.train_df,
                            purity_function=purity_function,
                            max_depth=max_depth)
    print(report)
```

Examining the change in maximum depth of the tree in the accuracy of the results obtained:
To examine and observe the changes in the results obtained, we call the model evaluation function with different values for the maximum allowed depth of the tree. These values range from a maximum depth of 0 to 21.

The chart below shows the results obtained for the evaluation of the model on the training and validation data separately:



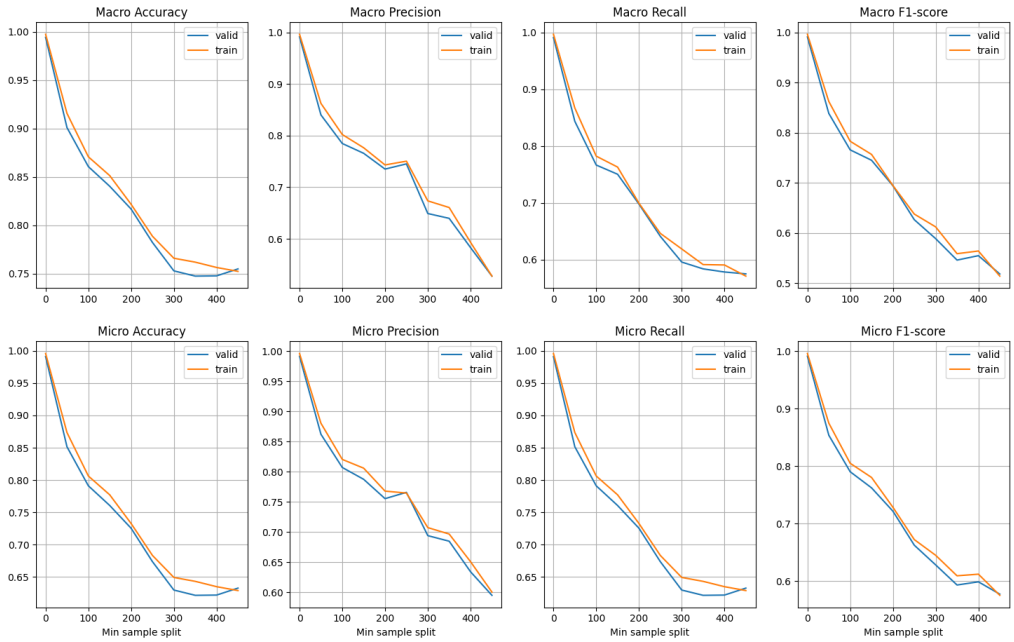
As you can see in the above chart, with the increase in the maximum allowed depth of the tree, high-accuracy results are obtained on both the training and validation data.

Important note: Based on the way the accuracy charts grow on the training and validation data, we realize that by increasing the depth of the tree up to a maximum depth of 20, the model has not suffered from overfitting challenges, and the accuracy on the validation data is also increasing. Therefore, to achieve the best results in the next stages, we will use the best value of the maximum depth of the tree for decision tree learning.

Examining the change in the minimum number of data entered into a branch for breaking it, in the accuracy of the results obtained:

```
explore_range = [0, 500, 50]
for min_sample_split in range(*explore_range):
    report = evaluate_model(self.train_df,
                            purity_function=purity_function,
                            max_depth=max_depth,
                            min_sample_split=min_sample_split)
    print(report)
```

The chart below shows the results obtained for the evaluation of the model on the training and validation data separately:



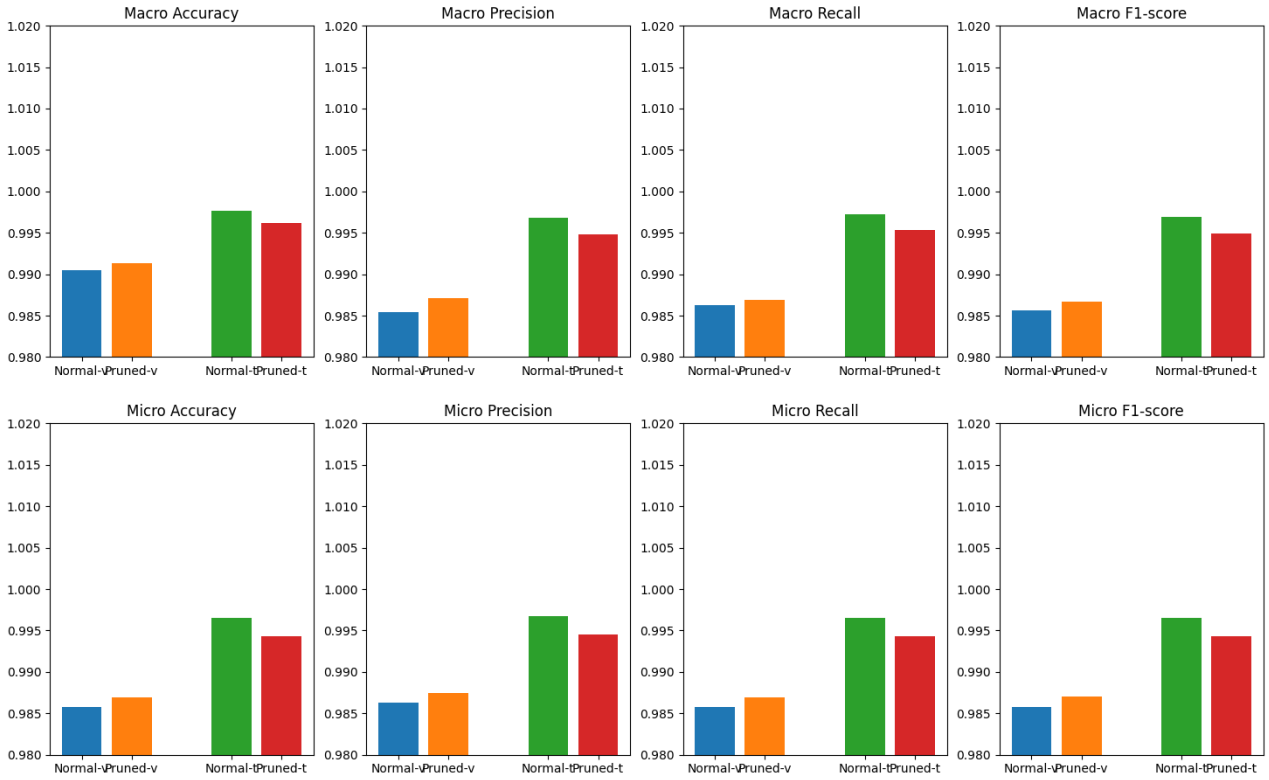
As you can see in the above image, increasing the minimum required data in each node for splitting leads to high accuracy results on both training and validation datasets. By increasing the minimum required data, we consider stricter rules for the decision tree algorithm as a node must have a large amount of data to be split into branches and generalized. In this case, the depth of the decision tree will decrease.

However, with the decrease in the depth of the tree, it will learn fewer rules for decision making and as a result, increasing the strictness of these rules will cause the model accuracy on both training and validation datasets to decrease, and the algorithm will face the challenge of underfitting.

Important note: As you can see in the charts, after the value of 300, the accuracy charts on the training and validation datasets diverge. This indicates the occurrence of the underfitting challenge in learning the decision tree. Therefore, to achieve the best results in the next stages, we will use the best value of this criterion for learning the decision tree.

Post-pruning: After building the decision tree completely, try to remove some of the subtrees. Does removing these subtrees affect the model performance? Report your experiment results.

The decision tree algorithm generally faces the challenge of overfitting. To prevent this from happening, we can use post-pruning. This pruning is done after the learning process is complete, and after that, the subtrees that have caused the model to overfit on the training data and decrease the accuracy on the validation data are pruned.



As you saw in the implementation section, in post-pruning, a leaf with a calculated label is first replaced by a subtree. If the model accuracy on the validation data increases, we will completely replace that leaf with the subtree.

In the chart below, you can see the results obtained after performing post-pruning on the training and validation datasets separately. As you can see, after performing post-pruning and removing the subtrees that caused the model to overfit on the training data, the model accuracy on the validation data has significantly increased.

It is obvious that after removing the branches that have been formed using the training data, the accuracy on the training data will decrease. Therefore, to achieve the best results in the next stages, we will use the best value of this criterion for learning the decision tree.

After performing the above steps, in this section, we will train a decision tree on the same dataset using the Scikit-learn library. By evaluating the model on the test dataset, we will demonstrate whether the results of the pre-built library differ from the results obtained in the previous sections. Analyze the reason for this difference.

After examining the above effective criteria and reaching the best learning conditions for the decision tree algorithm, we will train the model under the best conditions and compare its accuracy results with the model trained by the Sklearn library.

	Macro AVG					Micro AVG				
	Precision	Recall	F1-Score	Support	Accuracy	Precision	Recall	F1-Score	Support	Accuracy
My Score	0.9925926	0.9916667	0.9920353	106	0.9937107	0.9907757	0.990566	0.9905593	106	0.990566
Sklearn Score	1	1	1	106	1	1	1	1	106	1

You can see the results obtained on the test data in the chart above. As you can see in the results table, the results obtained from the implemented algorithm are very close to the results of the Sklearn library algorithm. As usual, when we see differences in performance between our machine learning algorithm implementation and existing libraries like Scikit-learn, there can be several reasons:

1. Difference in hyperparameter tuning: During this research activity, we tried to save the best results in the hyperparameters at each stage to use them in the next stage to achieve better results. However, as you know, the results obtained from machine learning algorithms are highly sensitive to hyperparameter tuning. Implementation algorithms in libraries like Sklearn use various techniques, including hyperparameter tuning, to achieve the best results, and the results of these algorithms are not comparable to the work done in this activity, and they will achieve better results for the model.

2. Difference in data preprocessing: Proper preprocessing of your data to achieve good performance with machine learning algorithms can be very important. If you preprocess your data differently from Scikit-learn, this can lead to differences in performance.

3. Difference in feature engineering: Feature engineering includes selecting and transforming features that are provided as input to your machine learning algorithm. If you use different features or changes in Scikit-learn, this can also lead to differences in performance.

4.Difference in implementation details: Machine learning algorithms can be implemented using different methods, and subtle differences in implementation can affect their performance.