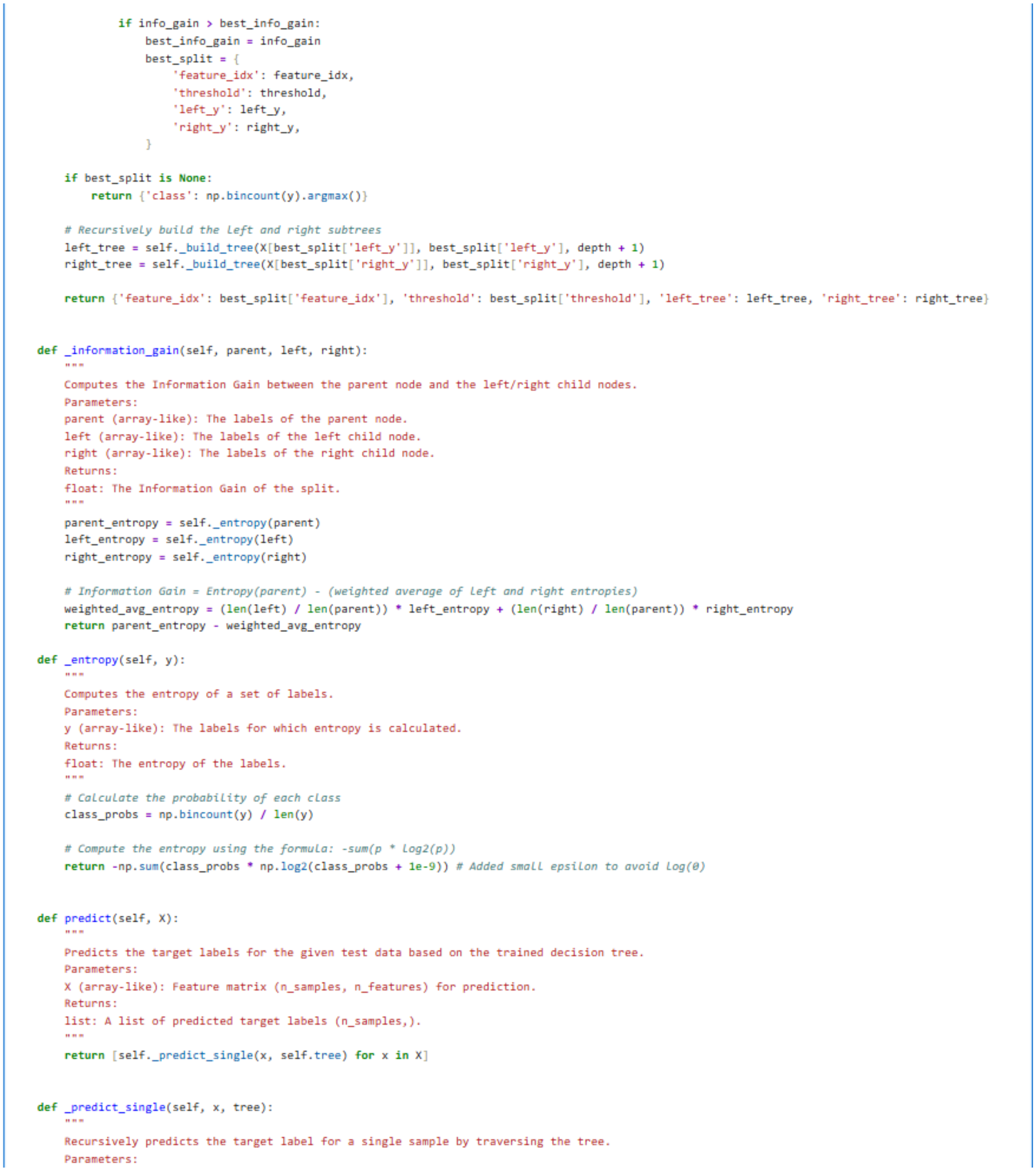
2. Custom Vs Scikit Learn Built Decision Tree

Step 1 : Building a Custom Decision Tree with Information Gain:

```
[6]: class CustomDecisionTree:
         def __init__(self, max_depth=None):
             """
             Initializes the decision tree with the specified maximum depth.
             Parameters:
             max_depth (int, optional): The maximum depth of the tree. If None, the tree is expanded until all
             leaves are pure or contain fewer than the minimum samples required to split.
             """
             self.max_depth = max_depth
             self.tree = None


         def fit(self, X, y):
             """
             Trains the decision tree model using the provided training data.
             Parameters:
             X (array-like): Feature matrix (n_samples, n_features) for training the model.
             y (array-like): Target labels (n_samples,) for training the model.
             """
             self.tree = self._build_tree(X, y)


         def _build_tree(self, X, y, depth=0):
             """
             Recursively builds the decision tree by splitting the data based on the best feature and threshold
             .
             Parameters:
             X (array-like): Feature matrix (n_samples, n_features) for splitting.
             y (array-like): Target labels (n_samples,) for splitting.
             depth (int, optional): Current depth of the tree during recursion.
             Returns:
             dict: A dictionary representing the structure of the tree, containing the best feature index,
             threshold, and recursive tree nodes.
             """
             num_samples, num_features = X.shape
             unique_classes = np.unique(y)

             # Stopping conditions: pure node or reached max depth
             if len(unique_classes) == 1:
                 return {'class': unique_classes[0]}
             if num_samples == 0 or (self.max_depth and depth >= self.max_depth):
                 return {'class': np.bincount(y).argmax()}

             # Find the best split based on Information Gain (using Entropy)
             best_info_gain = -float('inf')
             best_split = None
             for feature_idx in range(num_features):
                 thresholds = np.unique(X[:, feature_idx])
                 for threshold in thresholds:
                     left_mask = X[:, feature_idx] <= threshold
                     right_mask = ~left_mask
                     left_y = y[left_mask]
                     right_y = y[right_mask]

                     info_gain = self._information_gain(y, left_y, right_y)

                     if info_gain > best_info_gain:
                         best_info_gain = info_gain
                         best_split = {
                             'feature_idx': feature_idx,
                             'threshold': threshold,
```



```
                if info_gain > best_info_gain:
                    best_info_gain = info_gain
                    best_split = {
                        'feature_idx': feature_idx,
                        'threshold': threshold,
                        'left_y': left_y,
                        'right_y': right_y,
                    }

        if best_split is None:
            return {'class': np.bincount(y).argmax()}

        # Recursively build the left and right subtrees
        left_tree = self._build_tree(X[best_split['left_y']], best_split['left_y'], depth + 1)
        right_tree = self._build_tree(X[best_split['right_y']], best_split['right_y'], depth + 1)

        return {'feature_idx': best_split['feature_idx'], 'threshold': best_split['threshold'], 'left_tree': left_tree, 'right_tree': right_tree}


    def _information_gain(self, parent, left, right):
        """
        Computes the Information Gain between the parent node and the left/right child nodes.
        Parameters:
        parent (array-like): The labels of the parent node.
        left (array-like): The labels of the left child node.
        right (array-like): The labels of the right child node.
        Returns:
        float: The Information Gain of the split.
        """
        parent_entropy = self._entropy(parent)
        left_entropy = self._entropy(left)
        right_entropy = self._entropy(right)

        # Information Gain = Entropy(parent) - (weighted average of left and right entropies)
        weighted_avg_entropy = (len(left) / len(parent)) * left_entropy + (len(right) / len(parent)) * right_entropy
        return parent_entropy - weighted_avg_entropy

    def _entropy(self, y):
        """
        Computes the entropy of a set of labels.
        Parameters:
        y (array-like): The labels for which entropy is calculated.
        Returns:
        float: The entropy of the labels.
        """
        # Calculate the probability of each class
        class_probs = np.bincount(y) / len(y)

        # Compute the entropy using the formula: -sum(p * log2(p))
        return -np.sum(class_probs * np.log2(class_probs + 1e-9)) # Added small epsilon to avoid log(0)


    def predict(self, X):
        """
        Predicts the target labels for the given test data based on the trained decision tree.
        Parameters:
        X (array-like): Feature matrix (n_samples, n_features) for prediction.
        Returns:
        list: A list of predicted target labels (n_samples,).
        """
        return [self._predict_single(x, self.tree) for x in X]


    def _predict_single(self, x, tree):
        """
        Recursively predicts the target label for a single sample by traversing the tree.
        Parameters:
```

```
        return [self._predict_single(x, self.tree) for x in X]


    def _predict_single(self, x, tree):
        """
        Recursively predicts the target label for a single sample by traversing the tree.
        Parameters:
        x (array-like): A single feature vector for prediction.
        tree (dict): The current subtree or node to evaluate.
        Returns:
        int: The predicted class label for the sample.
        """
        if 'class' in tree:
            return tree['class']

        feature_val = x[tree['feature_idx']]
        if feature_val <= tree['threshold']:
            return self._predict_single(x, tree['left_tree'])
        else:
            return self._predict_single(x, tree['right_tree'])
```

Step 2 : Load and Split the Iris Datasets:

[7]:

```
# Necessary Imports
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score

# Load the Iris dataset
data = load_iris()
X = data.data
y = data.target

# Split into training and test sets (80% training, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

Step 3 : Train and Evaluate a Custom Decision Tree:

[8]:

```
# Train the custom decision tree
custom_tree = CustomDecisionTree(max_depth=3)
custom_tree.fit(X_train, y_train)

# Predict on the test set
y_pred_custom = custom_tree.predict(X_test)

# Calculate accuracy
accuracy_custom = accuracy_score(y_test, y_pred_custom)
print(f"Custom Decision Tree Accuracy: {accuracy_custom:.4f}")
```

```
Custom Decision Tree Accuracy: 0.8000
```

Step 4 : Train and Evaluate a Scikit Learn Decision Tree:

[9]:

```
# Train the Scikit-Learn decision tree
sklearn_tree = DecisionTreeClassifier(max_depth=3, random_state=42)
sklearn_tree.fit(X_train, y_train)

# Predict on the test set
y_pred_sklearn = sklearn_tree.predict(X_test)

# Calculate accuracy
accuracy_sklearn = accuracy_score(y_test, y_pred_sklearn)
print(f"Scikit-learn Decision Tree Accuracy: {accuracy_sklearn:.4f}")
```

```
Scikit-learn Decision Tree Accuracy: 1.0000
```

Step 5 : Result Comparision:

[10]:
```
print(f"Accuracy Comparison:")
print(f"Custom Decision Tree: {accuracy_custom:.4f}")
print(f"Scikit-learn Decision Tree: {accuracy_sklearn:.4f}")
```

```
Accuracy Comparison:
Custom Decision Tree: 0.8000
Scikit-learn Decision Tree: 1.0000
```

3. Exercise - Ensemble Methods and Hyperparameter Tuning:

1. Implement Classification Models:

Step 1: Loading and splitting the data

[23]:
```
from sklearn.datasets import load_wine
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import f1_score

# Load dataset
X, y = load_wine(return_X_y=True)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)
```

Step 2: Training Decision Tree Classifier:

[24]:
```
dt_clf = DecisionTreeClassifier(random_state=42)
dt_clf.fit(X_train, y_train)

y_pred_dt = dt_clf.predict(X_test)
f1_dt = f1_score(y_test, y_pred_dt, average='weighted')
```

Step 3: Training Random Forest Classifier:

[25]:
```
rf_clf = RandomForestClassifier(random_state=42)
rf_clf.fit(X_train, y_train)

y_pred_rf = rf_clf.predict(X_test)
f1_rf = f1_score(y_test, y_pred_rf, average='weighted')
```

Step 4: Comparing F1 Scores:

[26]:
```
print("Decision Tree F1 Score:", f1_dt)
print("Random Forest F1 Score:", f1_rf)
```

```
Decision Tree F1 Score: 0.9449614374099499
Random Forest F1 Score: 1.0
```

2. Hyperparameter Tuning:

Step 1: Identifying the three hyperparameters of Random Tree Classifier:

1. **n_estimators** - number of trees
2. **max_depth** - maximum tree depth
3. **min_sample_split** - minimum samples to split a node

Step 2: Using GridSearchCV to optimize these hyperparameters:

[27]:

```
from sklearn.model_selection import GridSearchCV

param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [None, 5, 10],
    'min_samples_split': [2, 5, 10]
}

grid_search = GridSearchCV(
    estimator=RandomForestClassifier(random_state=42),
    param_grid=param_grid,
    scoring='f1_weighted',
    cv=5,
    n_jobs=-1
)

grid_search.fit(X_train, y_train)

best_rf_clf = grid_search.best_estimator_

print("Best Parameters:", grid_search.best_params_)

y_pred_best = best_rf_clf.predict(X_test)
print("Optimized F1 Score:", f1_score(y_test, y_pred_best, average='weighted'))
```

```
Best Parameters: {'max_depth': None, 'min_samples_split': 2, 'n_estimators': 50}
Optimized F1 Score: 1.0
```

3. Implement Regression Model:

Step 1: Preparing Regression Data:

[28]:

```
from sklearn.tree import DecisionTreeRegressor
from sklearn.ensemble import RandomForestRegressor
from sklearn.metrics import mean_squared_error

# Use alcohol feature as target
X_reg = X[:, 1:]
y_reg = X[:, 0]

Xr_train, Xr_test, yr_train, yr_test = train_test_split(X_reg, y_reg, test_size = 0.2, random_state = 42)
```

Step 2: Training Decision Tree Regressor:

[29]:

```
dt_reg = DecisionTreeRegressor(random_state=42)
dt_reg.fit(Xr_train, yr_train)

pred_dt_reg = dt_reg.predict(Xr_test)
mse_dt = mean_squared_error(yr_test, pred_dt_reg)
```

Step 3: Training Random Forest Regressor:

[30]:

```
rf_reg = RandomForestRegressor(random_state=42)
rf_reg.fit(Xr_train, yr_train)

pred_rf_reg = rf_reg.predict(Xr_test)
mse_rf = mean_squared_error(yr_test, pred_rf_reg)

print("Decision Tree MSE:", mse_dt)
print("Random Forest MSE:", mse_rf)
```

```
Decision Tree MSE: 0.31197222222222226
Random Forest MSE: 0.15426672999999946
```

Step 4: Hyperparameter Tuning using RandomSearchCV

[31]:

```
from sklearn.model_selection import RandomizedSearchCV

param_dist = {
    'n_estimators': [50, 100, 200, 300],
    'max_depth': [None, 5, 10, 20],
    'min_samples_leaf': [1, 2, 4]
}

random_search = RandomizedSearchCV(
    estimator=RandomForestRegressor(random_state=42),
    param_distributions=param_dist,
    n_iter=10,
    scoring='neg_mean_squared_error',
    cv=5,
    random_state=42,
    n_jobs=-1
)

random_search.fit(Xr_train, yr_train)

best_rf_reg = random_search.best_estimator_

pred_best_rf = best_rf_reg.predict(Xr_test)
mse_best = mean_squared_error(yr_test, pred_best_rf)

print("Best Parameters:", random_search.best_params_)
print("Optimized Random Forest MSE:", mse_best)
```

```
Best Parameters: {'n_estimators': 200, 'min_samples_leaf': 1, 'max_depth': 10}
Optimized Random Forest MSE: 0.151070652463534
```