Custom Built Decision Tree:

```
import numpy as np
```

```
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()}
```

```
    # 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,
            'left_mask': left_mask,
            'right_mask': right_mask,
          }

    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_mask']], y[best_split['left_mask']], depth + 1)
    right_tree = self._build_tree(X[best_split['right_mask']], y[best_split['right_mask']], 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)
```

```
    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:
    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'])
```

```
        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'])
```

Load and Split the IRIS Dataset:

```
import numpy as np
```

```
import pandas as pd
```

```
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
```

```
data = load_iris()
```

```
X = data.data
```

```
y = data.target
```

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

Train and Evaluate a Custom Decision Tree:

```
# 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: 1.0000
```

Train and Evaluate a Scikit Learn Decision Tree:

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

DecisionTreeClassifier

```
DecisionTreeClassifier(max_depth=3, random_state=42)
```

```
# 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
```

Result Comparision:

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

Exercise - Ensemble Methods and Hyperparameter Tuning.

1. Implement Classification Models:

```
from sklearn.datasets import load_wine
```

```
from sklearn.model_selection import train_test_split
```

```
from sklearn.tree import DecisionTreeClassifier
```

```
from sklearn.ensemble import RandomForestClassifier
```

```
from sklearn.metrics import f1_score
```

```
data = load_wine()
```

```
X = data.data
```

```
y = data.target
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)
```

```
dt_clf = DecisionTreeClassifier(random_state=42)
```

```
dt_clf.fit(X_train, y_train)
```

```
dt_clf = DecisionTreeClassifier(random_state=42)
```

```
dt_clf.fit(X_train, y_train)
```

```
DecisionTreeClassifier
DecisionTreeClassifier(random_state=42)
```

```
dt_preds = dt_clf.predict(X_test)
```

```
rf_clf = RandomForestClassifier(random_state=42)
```

```
rf_clf.fit(X_train, y_train)
```

```
RandomForestClassifier
RandomForestClassifier(random_state=42)
```

```
rf_preds = rf_clf.predict(X_test)
```

```
dt_f1 = f1_score(y_test, dt_preds, average='weighted')
```

```
rf_f1 = f1_score(y_test, rf_preds, average='weighted')
```

```
print(f"Decision Tree F1 Score : {dt_f1:.4f}")
```

```
Decision Tree F1 Score : 0.9628
```

```
print(f"Random Forest F1 Score : {rf_f1:.4f}")
```

```
Random Forest F1 Score : 1.0000
```

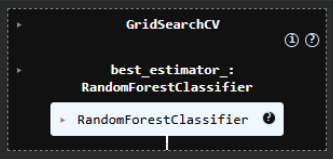
2. Hyperparameter Tuning:

```
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, cv=5, scoring='f1_weighted', n_jobs=-1)
```

```
grid_search.fit(X_train, y_train)
```




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

```
Best Parameters: {'max_depth': None, 'min_samples_split': 2, 'n_estimators': 50}
```

```
print("Best F1 Score:", grid_search.best_score_)
```

```
Best F1 Score: 0.9680809081527346
```

3. Implement Regression Model:

```
from sklearn.tree import DecisionTreeRegressor
```

```
from sklearn.ensemble import RandomForestRegressor
```

```
from sklearn.metrics import mean_squared_error
```

```
dt_reg = DecisionTreeRegressor(random_state=42)
dt_reg.fit(X_train, y_train)
dt_reg_preds = dt_reg.predict(X_test)
```

```
rf_reg = RandomForestRegressor(random_state=42)
rf_reg.fit(X_train, y_train)
rf_reg_preds = rf_reg.predict(X_test)
```

```
dt_mse = mean_squared_error(y_test, dt_reg_preds)
```

```
rf_mse = mean_squared_error(y_test, rf_reg_preds)
```

```
print("Decision Tree MSE:", dt_mse)
```

```
Decision Tree MSE: 0.14814814814814814
```

```
print("Random Forest MSE:", rf_mse)
```

```
Random Forest MSE: 0.057238888888888916
```

```
from sklearn.model_selection import RandomizedSearchCV
```

```
import numpy as np
```

```
param_dist = {'n_estimators': [50, 100, 200, 300], 'max_depth': [None, 5, 10, 20], 'min_samples_leaf': [1, 2, 4, 6]}
```

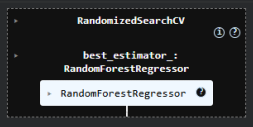
```
from sklearn.model_selection import RandomizedSearchCV
```

```
import numpy as np
```

```
param_dist = {'n_estimators': [50, 100, 200, 300], 'max_depth': [None, 5, 10, 20], 'min_samples_leaf': [1, 2, 4, 6]}
```

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

```
random_search.fit(X_train, y_train)
```




```
print("Best Parameters:", random_search.best_params_)
```

```
Best Parameters: {'n_estimators': 50, 'min_samples_leaf': 1, 'max_depth': None}
```

```
print("Best MSE:", -random_search.best_score_)
```

```
Best MSE: 0.04628640000000005
```