INF264 Obligatory Assignment 1

Brage L. Aasen

September 2024

1 Introduction

In this project I created decision tree and random forest classifiers from scratch. The learning algorithm I implemented was the Iterative Dichotomiser 3 (ID3) algorithm, which can use either entropy or Gini index as the impurity measure.

After creating the classifiers, I evaluating the implementations on real datasets, and did model selection to optimize the classifiers to the datasets. Lastly, I compared my implementations of the decision tree and random forest classifiers to existing library implementations from sklearn.

1.1 Data Analysis

For the analysis of data, I used matplotlib.pyplot and numpy to visualize the datasets.

1.1.1 Wine dataset

The features available in the Wine dataset are as follows:

- Citric acid
- Residual sugar
- pH
- Sulphates
- Alcohol

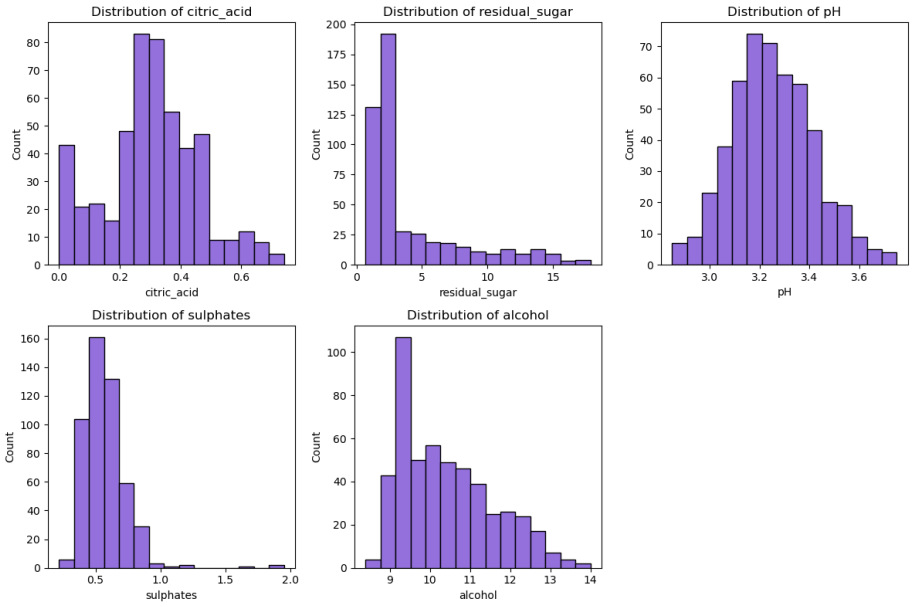

Figure 1: Wine Features

The wine plotting of the features distribution tells us about the variations of data within each feature. We can tell from the feature plots that some features have outliers such at sulphates and citric acid. We can also gather information such as wines with lower alcohol percentages are more common than those with higher ones.

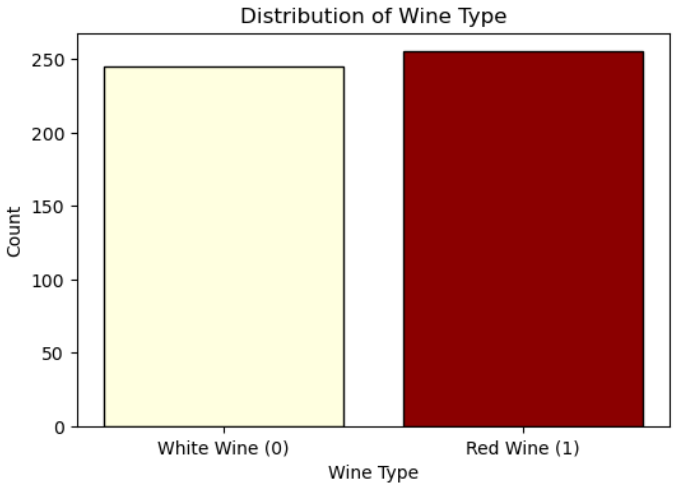

Figure 2: Wine Types

After plotting the wine type distribution, it is clear that there is a close to equal amount of white wine classification labels compared to red wine labels. Since both classes contributes approximately equally to the accuracy, the accuracy performance measure could be a viable choice.

1.1.2 Coffee dataset

The features available in the Coffee dataset are as follows:

- Aroma
- Flavor
- Aftertaste
- Acidity
- Body
- Balance
- Uniformity
- Sweetness

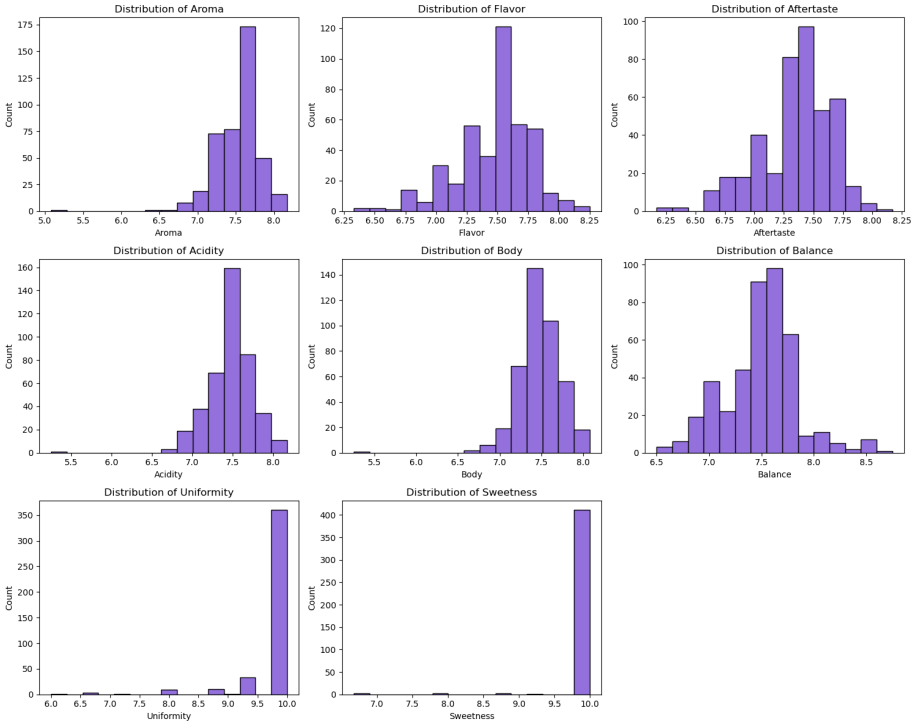

Figure 3: Coffee Features

The coffee plotting of the features distribution also tells us about the variations of data within each feature. Unlike the previous dataset, we can tell from the coffee plotting that there are many outliers in the different features. By

looking at the distributions, we can also make assumption's such as the coffee beans present in the dataset being sweet and acidic.

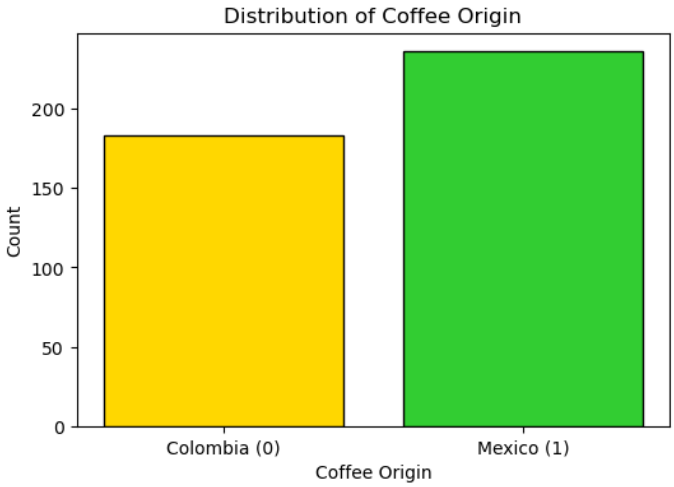

Figure 4: Coffee Origin

By plotting the coffee origin distribution, it is obvious that there exist somewhat of an imbalance in class samples. Specifically, Colombia has less class samples compared to Mexico. An imbalance like this has the potential to impact model performance.

1.2 Data Preprocessing

I chose not to do any data preprocessing when solving this obligatory assignment, and instead use the raw data provided. The provided datasets was already well-structured without any missing values. In addition to this, the outliers that I observed in the data analysis stage were not extreme enough to warrant alterations to the datasets. Another factor that made me skip data preprocessing, is that the decision tree model can handle unscaled features, e.i, feature normalization was not be necessary.

2 Implementations

2.1 Decision Trees

My implementation of a decision tree classifier builds a model by recursively splitting the data based on the best features and corresponding threshold to split on. The decision tree aims to minimize the impurity of the data based on a metric chosen by the user (either entropy or Gini index).

The recursion is stopped by base cases which returns leaf nodes with a classification.

These base cases are as follows:

- When all data points at a node have the same label, the recursion stops, and a leaf node with that label is returned.
- When all data points have identical feature values, the recursion stops, and a leaf node with the most common label is returned.
- If the maximum depth of the tree is specified and reached, the recursion stops, and a leaf node with the most common label is returned.

The main methods of my decision tree are:

1. `fit(self, X: np.ndarray, y: np.ndarray, current_depth: int = 0)`
2. `get_best_split(self, X: np.ndarray, y: np.ndarray)`
3. `calculate_information_gain(self, y_parent: np.ndarray, y_left: np.ndarray, y_right: np.ndarray) -> float`
4. `calculate_median_split(self, X: np.ndarray, y: np.ndarray, feature_index: int) -> tuple`
5. `predict(self, X: np.ndarray) -> np.ndarray`
6. `entropy(y: np.ndarray) -> float`
7. `gini_index(y: np.ndarray) -> float`

The fit method is the primary function responsible for training the model, e.i, fitting the decision tree as explained earlier.

The fit function are as follows:

```
def fit(self, X: np.ndarray, y: np.ndarray, current_depth: int = 0):
    """
    Fit the decision tree to the training data.
    """
    # Initialize the random seed
    np.random.seed(self.random_state)

    # Fit on the root
    if current_depth == 0:
        self.root = self.fit(X, y, current_depth=1)

    ## Check base cases

```

```
    # Return a leaf node with given label, when all data points have the same label
    if len(np.unique(y)) == 1:
        return Node(value=y[0])

    # Return leaf node with most common label, when all data points have identical feature values
    if np.all(X == X[0]):
        return Node(value=most_common(y))

    if self.max_depth is not None and current_depth >= self.max_depth: # Check if stopping condition is met
        return Node(value=most_common(y)) # Ensure a leaf node is returned, e.g not a None "node"


    # Get best feature and split, and use it to make the tree nodes
    best_feature, best_threshold, best_split_left, best_split_right = self.get_best_split(X, y)

    # Split data
    X_left, y_left = best_split_left
    X_right, y_right = best_split_right
    # If either split is empty, handle this case
    if len(X_left) == 0 or len(X_right) == 0:
        return Node(value=most_common(y))

    # Recursive call for left subtreee
    left_subtree = self.fit(X_left, y_left, current_depth + 1)

    # Recursive call for right subtree
    right_subtree = self.fit(X_right, y_right, current_depth + 1)

    return Node(feature=best_feature, threshold=best_threshold, left=left_subtree, right=right_subtree)
```

Listing 1: Decision Tree fit function

After the base cases are checked, the function gets the best feature split and uses it to make the tree nodes (decision nodes). Here, the best_feature, best_threshold, best_split_left and best_split_right are returned from the get_best_split(X, y) function.

get_best_split(X, y) is a function that iterates over a set of features to consider, the hyperparameter max_features in the decision tree constructor is the parameter that decides if and how to split the features (decided by the user whether to use all features, random subset of square root of the features or random subset of log2 of the features).

The splitting of features is done for the ensemble learning method used in section 2 (My implementation of Random Forest classifier), where the features are split to:

- **Reduce Overfitting:** In a single decision tree, overfitting can occur if the tree becomes too complex, such as when `max_depth` is too large. This can lead to the model capturing noise in the training data. By using a

random subset of features for each split, each tree in the Random Forest is less likely to overfit, as it operates with a limited scope over the entire feature space.

- **Explore More Aspects of the Data:** By providing each tree with a random subset of feature values to consider, different trees will explore various aspects of the data. This approach can uncover patterns between features that might have been missed if all features were used in every tree.
- **Reduce Redundant Features:** If the dataset contains highly correlated features, using all of them may introduce redundancy rather than adding more informative value. Randomly selecting subsets of features helps to focus on more diverse sets of features, reducing redundancy and improving the model's performance.

The get_best_split function are as follows:

```
def get_best_split(self, X: np.ndarray, y: np.ndarray):
    """
    Find the best feature and threshold to split the data to
        maximize information gain.
    """
    # Inits for the best values
    best_feature = None
    best_threshold = None
    best_information_gain = -float("inf")
    best_split_left = None
    best_split_right = None

    # Choose what features to consider for the split
    features_to_consider = None
    if self.max_features is None: # Use all features
        features_to_consider = np.arange(X.shape[1])
    elif self.max_features == "sqrt": # Use a random subset of sqrt
         of the features
        feature_subset_size = int(np.floor(np.sqrt(X.shape[1])))
        features_to_consider = self.rng.choice(np.arange(X.shape
            [1]), feature_subset_size, replace=False)
    elif self.max_features == "log2": # Use a random subset of log2
         of the features
        feature_subset_size = int(np.floor(np.log2(X.shape[1])))
        features_to_consider = self.rng.choice(np.arange(X.shape
            [1]), feature_subset_size, replace=False)

    # Loop over all features
    for feature_index in features_to_consider:
        X_left, y_left, X_right, y_right = self.
            calculate_median_split(X, y, feature_index)

        information_gain = self.calculate_information_gain(y,
            y_left, y_right)

        if information_gain > best_information_gain:
            best_feature = feature_index
```

```
            best_threshold = np.median(X[:, best_feature])
            best_split_left = (X_left, y_left)
            best_split_right = (X_right, y_right)
            best_information_gain = information_gain

    return best_feature, best_threshold, best_split_left,
        best_split_right
```

Listing 2: Decision Tree get_best_split function

After considering which features to consider for the split, the function get_best_split(X, y) begins the iterations over the given set of features. For each individual feature, we need to calculate the median split. We calculate the median split to find a potential optimal threshold for splitting the data, based on the feature's values. The function that does this, is calculate_median_split(X, y, feature_index).

```
def calculate_median_split(self, X: np.ndarray, y: np.ndarray,
    feature_index: int) -> tuple:
    """
    Calculate the median split on the features
    """
    # Median feature value
    features = X[:, feature_index]
    median = np.median(features)

    # Split
    left_boolean_mask = split(features, median)
    right_boolean_mask = np.logical_not(left_boolean_mask)

    X_left, y_left = X[left_boolean_mask], y[left_boolean_mask]
    X_right, y_right = X[right_boolean_mask], y[right_boolean_mask]

    return X_left, y_left, X_right, y_right
```

Listing 3: Decision Tree calculate_median_split function

calculate_median_split(X, y, feature_index) returns the given median split dataset X_left, y_left, X_right and y_right. These are new datasets where one contains values less than or equal to the median, whilst the other contains values greater than the median. It is with these sets we need to calculate the information gain, before we decide to use this specific split and threshold for our decision node. The information gain is calculated by the function calculate_information_gain(y_parent, y_left, y_right).

```
def calculate_information_gain(self, y_parent: np.ndarray, y_left:
    np.ndarray, y_right: np.ndarray) -> float:
    """
    Calculate the information gain on the current split
    """

    # Calculate weights
    weight_l = len(y_left) / len(y_parent)
    weight_r = len(y_right) / len(y_parent)

```

```
    if self.criterion == "entropy":
        # Use entropy -> (parent entropy - weighted average child entropy)
        information_gain = entropy(y_parent) - (weight_l * entropy(y_left) + weight_r * entropy(y_right))
    elif self.criterion == "gini":
        # Use Gini index
        information_gain = gini_index(y_parent) - (weight_l * gini_index(y_left) + weight_r * gini_index(y_right))

    return information_gain
```

Listing 4: Decision Tree calculate_information_gain function

The get_best_split(X, y) function keeps track of every current best splits information, e.i:

sbest_feature best_threshold best_information_gain best_split_left best_split_right

These variables are updated every time the calculation of information gain finds a new best split to use for the decision node. The information gain function calculates how much impurity the data is reduced by the given split. Here, a higher information gain will indicate that the input split is a better data split than previously considered splits.

The function uses either entropy or Gini index (decided by the user as a hyperparameter) to measure the impurity of the data. It compares the impurity of the parent node, with the weighted average impurity of the child nodes. By doing this calculation, we get the resulting change of information gain after the split. This change is what indicate whether or not the split has improved the purity of the data.

After finding the split with the best information gain, the get_best_split(X, y) function can finally return a decision node with the given split, feature and threshold to use.

Lastly, I would like to walk through the predict(X) with its corresponding predict_sample(sample, node) helper function.

```
def predict(self, X: np.ndarray) -> np.ndarray:
    """
    Given a NumPy array X of features, return a NumPy array of predicted integer labels.
    """
    predictions = np.array([self.predict_sample(sample, self.root) for sample in X])
    return predictions
```

Listing 5: Decision Tree predict function

The predict function is used to make the trained decision tree be able to make predictions on new and unseen data. It predicts the class labels for all the samples in X, by calling the helper function predict_sample to traverse the tree, and determine the predicted label.

This is an important functionality for our decision tree to have, because it gives us a way to classify new trees.

```
def predict_sample(self, sample, node):
    ## Check base case
    if node.is_leaf():
        return node.value

    # Recursive call
    if sample[node.feature] <= node.threshold:
        return self.predict_sample(sample, node.left)
    else:
        return self.predict_sample(sample, node.right)
```

Listing 6: Decision Tree predict_sample function

The predict sample function works by first checking if the current node is a leaf, if yes, return its value. If the node is not a leaf, e.i, it is a decision node, recursively make a prediction to the left and right.

There are other smaller functions which provides necessary functionality to my decision tree, but I choose to not go into detail on these.

2.2 Random Forest

My custom random forest classifier uses the implementation of my decision tree to create a forest of different trees. This classifier works as an important ensemble learning method, because it makes use of multiple decision trees to improve the overall generalization and performance of my model.

As discussed in the lectures, a single decision tree can be prone to overfitting, which is especially common when the model becomes too complex. By the use of multiple decision trees, where each tree is built on a random subset of data and features, the model will be less likely to overfit.

In addition to this, the random forest classifier gives us a better predictive accuracy by averaging (regression) or taking the majority votes (classification, our use-case) from multiple trees. Even when individual trees make mistakes, the majority will decide the outcome with more accurate decisions.

Lastly, given that features sometimes are correlated, the addition of random forest with random feature subsets, ensure more diverse feature splits (which was discussed earlier in my decision tree implementation).

The main methods of my decision tree are:

1. `fit(self, X: np.ndarray, y: np.ndarray)`
2. `predict(self, X: np.ndarray)`

Once again, the fit method is the primary function responsible for training the model, e.i, fitting the random forest.

The fit function are as follows:

```python
def fit(self, X: np.ndarray, y: np.ndarray):
    for _ in range(self.n_estimators): # Create the forest
        # Initialize the random seed for each tree
        np.random.seed(self.random_state)

        # Sample random subset (indicies)
        random_data_subset = self.rng.choice(np.arange(X.shape[0]),
             X.shape[0], replace=True)

        X_subset = X[random_data_subset]
        y_subset = y[random_data_subset]

        # Create the tree
        tree = DecisionTree(max_depth=self.max_depth, max_features=
            self.max_features, criterion=self.criterion)

        # Fit the tree
        tree.fit(X_subset, y_subset)

        self.trained_trees.append(tree)
```

Listing 7: Random Forest fit function

In this function, we iterate through n_estimators, and train a decision tree in every iteration. For each tree, a random data subset is made by the use of self.rng.choice, with replacement (bootstrapping). The decision trees are given the hyperparameters of the random forest class' parameters. The decision trees are then fitted, and appended to a list of trained trees.

The predict function is the next main function of my random forest classifier implementation, and looks like this:

```python
def predict(self, X: np.ndarray) -> np.ndarray:
    # Get the predictions of every tree
    tree_predictions = np.array([tree.predict(X) for tree in self.
        trained_trees])

    # Take the majority vote across all trees
    majority_vote = []
    for i in range(tree_predictions.shape[1]): # Look at each
        sample's predictions
        votes = tree_predictions[:, i] # Collect predictions
        majority_vote.append(np.bincount(votes).argmax()) # Find
            the most common vote for this tree

    majority_vote = np.array(majority_vote)
    return majority_vote
```

Listing 8: Random Forest predict function

This function starts by gathering the predictions from each individual trained decision tree in the forest. For each of the predicted sample, it finds the most common vote for that tree, and appends the majority vote to a list. After all of the majority votes are determined, it returns the result as a numpy array.

When we combine all of the predictions from multiple decision trees, we can reduce the chance of making incorrect predictions like discussed earlier.

3 Model Selection and Evaluation

3.1 Model Selection Process

For the model selection process, I defined a function that would be able to tune both the decions tree, and the random forest classifier. The function is as follows:

```
def tune_hyperparameters(model_class: DecisionTree |
    DecisionTreeClassifier | RandomForest | RandomForestClassifier,
     parameter_grid):
    best_average_accuracy = 0
    best_hyperparameters = {}

    for params in parameter_grid:
        accuracies = []

        # Perform k-fold cross-validation
        for train_index, val_index in kf.split(X_train_and_val):
            X_train, X_val = X_train_and_val[train_index],
                X_train_and_val[val_index]
            y_train, y_val = y_train_and_val[train_index],
                y_train_and_val[val_index]

            model = model_class(**params)

            # Train the tree
            model.fit(X_train, y_train)

            # Make predictions on the validation fold
            y_pred = model.predict(X_val)

            # Compute accuracy
            accuracy = accuracy_score(y_val, y_pred)
            accuracies.append(accuracy)

        # Calculate the mean accuracy across all folds
        mean_accuracy = np.mean(accuracies)

        # Only keep the best accuracy and hyperparameters
        if mean_accuracy > best_average_accuracy:
            best_average_accuracy = mean_accuracy
            best_hyperparameters = params

    return best_hyperparameters, best_average_accuracy
```

Listing 9: Tuning the hyperparameters function

3.1.1 Hyperparameters, Reproducibility and Fairness

The hyperparameters I chose to tune for the decision tree was:

`max_depth:`	Setting the maximum depth of each tree.
`criterion:`	How to measure the quality of a split (either "entropy" or "gini").

And the hyperparameters I chose to tune for the random forest was:

`n_estimators:`	The number of trees in the forest.
`max_depth:`	Setting the maximum depth of each tree.
`max_features:`	The number of features to consider when searching for the best split.
`criterion:`	How to measure the quality of a split (either "entropy" or "gini").

The values I chose to consider for the different hyperparameters was as follows:

```
## Hyperparameters
criterion_values = ["gini", "entropy"]

# Decision Tree
dt_max_depth_values = [3, 5, 8, 10, 15, None]

# Random Forest
rf_n_estimators_values = [5, 10, 15, 30]
rf_max_depth_values = [3, 5, 8, 10, 15, None]
rf_max_features_values = ["sqrt", "log2", None]
```

Listing 10: Hyperparameters to consider

I wanted enough different values for the classifiers to explore many different variations, but not too many such that the tuning would take too long. After trial and failure, I found that this set of values proved to tune the classifiers decently well. Even though a larger amount of n_estimators may have resulted in better performance, it would come at an increased computational cost, of which I did not deem necessary for my use case. For the max_depth, I chose to tune both classifiers to the same depth ranges. With the range of values I chose, I tried to balance between underfitting and overfitting the models.

The hyperparameter random_seed was defaulted to 0 to esure that the results from the tuning were reproducible. The random_seed was used consistently with the random_state argument throughout the running of experiments. That is, both for the splitting of data, kfold, and model instantiation. This made every component with randomization be controlled by the seed, and ensure consistent results for each execution. In addition to this, I created a random number generator using np.random.default_rng(random_state). I put an argument in the init of both my decision tree and random forest classifiers called self.rng, of which I used self.rng.choice() to generate reproducible and random numbers.

```
# Create seed for reproducibility
seed = 0
np.random.seed(seed)
```

Listing 11: Setting random seed to 0

For the fairness, I used a combination of different techniques to ensure that the model selection was unbiased. For example, I split the dataset into both training, validation and test sets to prevent data leakage. Further techniques I used to create reliable and generalizable model performance will be discussed now.

3.1.2 Model Selection Method

I chose to use k-fold cross-validation, with 5-folds to evaluate each combination of hyperparameters. By using k-fold cross-validation, I could split the dataset 5 times into 5 equal parts, with one part being validation set, and rest being training sets. This ensured that each part of the dataset was used for validation once.

I leaned more towards k-fold cross-validaiton than hold-out validation, because I felt I really understood how k-fold worked, and liked the explanation from the lectures. In addition to this, k-fold provides a good estimate of model performance.

3.1.3 Performance Measure

For the performance measure, I chose to go with accuracy. This is because accuracy is a good performance metric with binary classification tasks like the datasets in this assignment. If the dataset were more imbalanced, I would've chosen to go with F1-Score instead. But I concluded that the imbalance was not large enough in the data analysis part, that it would be necessary use F1-score.

3.2 Model Selection Results (Wine Dataset)

After performing the tuning on the Wine Dataset, I got the following output from my function:

Model	Details
Decision Tree (self)	
Mean Validation Accuracy	0.7829
Parameters	{ `max_depth`: 10, `criterion`: `'gini'`, `random_state`: 0 }
Random Forest (self)	
Mean Validation Accuracy	0.8628
Parameters	{ `n_estimators`: 30, `max_depth`: 8, `max_features`: `'sqrt'`, `criterion`: `'gini'`, `random_state`: 0 }
Decision Tree (sklearn)	
Mean Validation Accuracy	0.8543
Parameters	{ `max_depth`: 5, `criterion`: `'gini'`, `random_state`: 0 }
Random Forest (sklearn)	
Mean Validation Accuracy	0.8743
Parameters	{ `n_estimators`: 30, `max_depth`: 10, `max_features`: `'sqrt'`, `criterion`: `'gini'`, `random_state`: 0 }

Table 1: Performance of mean validation accuracy and best hyperparameters for Decision Tree and Random Forest models (both self and sklearn).

In order to evaluate each of the tuned models, I created a function to do this.

```
# Evaluate model function => Train and test the best models
def evaluate_model(model_class, params):
    model = model_class(**params)
    # Train the model
    model.fit(X_train_and_val, y_train_and_val)
    y_pred = model.predict(X_test)
    return accuracy_score(y_test, y_pred)
```

Listing 12: Evaluate model function

Here I trained the model on this given best hyperparameters, predicted on the test set, and used accuracy score to get the accuracy of the model.

3.2.1 Decision Trees

Here is the model selection results of my decision tree implementation:

Model	Details
Decision Tree (self)	
Test Accuracy	0.7667
Parameters	{ `max_depth`: 10, `criterion`: `'gini'`, `random_state`: 0 }

Table 2: Test accuracy and best hyperparameters for Decision Tree (self).

Here we can see that the model had its best accuracy with max_depth set to 10 and criterion to gini.

3.2.2 Random Forest

Here is the model selection results of my random forest implementation:

Model	**Details**
Random Forest (self)	
Test Accuracy	0.8400
Parameters	{ `n_estimators`: 30, `max_depth`: 8, `max_features`: `'sqrt'`, `criterion`: `'gini'`, `random_state`: 0 }

Table 3: Test accuracy and best hyperparameters for Random Forest (self).

The model performed with the best accuracy with 30 estimators, 8 in max_depth, sqrt as max_features and gini as the criterion.

3.2.3 Comparing Decision Trees and Random Forests

Table of model performance:

Model	**Decision Tree (self)**	**Random Forest (self)**
Test Accuracy	0.7667	0.8400
Parameters	{ `max_depth`: 10, `criterion`: `'gini'`, `random_state`: 0 }	{ `n_estimators`: 30, `max_depth`: 8, `max_features`: `'sqrt'`, `criterion`: `'gini'`, `random_state`: 0 }

Table 4: Comparison of Decision Tree and Random Forest (self).

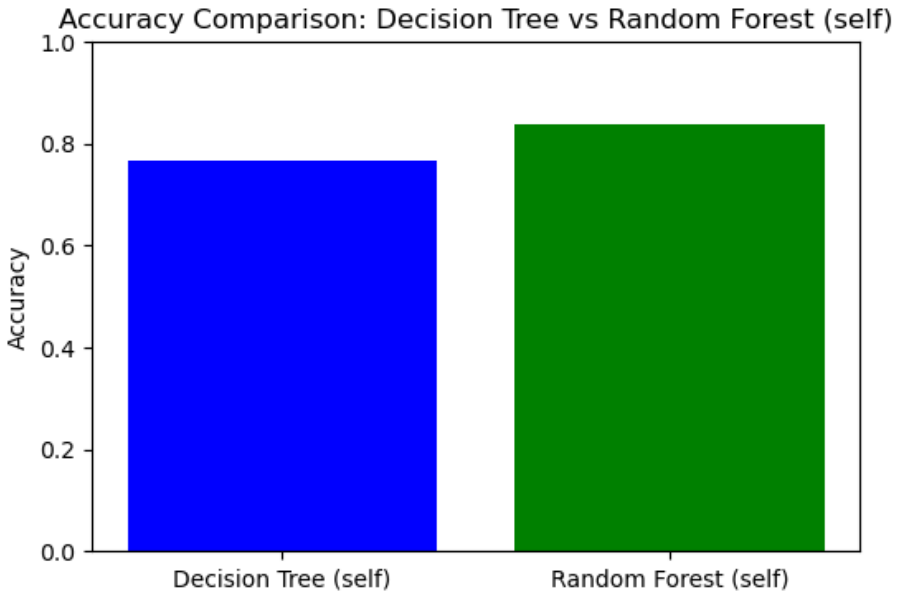

Figure 5: Comparison of Decision Tree and Random Forest (self).

Both from the table of model performance, and from the plot, we can see that the random forest classifier outperformed the decision tree classifier.

3.2.4 Comparison to Existing Implementations

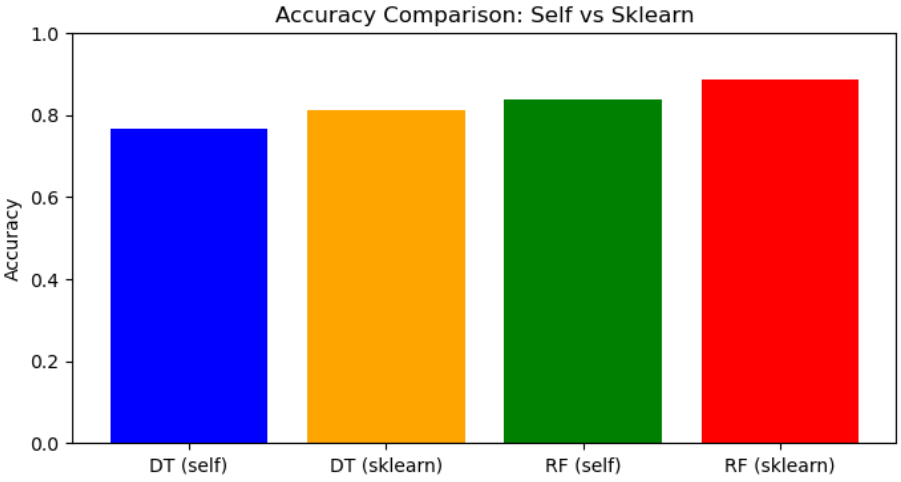

Figure 6: Comparison of Decision Tree and Random Forest (all).

From this plot of performance, we can see that sklearn outperformed my implementations both for decision tree and random forest respectively. There seems to be around 4-5 percent performance difference (accuracy) in both of my classifiers compared to sklearn.

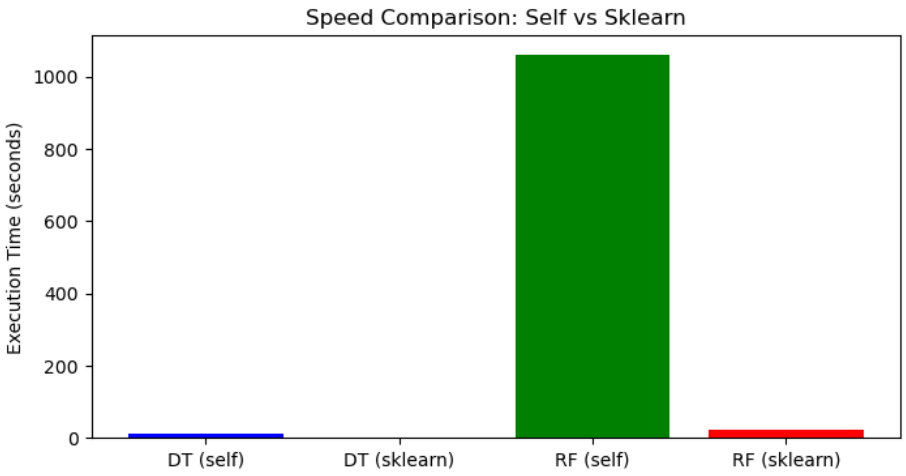

Figure 7: Comparison of Decision Tree and Random Forest Speed (all).

Sklearn's existing implementation of the classifiers also outperformed my implementations in terms of speed. As one can tell from this plot of model speed, there is a large gap between the implementations.

3.3 Results on the Coffee Dataset

I repeated all the steps on the Coffee Dataset as for the Wine Dataset, and got this output after tuning and evaluating:

Model	Details
Decision Tree (self)	
Test Accuracy	0.8571
Parameters	{ `max_depth`: 3, `criterion`: `'gini'`, `random_state`: 0 }
Random Forest (self)	
Test Accuracy	0.8016
Parameters	{ `n_estimators`: 5, `max_depth`: 15, `max_features`: `'None'`, `criterion`: `'gini'`, `random_state`: 0 }
Decision Tree (sklearn)	
Test Accuracy	0.8333
Parameters	{ `max_depth`: 3, `criterion`: `'gini'`, `random_state`: 0 }
Random Forest (sklearn)	
Test Accuracy	0.8492
Parameters	{ `n_estimators`: 10, `max_depth`: 3, `max_features`: `'None'`, `criterion`: `'gini'`, `random_state`: 0 }

Table 5: Performance of test accuracy and best hyperparameters for Decision Tree and Random Forest models (both self and sklearn).

An interesting observation is that the numbers of estimators used to produce good test results was substantially reduced. We can see that the self random forest uses 5 estimators, and the sklearn random forest uses 10, which is lower compared to the Wine Dataset. Another observation is that both of the decision trees uses less max_depth to get a good accuracy.

Lastly, a peculiar observation is that my implementation of the decision tree beats my implementation of the random forest. This may be explained by the fact that the random forest only found a optimal set of hyperparameters where the number of estimators was 5. If given more time, I could've tuned with many more possibilities, and my random forest may could have proven to beat the decision tree after all.

4 Conclusion

I learned a lot about machine learning, and how each method works under the hood. I have gotten to appreciate the already available and implemented functionalities of learning algorithms.

If I were to do anything differently, I would focus on deterministic results earlier. This prove to be a significant issue for me later down the line in terms of debugging, and getting weird results on my classifier tuning. I would also try out more hyperparameter combinations if I had more time to play around with.