Inf264 Oblig 1 Report

Introduction

(Briefly describe what you did in this project.)

In this project we implemented our own decision tree classifier and random forest classifier. We tested our models on some wine data and on coffee data. We also compared our classifiers to sklearn's classifiers.

Division of labor

Me and Magnus Brørby worked together, we both did the project each on our own computers. We helped each other and chose what parts of the implementation, testing and tuning that we thought was best. After implementing we worked together on the report.

Data Analysis

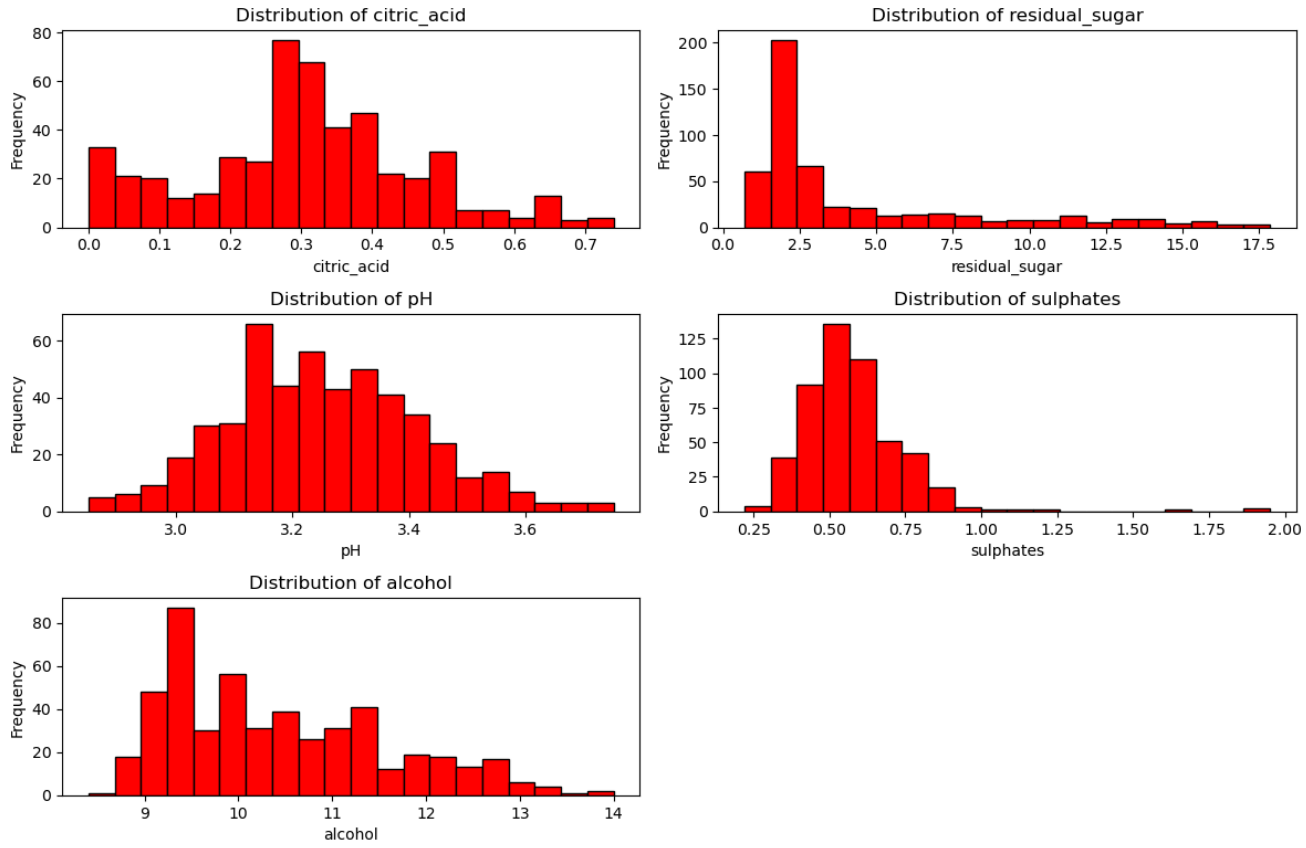
(Analysis of the data. What features are available? What do they look like? What are the distributions of the features and labels? Figure(s) are encouraged.)

Analysis of wine data.
In the wine data we have 5 features which is citric acid, residual sugar, pH, sulphates and alcohol.

We see that there is a pretty normal distribution of the data, we also see that sulphates may have some outliers.

And we also see that the distribution of the target values are very balanced which means we can use accuracy to validate the model.

Features:



Target:

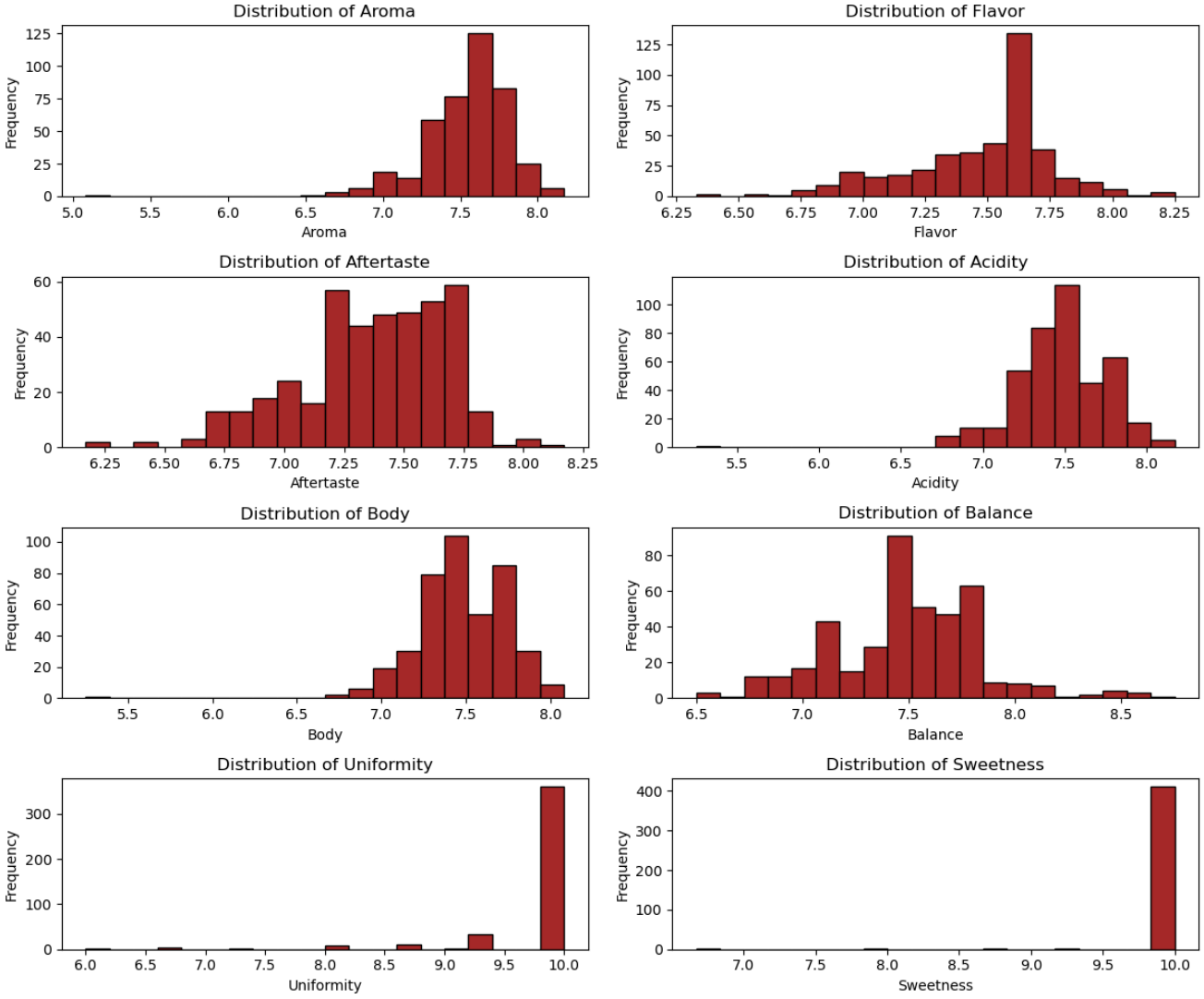
Analysis of coffee data.

In the coffee data we have 8 features which are aroma, flavor, aftertaste, acidity, body, balance, uniformity and sweetness.

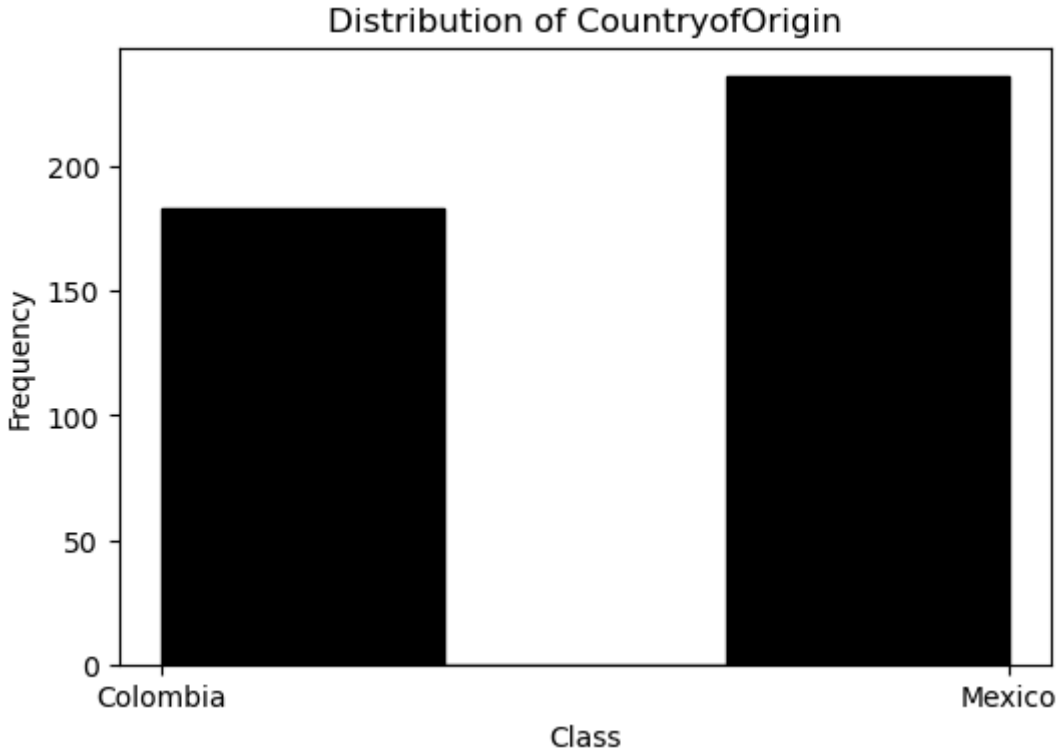
We see that uniformity and sweetness may have some outliers.

We also see that there is a little more imbalance in the targets here than the wine data but not enough that we can not use accuracy.

Features:



Target:



Data Preprocessing

(Did you do any data preprocessing? In either case, explain why you did or did not do it.)

We did no data preprocessing. We did not normalize or standardize the data since decision tree and random forest are tree-based models which are scale-invariant. That means that even though we can have a feature value that ranges from 0 to 1000 and one that ranges from 0 to 10, the first feature value won't have more to say then the feature value that ranges from 0 to 10.

We did not correct any imbalance since we saw that there was not enough imbalance in either of the datasets.

Implementations

Decision Trees

(Description of your decision tree implementation.)

In decision_tree.py you can see our implementation of decision tree.

The class DecisionTree has some Initialization values you need to set when creating an instance of DecisionTree. Self is for setting and getting important values in the decision tree. Max_depth is how deep the tree can go. Max_depth that is lower or equal to 1 will raise an

Exception, for us it made sense that the depth had to be at least 2. When the depth is set to 2 it means we have a root node and two leaf nodes.

```python
class DecisionTree:
    def __init__(
        self,
        max_depth: int | None = None,
        criterion: str = "entropy",
        max_features: str | None = None,
        seed: int = 0
    ) -> None:
        self.root = None
        self.criterion = criterion
        self.max_depth = max_depth
        self.max_features = max_features
        self.seed = seed
        self.rng = np.random.default_rng(seed)
        if (max_depth is not None and max_depth <= 1):
            raise Exception("The tree can not be less then 2 in depth")
```

Here is an example of when the max_depth is 2:

```
Node: Feature 8 <= -0.03568989086623185
  Leaf: Predict 0
  Leaf: Predict 1
```

Criterion is how the information gain will be computed, this value can either be "entropy" or "gini" for gini index. Max_features is how many features we should consider when splitting, this value can either be None, "sqrt" for square root or "log2". We added the seed so that we could reproduce the same result when running multiple times. The self.rng is so that when we want to do something random multiple times it won't do the same thing multiple times.

Here is an example of using the decision tree:

```python
rf = DecisionTree(max_depth=2, criterion="entropy", max_features="sqrt")
rf.fit(X_train, y_train)

print_tree(rf.root)


print(f"Training accuracy: {accuracy_score(y_train, rf.predict(X_train))}")
print(f"Validation accuracy: {accuracy_score(y_val, rf.predict(X_val))}")
```

Fit function

We will go through the fit, print_tree and the predict functions. The fit function is responsible for creating the decision tree and inputs X_train and y_train.

```python
# If the depth is at 0 we will assign the root and continue the creation of the tree
if depth == 0:
    self.root = self.fit(X, y, depth = 1)
    return self.root


# If all the labels are the same, return a pure  leaf node
if len(np.unique(y)) == 1:
    return Node(value= y[0]) #return a pure leaf node


# If max depth is not none and the current depth is equal or more than max depth return a leaf node with the most common label
if self.max_depth is not None and depth >= self.max_depth:
    return Node(value = most_common(y))


# Return the best feature and the threshold where we split using the find_best_split_mean function
best_feature, best_threshold = find_best_split_mean(X,y, self)

# If it can't find a split return a leaf node with the most common label
if best_feature == None:
    return Node(value = most_common(y))


# Here we split the X values on the feature we found this will return lists with true and false
left_mask = split(X[:, best_feature], best_threshold)
right_mask = ~left_mask  # Opposite list with true and false then above


# Create the two subtrees
left_subtree = self.fit(X[left_mask], y[left_mask], depth +1)
right_subtree = self.fit(X[right_mask], y[right_mask], depth +1)


# Keep splitting the values and return the two subtrees as left and right in the node
return Node(feature=best_feature, threshold=best_threshold, left=left_subtree, right=right_subtree)
```

Inside the fit function we check if the depth is zero, and if it is we return a root node that calls upon fit further. This is so that we set the root node correctly. When fit is first called the depth will always be zero.

If all the values in y is the same we return a leaf node with that value for y.

If the current depth is higher than max depth we return a node with the most common value for y.

The find_best_split_mean function finds the best split and returns the feature to split on and what value the threshold for that feature is.

If the function couldn't find a split it will return a node with the most common value in y. This handles the case where all data points have identical feature values and returns a leaf node with the most common label.

But if it found a split it will create two lists containing true and false to create the split in the X and y values.

Further the code will create two subtrees with this split that calls upon fit further. And then return a node that has these subtrees as left and right, this node also has the best_feature as feature and best_threshold as threshold.

Finding the best split function

"Finding the best split" function inside fit:

```
def find_best_split_mean(X: np.array,y: np.array, self):

    seed = self.seed

    np.random.seed(seed) # Sets the seed as set in the init of decision tree

    best_gain = -1
    best_feature = None
    best_threshold = None
    selected_features = None

    # Consider all features
    if self.max_features is None:
        selected_features = np.arange(X.shape[1])

    # Consider the square root of features
    elif self.max_features == "sqrt":
        n = int(np.sqrt(X.shape[1]))

        selected_features = np.random.choice(np.arange(X.shape[1]), size=n, replace=False)

    # Conisider the log2 of features
    elif self.max_features == "log2":
        n = int(np.log2(X.shape[1]))

        selected_features = np.random.choice(np.arange(X.shape[1]), size=n, replace=False)
    else:
        raise ValueError("Invalid value for max_features")
```

```
def find_best_split_mean(X: np.array,y: np.array, self):

    best_gain = -1
    best_feature = None
    best_threshold = None
    selected_features = None

    # Consider all features
    if self.max_features is None:
        selected_features = np.arange(X.shape[1])

    # Consider the square root of features
    elif self.max_features == "sqrt":
        n = int(np.sqrt(X.shape[1]))

        selected_features = self.rng.choice(np.arange(X.shape[1]), size=n, replace=False)

    # Conisider the log2 of features
    elif self.max_features == "log2":
        n = int(np.log2(X.shape[1]))

        selected_features = self.rng.choice(np.arange(X.shape[1]), size=n, replace=False)
    else:
        raise ValueError("Invalid value for max_features")
```

First we will calculate which features we will consider when trying to find the best split.

If the max_features is set to None we will consider all features.

If the max_features is set to "sqrt" we will consider the square root of all the features. So if we have 100 features only 10 random will be considered.

If the max_features is set to "log2" we will consider the log2 of all the features.

Here we use self.rng(choice) to choose a certain amount of random features, when this runs multiple times it won't choose the same features.

The rest of the Finding the best split function:

```
# Go through the features and find the best split
for feature in selected_features:
    threshold = np.mean(X[:, feature]) # Compute the threshold


    left_mask = split(X[:, feature], threshold) # Mask with the left part of the split
    right_mask = ~left_mask # Mask with the right part of the split

    if len(y[left_mask]) == 0 or len(y[right_mask]) == 0: # If the split is invalid continue to try to find a split
        continue


    gain = information_gain(y, y[left_mask], y[right_mask], self.criterion) # compute the information gain of the split

    if gain > best_gain: # if the gain is better than default or the previous then choose that split
        best_gain = gain
        best_feature = feature
        best_threshold = threshold

return best_feature, best_threshold
```

Further in the function we iterate through the chosen features, selected_features will be a list of indexes.

We compute the threshold on the current chosen feature by taking the mean of these feature values, then we use this threshold to make a split mask. We check if the split is valid by checking that the length is not 0.

Further we compute the information gain for that current feature, if that gain is better then the last gain we choose this split. We do this for all selected features and return the best feature and threshold to split on.

Print_tree

print_tree function is just for visualizing the tree:

```
def print_tree(node, level=0):
    if node.is_leaf():
        print("  " * level + f"Leaf: Predict {node.value}")
    else:
        print("  " * level + f"Node: Feature {node.feature} <= {node.threshold}")
        if node.left:
            print_tree(node.left, level + 1)
        if node.right:
            print_tree(node.right, level + 1)
```

Example of visualizing the tree:

```
Node: Feature 8 <= -0.035689890866231855
  Leaf: Predict 0
  Leaf: Predict 1
```

Predict function

The predict function is where we input a list of observations and we predict the target of each observation in X.

So for each observation we get the feature value in the observation that the node splits on. And decide if we are going to the left or to the right node depending on the threshold and the feature value. We traverse the tree until we get a target value.

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

    threshold = self.root.threshold

    feature = self.root.feature

    for el in X: # for each feature
        xfeatureval = el[feature] # get the feature value


        # decide if we are going to the left or right depending on the threshold and the feature value
        if xfeatureval <= threshold:
            list_of_predicts.append(predict_recursive(self.root.left, el))

        else:
            list_of_predicts.append(predict_recursive(self.root.right, el))


    return np.array(list_of_predicts)
```

The predic_recursive function just repeats this process until we hit a leaf. When it hits a leaf we return the value the leaf has, this becomes our prediction for target for that observation.

```python
def predict_recursive(currentNode: Node, X: np.ndarray) -> int:

    if currentNode.is_leaf():
        return currentNode.value # If leaf return the value of leaf

    feature = currentNode.feature

    threshold = currentNode.threshold

    # decide to go to left or right node
    if X[feature] <= threshold:
        return predict_recursive(currentNode.left, X)
    else:
        return predict_recursive(currentNode.right, X)
```

We won't go through every helper function, but these can be found in the decision_tree.py file.

Random Forest

(Description of your random forest implementation.)

Constructing the random forest.
In random_forest.py you can see our implementation of random forest.

Here we also need to initialize the random forest with a couple of variables.

n_estimators is how many trees the forest should contain.

max_depth, criterion, max_features, seed and the self.rng work in the same way as in decision tree.

We will store our trees in a list called self.trees.

```python
class RandomForest:
    def __init__(
        self,
        n_estimators: int = 100,
        max_depth: None | int = 5,
        criterion: str = "entropy",
        max_features: None | str = "sqrt",
        seed: int = 0
    ) -> None:
        self.n_estimators = n_estimators
        self.max_depth = max_depth
        self.criterion = criterion
        self.max_features = max_features
        self.trees = []
        self.seed = seed
        self.rng = np.random.default_rng(seed)
```

Fit function

In the random forest fit function we will create n_estimators amount of trees. We will choose random observations where observations can occur multiple times. Then we create the tree and fit it on the sampled X and y. Further we add this tree to the tree list and do this for every tree.

```
def fit(self, X: np.ndarray, y: np.ndarray):

    num_trees = self.n_estimators   # num of trees to be created

    n = len(X) # how many features we have

    for _ in range(num_trees): # for each tree

        np.random.seed(self.seed)

        X_indexes = self.rng.choice(np.arange(n), size=n, replace=True) #choose observations at random,
        # where observations can occur multiple times

        X_sampled = X[X_indexes]

        y_sampled = y[X_indexes]

        rf = DecisionTree(max_depth=self.max_depth, criterion= self.criterion, max_features=self.max_features) # create the tree
        rf.fit(X_sampled, y_sampled) # fit the tree
        self.trees.append(rf) # add the tree to the list
```

Predict function

In the predict function we predict what target each observation should have.

We do this by letting each tree predict what target all the observations should be. Then we take the majority of the predicted target for each observation and return that.

If 5 trees predicted that the first observation has target 1 and 3 trees predicted that it has target 0, the random forest would predict that the first observation has target 1.

```
def predict(self, X: np.ndarray) -> np.ndarray:

    list_of_predicts = []

    for index in range(len(self.trees)): #for every tree
        predict = self.trees[index].predict(X) #get the current tree and predict all the X values for that tree
        list_of_predicts.append(predict) # add to list

    list_of_predicts = np.array(list_of_predicts) #convert into np.array

    # compare every prediction
    prediction = []

    for index in range(len(X)): # for each feature in X
        labels = list_of_predicts[:, index] # this becomes a list of all the trees prediction of the current feature

        most_common_element = np.bincount(labels).argmax() # get the most common target for the current feature
        prediction.append(most_common_element) # add this to the list


    return prediction
```

Model Selection and Evaluation

Model Selection Process

(Detailed description of model selection process. Which hyperparameters did you tune and which values did you consider? Which model selection technique did you use? Which scoring metric did you use? Explain your choices!)

We decided to use k-fold cross validation when tuning.

We used k-fold cross validation since with k-fold cross validation we will reduce overfitting risk since it chooses the best average over multiple splits. It is less biased then a single split and it uses the dataset efficiently.

```
#Split the data into test and (training and validation)

X_train_val, X_test, y_train_val, y_test = train_test_split(X,y, test_size=0.3, random_state=seed, shuffle=True)

# Use k fold cross validation
kf = KFold(n_splits=5, shuffle=True, random_state=seed)
```

Here is the values we considered for tuning:

```
# Lists of hyperparameters that i want to tune

max_depth_params = [3,10,15,20,None]

n_estimators = [2,10,15,30]

criterion = ["gini", "entropy"]

max_features = ["sqrt", "log2", None]
```

We chose to include all the hyperparameters we could choose, since this will more likely give us a better result than only including for example criterion.

We wanted to include more values for each hyperparameter since including more will potentially improve our hyperparameters but we figured out that it takes a lot of time to run, so we chose to only have these.

```python
# Function for tuning decision tree using k-fold cross validation

def tune_decision_tree(model_type, X_train_val, y_train_val, max_depth_params, criterion, max_features, kf):
    best_accuracy = [0]  # To store the best average accuracy and the corresponding hyperparameters

    # Iterate through all combinations of hyperparameters
    for maxdp in max_depth_params:
            for crit in criterion:
                for mf in max_features:

                    # Init the model
                    if model_type == "DecisionTree":
                        rf = DecisionTree(max_depth=maxdp, criterion=crit, max_features=mf, seed=seed)
                    elif model_type == "SklearnDecisionTree":
                        rf = DecisionTreeClassifier(max_depth=maxdp, criterion=crit, max_features=mf , random_state=seed)
                    else:
                        raise ValueError("Invalid model_type. Choose either 'RandomForest' or 'SklearnRandomForest'.")

                    # List to store accuracies for each fold
                    fold_accuracies = []

                    # k-fold cross-validation
                    for train_index, val_index in kf.split(X_train_val):
                        # Split the data into training and validation sets for each fold
                        X_training, X_val = X_train_val[train_index], X_train_val[val_index]
                        y_training, y_val = y_train_val[train_index], y_train_val[val_index]

                        # fit on training data
                        rf.fit(X_training, y_training)

                        # Predict validation
                        y_pred = rf.predict(X_val)

                        # Calculate accuracy for the current fold
                        accuracy = accuracy_score(y_val, y_pred)
                        fold_accuracies.append(accuracy)

                    # Calculate average accuracy across all folds
                    average_accuracy = np.mean(fold_accuracies)

                    # If current average is better then update the best average accuracy
                    if average_accuracy > best_accuracy[0]:
                        best_accuracy = [average_accuracy, maxdp, crit, mf]

    # Return the best hyperparameters and accuracy
    return best_accuracy
```

Above is our function for tuning a decision tree model either our own implementation or the one from sklearn given the values mentioned earlier. We have a similar function to this for random forest:

```python
# Function for tuning random forest using k-fold cross validation

def tune_random_forest(model_type, X_train_val, y_train_val, n_estimators, max_depth_params, criterion, max_features, kf):
    best_accuracy = [0]  # To store the best average accuracy and the hyperparameters

    # Iterate through all combinations of hyperparameters
    for maxdp in max_depth_params:
        for n_est in n_estimators:
            for crit in criterion:
                for mf in max_features:

                    # Init the model
                    if model_type == "RandomForest":
                        rf = RandomForest(n_estimators=n_est, max_depth=maxdp, criterion=crit, max_features=mf, seed=seed)
                    elif model_type == "SklearnRandomForest":
                        rf = RandomForestClassifier(n_estimators=n_est, max_depth=maxdp, criterion=crit, max_features=mf , random_state=seed)
                    else:
                        raise ValueError("Invalid model_type. Choose either 'RandomForest' or 'SklearnRandomForest'.")

                    # List to store accuracies for each fold in the k-fold cross validation
                    fold_accuracies = []

                    # k-fold cross-validation
                    for train_index, val_index in kf.split(X_train_val):
                        # Split the data into training and validation sets for each fold
                        X_training, X_val = X_train_val[train_index], X_train_val[val_index]
                        y_training, y_val = y_train_val[train_index], y_train_val[val_index]

                        # Fit random forest on the training data
                        rf.fit(X_training, y_training)

                        # Predict validation
                        y_pred = rf.predict(X_val)

                        # Calculate accuracy for the current fold
                        accuracy = accuracy_score(y_val, y_pred)
                        fold_accuracies.append(accuracy)

                    # Calculate average accuracy across all folds
                    average_accuracy = np.mean(fold_accuracies)

                    # If current average is better then update the best average accuracy
                    if average_accuracy > best_accuracy[0]:
                        best_accuracy = [average_accuracy, n_est, maxdp, crit, mf]

    # Return the best hyperparameters and accuracy
    return best_accuracy
```

Using the tuning function:

```python
# Find best values for hyperparameters for our own decision tree classifier
best_hyperparameters_for_decisionTree = tune_decision_tree("DecisionTree",
X_train_val, y_train_val, max_depth_params, criterion, max_features, kf)


print(best_hyperparameters_for_decisionTree)


print(f"The best average accuracy is
{best_hyperparameters_for_decisionTree[0]} with Max_depth:
{best_hyperparameters_for_decisionTree[1]}, Criterion:
{best_hyperparameters_for_decisionTree[2]}, Max_features:
{best_hyperparameters_for_decisionTree[3]}.\n")
```

Output:

```
[np.float64(0.7885714285714285), 10, 'gini', None]
The best average accuracy is 0.7885714285714285 with Max_depth: 10, Criterion: gini, Max_features: None.
```

In the code above we show how we store the hyperparameters that we find after tuning.

Then we test this model with the hyperparameters on the test data:

```
# Testing our DecisionTree with the the hyperparameters
rf = DecisionTree(max_depth=best_hyperparameters_for_decisionTree[1],
criterion=best_hyperparameters_for_decisionTree[2],
max_features=best_hyperparameters_for_decisionTree[3], seed=seed)


rf.fit(X_train_val, y_train_val)


print(f"Training and validation accuracy: {accuracy_score(y_train_val,
rf.predict(X_train_val))}")

print(f"Test accuracy: {accuracy_score(y_test, rf.predict(X_test))}")
```

Output:

```
Training and validation accuracy: 0.9942857142857143
Test accuracy: 0.76
```

Here we see that our model got an accuracy of 76% on the test data. We use accuracy since this is fine when there is not enough imbalance on the target distribution.

We did a very similiar process further when tuning the random forest and sklearn's equivalents, this can be found in the run_experiments.ipynb file.

Model Selection Results (Wine Dataset)

(If you want to test if you get the same results as us, you need to remember to set seed to 0)

Decision Trees

(Description of model selection results for your decision tree implementation. Which hyperparameters were picked for the final model?)

The final hyperparameters were with Max_depth: 10, Criterion: gini, Max_features: None.

This got the best average accuracy of 0.7885714285714285.

Random Forest

(Description of model selection results for your random forest implementation. Which hyperparameters were picked for the final model?)

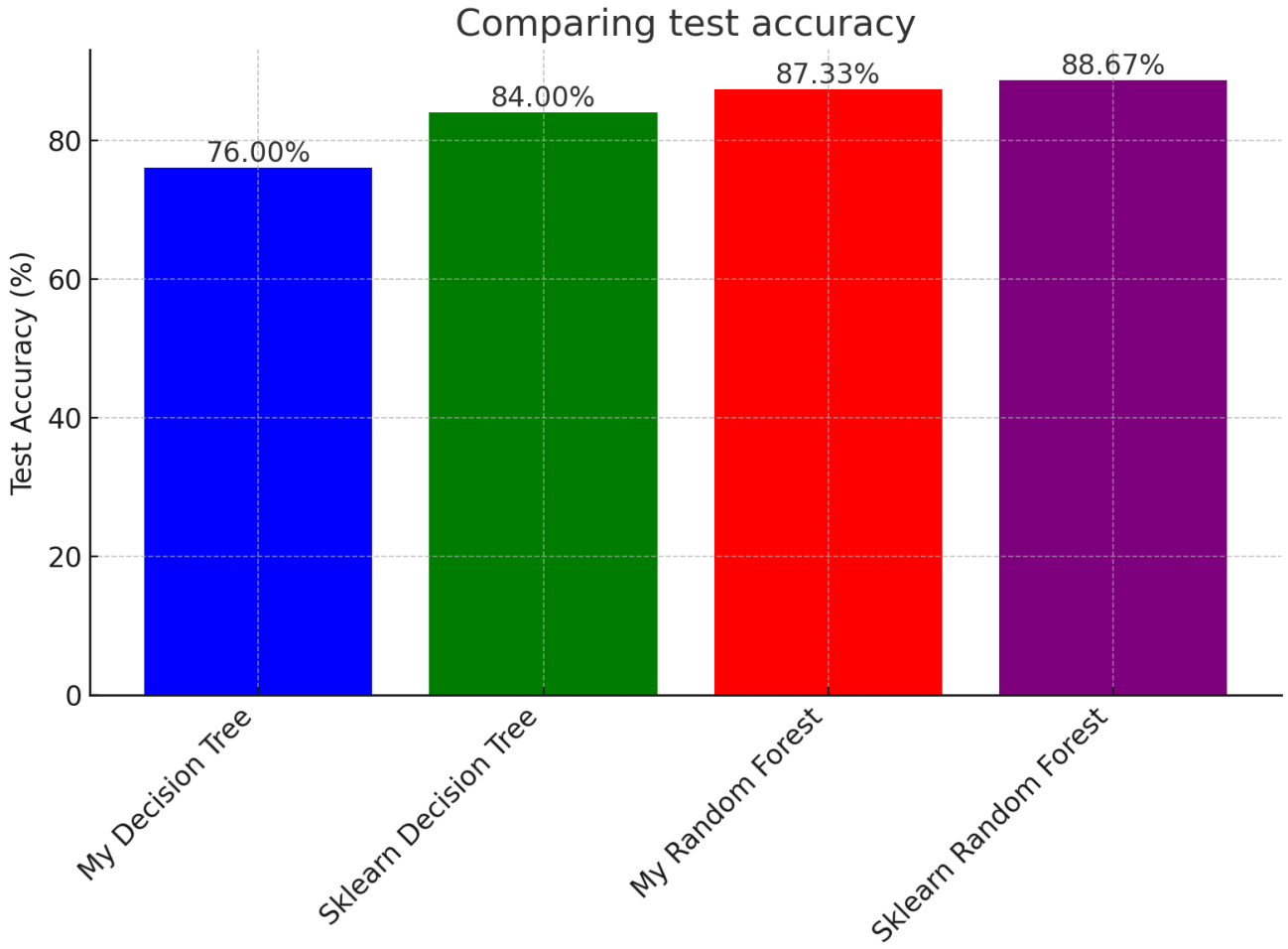
The final hyperparameters were with N_estimators: 10, Max_depth: 10, Criterion: gini, Max_features: sqrt.

This got the best average accuracy of 0.9457142857142857

Comparing our Decision Tree and Random Forests with sklearn's

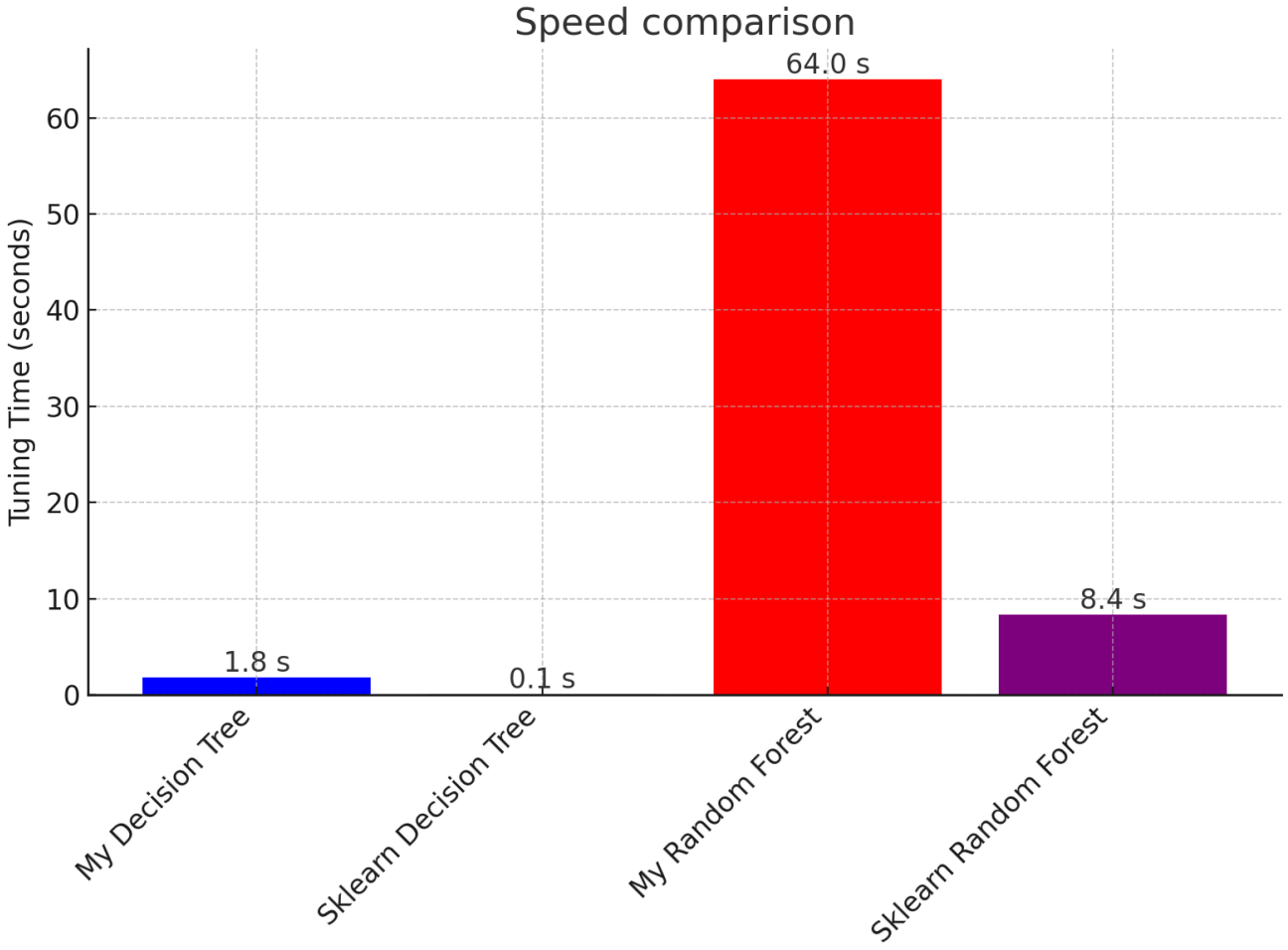
(Comparison of the two models. Which one performed better? Figure(s) are encouraged.) Comparison to Existing Implementations (Comparison of my implementation to sklearn's implementation. How do they compare in terms of performance and speed?)

Here is a comparison of the different classifiers on the test accuracy:



We can see that sklearn's versions does a little better than our classifiers. And that random forest does better than decision tree in both ours implementation and in sklearn's.

Comparison of the different model's speed when tuning:



We see that sklearn's decision tree is 18 times faster then my own. And for sklearn's random forest it is about 7.5 times faster. This has a lot to say when working with bigger datasets and tuning with more values.

Results on the Coffee Dataset

(How did your models perform on the coffee dataset? What were the results? Were the hyperparameters the same as for the wine dataset? Any interesting observations?)

Decision tree

The final hyperparameters for our own decision tree on the coffee dataset were with Max_depth: 10, Criterion: entropy, Max_features: log2.

Where the the best average accuracy was 0.7406195207481006

The test accuracy was: 0.7698412698412699

Random forest

The final hyperparameters for our own random forest on the coffee dataset were with N_estimators: 30, Max_depth: 15, Criterion: gini, Max_features: None.

Where the best average accuracy was 0.9423728813559322

The test accuracy was: 0.8174603174603174

Comparing the hyperparameters in coffee to the wine data:

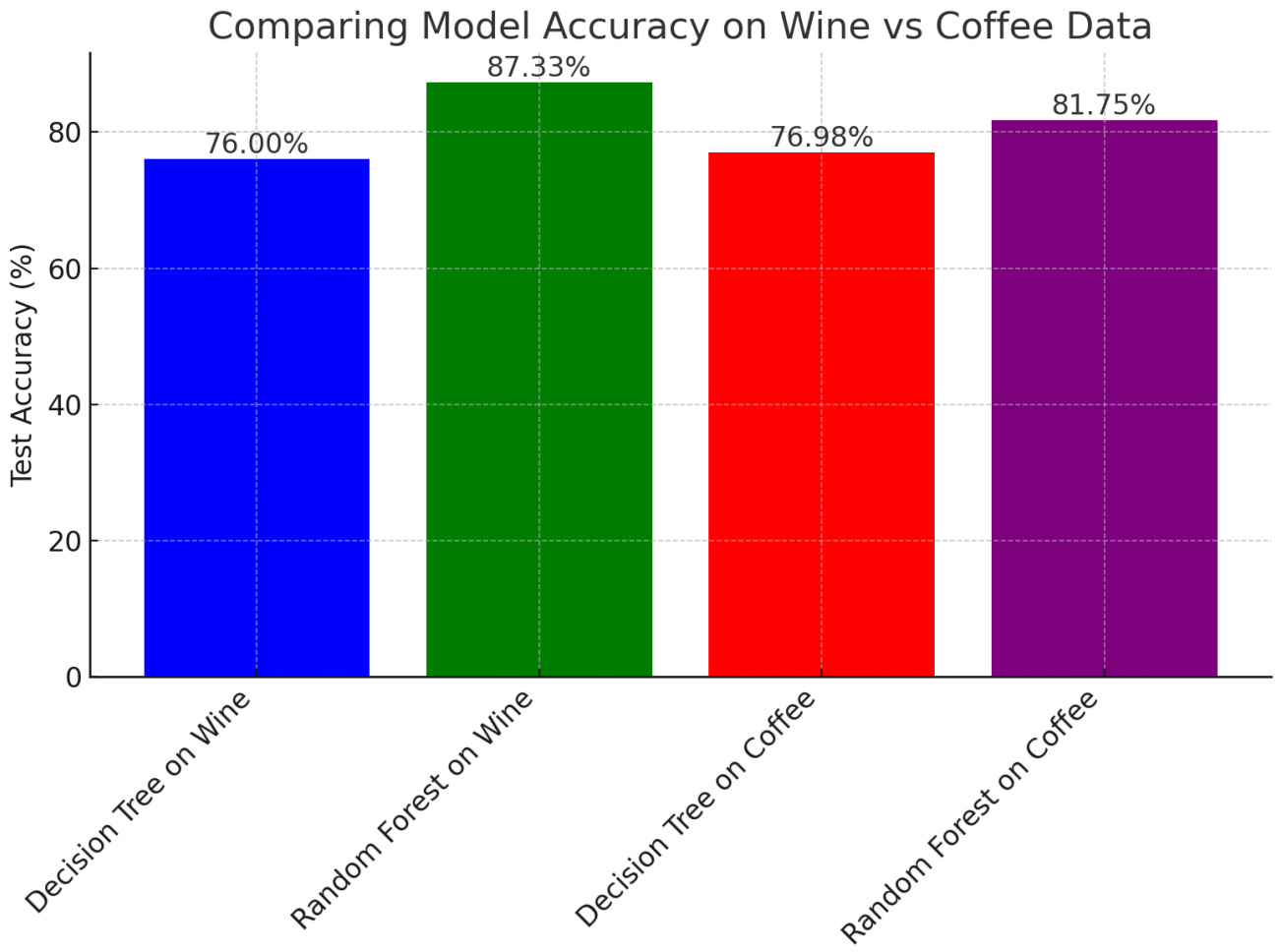
	Model	Max Depth	Criterion	Max Features	N Estimators
1	Wine Decision Tree	10	gini	None	-
2	Wine Random Forest	10	gini	sqrt	10
3	Coffee Decision Tree	10	entropy	log2	-
4	Coffee Random Forest	15	gini	None	30

We did not get exactly the same hyperparameters in the different datasets.

All of the models chose 10 as max depth except for Random forest for coffee data who chose 15. Also all the models chose gini as criterion except for Decision tree for coffee data who chose entropy. The Random forest for the coffee data chose none as max features and 30 trees. While Random forest for wine data chose sqrt as max features and 10 trees. Decision tree for wine data chose none as max features which is not surprising, but decision tree chose log2 as max feature which I did not expect.

Decision tree on coffee data:
The test accuracy was: 0.7698412698412699
Random forest on coffee data:
The test accuracy was: 0.8174603174603174

Comparing the models with different data:



We see that the random forest performs better then decision trees on both datasets. We also see that the decision trees are very similar and that the random forest in coffee performs worse then random forest in wine.

Maybe the results are like they are because of imbalance in the target distribution or maybe the dataset has less correlation between the features and target.

Conclusion

(Conclusion of the project. What did you learn? What would you do differently next time? Did you encounter any problems?)

We learned how decision tree and random forest works. How the different hyperparameters affect their accuracy. We learned how the k-fold cross validation works. Learned how to correctly measure accuracy.

We spent too much time trying to find the right amount of values for tuning. We should have just kept it like it is now. With these values we get the essence of the project without having to tune for an hour or more.

We had a problem not being able to reproduce the outputs but we fixed it after implementing seed and using rng in our decision tree and random forest.