Supervised and Experiential Learning
Practical Work 2
Combining Multiple Classifiers

Gerard Navarro Pérez

May 9, 2024

Contents

1 Introduction

This report delves into the outcomes of practical work 2, focusing on the examination of two ensemble of classifiers: Random Forest, pioneered by Leo Breiman in 2001, and Decision Forest, introduced by Tin Kam Ho in 1998. Initially, we will explain the concepts behind these classifiers, detailing their implementations and execution mechanisms. Following this, we will dig into the results derived from applying both classifiers across diverse datasets. Finally, we will draw conclusions based on the observed outcomes.

2 Ensemble of classifiers

Classifier ensemble methods aim to create a collection of discriminant models, or classifiers, with the objective of reducing the discrimination error inherent in each individual classifier. These models can be of the same type or diverse, derived from different samples or through varied attribute selection techniques, all within a supervised database context where the class attribute is qualitative. The essential criteria for application involve having a sufficient number of representative examples across different class labels. Among the most prevalent methods are Bagging, Boosting (including AdaBoost and Gradient Boosting), as well as Random Decision Forests, Decision Forests, and Random Forests. The output of these methods is a set of classifiers capable of accurately classifying the qualitative attribute of interest. During the discrimination or classification process, when a new instance requires classification, all the classifiers are employed to predict its class label. The final class assigned to the instance is determined through a voting mechanism, which may involve weighting the predictions from each classifier.

2.1 Random Forest vs. Decision Forest

Decision Forests, as proposed by Tin Kam Ho in 1998, and Random Forests, introduced by Leo Breiman in 2001, are both ensemble methods aimed at reducing error correlation between classifiers. The main idea behind both methods is to construct a larger number of un-pruned decision trees. However, they differ in their approach to feature selection and dataset sampling. In Ho's Decision Forests, each tree is grown using a random subspace of features that remains consistent across all node splits. Conversely, Breiman's Random Forests employ a random subspace of features for splitting at each node, and the training set for each tree is sampled through bootstrapping from the original dataset. This variance in feature selection and sampling techniques contributes to the distinct characteristics and performance of Decision Forests and Random Forests.

2.2 Base-learner: Decision Tree

In order to induce the decision trees within both Decision Forests and Random Forests, the base-learner employed is the CART (Classification and Regression Trees) method. CART is an algorithm that recursively partitions the dataset into subsets based on the values of input features, aiming to minimize impurity within each subset. The CART method works by iteratively selecting the feature and the split point that results in the greatest reduction of impurity, typically measured using the GINI index. The GINI index quantifies the impurity of a set of data points by measuring the probability of misclassifying a randomly chosen element from the set. The formula for the GINI impurity is

$$I_G = 1 - \sum_{j=1}^{c} p_j^2$$

Therefore, the CART algorithm effectively constructs decision trees by repeatedly partitioning the dataset into increasingly homogeneous subsets until a stopping criterion, such as a maximum tree depth or minimum number of samples per leaf, is reached. By utilizing the CART method with the GINI measure, Decision Forests and Random Forests can efficiently learn complex decision boundaries while minimizing impurity, thereby enhancing their predictive accuracy and robustness.

3 Algorithms implementation

The upcoming section will feature the pseudo-code of the implemented algorithms for both Random Forest and Decision Forest techniques. A structured outline of the step-by-step process involved in constructing these ensemble classifiers will be provided, encompassing key components such as tree induction, feature selection, and dataset sampling.

3.1 Decision Forest

3.1.1 Fit function

```
FUNCTION fit(X, y):
    trees = []
    FOR 1:NT:
        subset_feat = randomly_select_feat(n_feat, F)
        tree = DecisionTree(MAX_DEPTH, subset_feat)
        tree.fit(X, y)
        trees.append(tree)
    END FOR

    feature_counts = dict{}
    FOR EACH tree IN trees:
        FOR EACH feature, freq IN tree.feature_freq:
            IF feature NOT IN feature_counts:
                feature_counts[feature] = 0
            feature_counts[feature] += freq
        END FOR
    END FOR

    feature_importances = dict{}
    FOR EACH feature, count IN feature_counts:
        feature_importances[feature] = count / NT
    END FOR

    feature_importances = sort_descending(feature_importances)
END FUNCTION
```

The `fit()` function from the Decision Forest implementation is used to train the ensemble classifier following the Ho's proposal [1998]. The method accepts two arrays, one being the whole feature dataset X $(N \times M)$ and the other one being the target labels y which need to

be encoded to integers. The *.csv* file is entered into the wrapper `process_dataset()` instead of to the fitting function.

In terms of the algorithm implementation, the method basically creates a list of trees forming the forest itself. Each tree is fitted using the CART method and fed with all the samples of the training set but a subset of the features defined by F (number of random features used in each tree). This behaviour is the key component of the Decision Forest algorithm. Subsequently, for each tree the frequency of best features defined by the best GINI value are counted, therefore having an ordered list of the most crucial features. It is important to note that for medium/big datasets the trees are built in parallel.

3.1.2 CART fit function

```
FUNCTION fit(X, y):
    tree = grow_tree(X, y)
END FUNCTION

FUNCTION grow_tree(X, y, depth=0):
    n_labels = count_unique_elements(y)

    IF n_labels == 1:
        RETURN {'prediction': y[0]}

    IF depth == MAX_DEPTH:
        RETURN {'prediction': most_frequent_element(y)}

    best_feature, best_threshold = find_best_criteria(X, y)

    left_idx, right_idx = split_data(X, best_feature, best_threshold)

    left_subtree = grow_tree(X[left_idx], y[left_idx], depth + 1)
    right_subtree = grow_tree(X[right_idx], y[right_idx], depth + 1)

    RETURN {'feature': best_feature,
            'threshold': best_threshold,
            'left': left_subtree,
            'right': right_subtree}
END FUNCTION

FUNCTION find_best_criteria(X, y, feature_idx):
    best_gini = infinity

    FOR EACH feature_idx IN feature_idx:
        thresholds = midpoints_between_contiguous_sorted_values(X)
        FOR EACH threshold IN thresholds:
            left_idx, right_idx = split_data(X, feature_idx, threshold)
            gini = gini_impurity(y[left_idx]) * n_left_idx / length(y)
                + gini_impurity(y[right_idx]) * n_left_idx / length(y)
            if gini < best_gini:
                best_gini = gini
                best_feature = feature_idx
                best_threshold = threshold
        END FOR
    END FOR
```

```
    RETURN best_feature, best_threshold
END FUNCTION

FUNCTION gini_impurity(y):
    counts = get_unique_elements_counts(y)
    probabilities = calculate_probabilities(counts, length(y))
    return 1 - sum(probabilities^2)
END FUNCTION
```

Regarding the `fit()` function for the CART, the method just grows a single tree. The tree grows computing the best feature and threshold by just trying all the combinations, and taking the one with the lowest GINI index, *i.e.*, the lower the likelihood of misclassifying a random sample. Once the best combination of feature/threshold is chosen, the data is splitted according to the threshold and two more trees (split of nodes) are grown with this subspace of the dataset. The node splitting is stopped if the labels for all the samples of the subset are the same, or if a MAX_DEPTH value is achieved.

3.1.3 Predict function

```
FUNCTION predict(X):
    predictions = []
    FOR EACH x IN X:
        prediction = predict_tree(x, self.tree)
        predictions.append(prediction)
    RETURN predictions
END FUNCTION

FUNCTION predict_tree(x, tree):
    IF 'prediction' IN tree:
        RETURN tree['prediction']

    feature_value = x[tree['feature']]
    IF feature_value < tree['threshold']:
        return predict_tree(x, tree['left'])
    ELSE:
        return predict_tree(x, tree['right'])
END FUNCTION
```

The `predict()` method from CART receives a dataset to predict and iterates trough each sample to call the `predict_tree()`, that returns the label corresponding to its prediction. This method primarily selects the deciding feature of the node and recursively goes to the following node split depending on the side the value falls considering the threshold of the current node. When the sample arrives at a leaf, it just returns the label associated to it. Therefore, the `predict()` function returns a list containing all the inferred labels.

Once a list of labels for each sample and tree are predicted, the ensemble classifier returns as the final predictions the one most repeated among the trees, *i.e.*, performs a majority voting technique for each sample to infer.

3.2 Random Forest

3.2.1 Fit function

```
FUNCTION fit(X, y):
    trees = []
    FOR 1:NT:
        bootstrap_idx = randomly_select_sample_idx(X)
        bootstrap_X = select_elements(X, bootstrap_idx)
        bootstrap_y = select_elements(y, bootstrap_idx)
        tree = DecisionTree(MAX_DEPTH, F)
        tree.fit(bootstrap_X, bootstrap_y)
        trees.append(tree)
    END FOR

    feature_counts = dict{}
    FOR EACH tree IN trees:
        FOR EACH feature, freq IN tree.feature_freq:
            IF feature NOT IN feature_counts:
                feature_counts[feature] = 0
            feature_counts[feature] += freq
    END FOR

    feature_importances = dict{}
    FOR EACH feature, count IN feature_counts:
        feature_importances[feature] = count / NT
    END FOR

    feature_importances = sort_descending(feature_importances)
END FUNCTION
```

The `fit()` function from the Random Forest implementation is used to train the classifier following the Breiman's proposal [2001]. As the previous `fit()` function described, the method accepts two arrays, one being the whole feature dataset X ($N \times M$) and the other one being the target labels y which need to be encoded to integers.

The main difference in terms of implementation is the bootstrapped sampling of the original training set to fit a single Decision Tree, also following the CART method. In this case, we randomly select some samples from the original set and drop them to train a tree, although we preserve all the features which will be randomly selected later when splitting the nodes. Again, for medium/big datasets the trees are built in parallel.

3.2.2 CART fit function

```
FUNCTION grow_tree(X, y, F, depth=0):
    n_labels = count_unique_elements(y)

    IF n_labels == 1:
        RETURN {'prediction': y[0]}

    IF depth == MAX_DEPTH:
        RETURN {'prediction': most_frequent_element(y)}

    feat_subset = randomly_select_feat(n_feat, F)
```

```
    best_feature, best_threshold = find_best_criteria(X, y, feat_subset)

    left_idx, right_idx = split_data(X, best_feature, best_threshold)

    left_subtree = grow_tree(X[left_idx], y[left_idx], depth + 1)
    right_subtree = grow_tree(X[right_idx], y[right_idx], depth + 1)

    RETURN {'feature': best_feature,
            'threshold': best_threshold,
            'left': left_subtree,
            'right': right_subtree}
END FUNCTION
```

When it comes to the decision tree growing, the implementation is the same as for the Decision Forest with the exception of the subset of features which is selected at every node split, instead of being passed to the method and be the same for all the nodes split.

3.2.3 Predict function

The `predict()` function for the Random Forest implementation is the same as the one from the Decision Forest, since the navigation across the nodes is the same.

4 Evaluation of the ensemble classifiers

In this section, we'll examine the outcomes of applying the algorithm to three datasets sourced from the UCI repository: ecoli, wine, and segment. It's worth noting that there were no missing values, thus no preprocessing was necessary. Additionally, the data was divided randomly and proportionally into training and testing sets, with an 80/20 split.

The methodology utilized was as follows: Various forests were trained with different dimensions, specifically $NT = 1, 10, 25, 50, 75, 100$. For decision forests, the number of features tested were $F = int(\frac{M}{4}), int(\frac{3M}{2}), int(\frac{3M}{4}), Runif(1, M)$, where M represents the number of features/attributes in a given dataset, and $Runif()$ is a function that generates a random number within a specified range following a uniform distribution. For random forests, the lengths of the subsets of features tested were denoted by $F = 1, 3, int(log_2 M + 1), \sqrt{M}$.

4.1 Small dataset - Ecoli

The dataset Ecoli pertains to protein localization sites, containing attributes derived from various methods of signal sequence recognition and analysis of amino acid content. It comprises 336 instances for the E.coli dataset, each with 7 predictive attributes and 1 name attribute. Attributes include methods like McGeoch's and von Heijne's for signal sequence recognition, scores from discriminant analysis, and predictions from membrane spanning region programs. The dataset aims to predict the localization site of proteins, with classes including cytoplasm, inner membrane, perisplasm, outer membrane, and various lipoproteins. There are 143 instances of cytoplasm, 77 of the inner membrane without signal sequence, 52 of perisplasm, 35 of inner membrane with uncleavable signal sequence, 20 of outer membrane, and small numbers of outer and inner membrane lipoproteins with or without cleavable signal sequences.

4.1.1 Decision forest

Table 1: Performance Results for ECOLI Dataset with Decision Forest

NT	F	Fit Time (s)	Accuracy (%)	Feature Importances
1	1	0.00	42.65	lip
1	3	0.07	42.65	gvh, aac
1	5	0.11	70.59	aac, alm2, mcg, alm1
1	runif	0.02	47.06	aac
10	1	0.14	42.65	mcg, alm1, alm2, gvh, aac, lip
10	3	0.62	64.71	aac, gvh, mcg, alm1, alm2, lip
10	5	1.05	67.65	aac, alm2, gvh, alm1, mcg, lip
10	runif	0.72	66.18	gvh, alm1, alm2, aac, mcg, lip
25	1	0.41	45.59	gvh, mcg, aac, alm1, alm2, lip
25	3	1.90	69.12	alm1, mcg, gvh, alm2, aac, lip
25	5	2.74	66.18	gvh, aac, mcg, alm1, alm2, lip
25	runif	1.37	55.88	gvh, mcg, aac, alm1, alm2, lip
50	1	0.71	44.12	alm2, alm1, aac, gvh, mcg, lip
50	3	3.05	66.18	aac, gvh, mcg, alm1, alm2, lip
50	5	5.66	69.12	mcg, gvh, aac, alm2, alm1, lip
50	runif	3.68	70.59	alm2, gvh, alm1, mcg, aac, lip
75	1	1.15	44.12	alm2, mcg, alm1, gvh, aac, lip
75	3	5.24	67.65	mcg, gvh, aac, alm1, alm2, lip
75	5	8.07	69.12	aac, gvh, alm1, alm2, mcg, lip
75	runif	4.60	69.12	gvh, mcg, alm1, aac, alm2, lip
100	1	1.47	42.65	alm2, alm1, gvh, mcg, aac, lip
100	3	6.89	69.12	alm1, gvh, alm2, aac, mcg, lip
100	5	10.84	70.59	aac, gvh, alm2, mcg, alm1, lip
100	runif	6.41	70.59	aac, alm2, gvh, alm1, mcg, lip

Table 1 shows the results for the Decision Forest interpreter with all the executions. As expected, the training times are low due to the size of the dataset (336 samples and 7 features). When it comes to the accuracy, it varies a lot among executions, ranging from $\approx$ 42% to $\approx$ 42%. The highest accuracies are achieved when the F is the highest (5), being almost identical independently of the size of the forest. Regarding the feature importance, it is clear that least decisive attribute is lip. Considering that are 8 types of classes to predict and it quite an unbalanced dataset, the results are quite good.

4.1.2 Random forest

Table 2: Performance Results for ECOLI Dataset with Random Forest

NT	F	Fit Time (s)	Accuracy (%)	Feature Importances
1	1	0.01	61.76	alm1, mcg, gvh, lip, alm2, aac
1	2	0.03	66.18	mcg, aac, alm2, alm1, gvh, lip
1	3	0.04	64.71	mcg, alm2, alm1, gvh, aac, lip
1	2	0.03	69.12	gvh, alm2, mcg, aac, alm1, lip
10	1	0.09	61.76	mcg, alm2, alm1, aac, gvh, lip
10	2	0.31	72.06	mcg, alm2, alm1, aac, gvh, lip
10	3	0.46	72.06	mcg, gvh, alm1, alm2, aac, lip
10	2	0.33	72.06	mcg, alm1, alm2, aac, gvh, lip
25	1	0.29	63.24	aac, mcg, alm2, alm1, gvh, lip
25	2	0.79	70.59	mcg, alm1, aac, gvh, alm2, lip
25	3	1.19	69.12	mcg, alm1, aac, gvh, alm2, lip
25	2	0.83	72.06	aac, alm1, alm2, mcg, gvh, lip
50	1	0.53	66.18	mcg, aac, alm1, gvh, alm2, lip
50	2	1.66	70.59	mcg, alm1, gvh, alm2, aac, lip
50	3	2.33	70.59	mcg, alm1, gvh, aac, alm2, lip
50	2	1.65	70.59	alm1, mcg, gvh, aac, alm2, lip
75	1	0.79	64.71	mcg, alm1, aac, gvh, alm2, lip
75	2	2.49	72.06	mcg, alm1, alm2, aac, gvh, lip
75	3	3.49	73.53	mcg, alm1, aac, gvh, alm2, lip
75	2	2.42	67.65	mcg, gvh, alm1, aac, alm2, lip
100	1	0.94	69.12	alm2, aac, mcg, gvh, alm1, lip
100	2	3.16	69.12	mcg, aac, alm1, alm2, gvh, lip
100	3	4.76	72.06	mcg, aac, alm1, gvh, alm2, lip
100	2	3.24	70.59	gvh, alm1, mcg, aac, alm2, lip

Table 2 shows an increase in accuracy while keeping lower training times. The accuracy scores varies from $\approx 61\%$ to $\approx 72\%$. Interestingly, the results are more consistent across the experiments, probably due to the fact that random forest implements a bootstrap technique to grow the trees and selects a subset of features for each split. These two characteristics can add an extra generalization that yield better results. Again, the least important feature is lip and mcg is quite repeated as the most important one, a pattern that we could not observe for the Decision Forest results,

4.2 Medium dataset - Wine

The dataset comprises physicochemical and sensory variables from red and white variants of Portuguese "Vinho Verde" wine. With input variables derived from various physicochemical tests such as fixed acidity, volatile acidity, and alcohol content, the dataset offers insights into wine quality prediction. The output variable, quality, represents sensory data scored between 0 and 10. The datasets, available from the UCI machine learning repository and referenced to Cortez et al. (2009), present both classification and regression tasks, with classes ordered but unbalanced, reflecting more normal wines than excellent or poor ones. Notably, due to privacy and logistic concerns, the dataset excludes information on grape types, wine brand,

and selling prices.

The features are:

1 - fixed acidity ; 2 - volatile acidity ; 3 - citric acid ; 4 - residual sugar ; 5 - chlorides ; 6 - free sulfur dioxide ; 7 - total sulfur dioxide ; 8 - density ; 9 - pH ; 10 - sulphates ; 11 - alcohol

Where each one is represented by VX where X is the number of the feature.

4.2.1 Decision forest

Table 3: Performance Results for WINE Dataset with Decision Forest

NT	F	Fit Time (s)	Accuracy (%)	Feature Importances
1	2	0.75	55.31	V7, V5
1	5	0.91	60.62	V1, V4, V7, V9, V10
1	8	1.41	60.62	V3, V4, V5, V6, V1, V8, V9, V10
1	runif	1.63	58.13	V5, V7, V9, V2, V8, V6, V1, V4
10	2	0.61	60.62	V10, V5, V7, V9, V1, V6, V4, V8, V11, V3, V2
10	5	1.72	63.12	V8, V1, V7, V9, V2, V4, V5, V6, V10, V3, V11
10	8	2.23	63.44	V9, V3, V6, V2, V7, V11, V4, V1, V10, V5, V8
10	runif	2.03	65.62	V3, V11, V1, V4, V6, V5, V2, V10, V9, V7, V8
25	2	1.05	61.56	V9, V8, V4, V3, V6, V11, V5, V2, V10, V1, V7
25	5	2.95	63.75	V2, V1, V8, V7, V5, V9, V6, V11, V10, V3, V4
25	8	4.93	66.56	V9, V5, V3, V7, V4, V8, V10, V6, V1, V11, V2
25	runif	3.26	65.31	V9, V2, V7, V1, V4, V6, V5, V3, V11, V10, V8
50	2	2.04	61.56	V3, V9, V5, V2, V8, V10, V1, V7, V11, V4, V6
50	5	6.14	67.19	V2, V8, V3, V9, V1, V5, V4, V6, V7, V11, V10
50	8	10.32	66.88	V2, V5, V8, V7, V4, V3, V9, V10, V1, V6, V11
50	runif	6.63	67.50	V1, V5, V3, V11, V2, V4, V10, V9, V7, V6, V8
75	2	2.77	61.88	V3, V7, V9, V4, V6, V10, V5, V2, V1, V8, V11
75	5	9.56	65.31	V7, V2, V8, V5, V3, V1, V4, V6, V10, V9, V11
75	8	14.43	65.94	V7, V3, V2, V4, V5, V1, V10, V9, V6, V8, V11
75	runif	10.28	67.50	V7, V8, V3, V9, V2, V5, V10, V4, V6, V1, V11
100	2	3.60	63.44	V2, V3, V7, V5, V9, V8, V4, V6, V11, V10, V1
100	5	11.30	66.88	V1, V2, V10, V9, V8, V5, V3, V4, V6, V11, V7
100	8	18.95	65.31	V2, V8, V4, V9, V5, V3, V7, V1, V10, V11, V6
100	runif	11.96	66.88	V8, V2, V3, V6, V5, V7, V4, V10, V9, V1, V11

Table 3 depicts the executions of the Decision Forest for the wine dataset. In this case, the accuracies are quite consistent ranging from $\approx 55\%$ to $\approx 67\%$. Again, the higher values of F perform better regardless of the size of the forest, except for the case that we have 1 tree, that clearly underperforms. With regard to the importance of features, V11 (alcohol) seems to be the least decisive one throughout the executions, and the most relevant between V7 (total sulfur dioxide) and V9 (pH).

4.2.2 Random forest

Table 4: Performance Results for WINE Dataset with Random Forest

NT	F	Fit Time (s)	Accuracy (%)	Feature Importances
1	1	0.18	57.50	V10, V4, V3, V1, V5, V11, V9, V2, V8, V7, V6
1	2	0.31	53.12	V11, V7, V5, V2, V8, V1, V9, V10, V3, V6, V4
1	4	0.56	52.19	V9, V11, V2, V4, V5, V7, V10, V8, V6, V3, V1
1	3	0.48	54.37	V2, V8, V3, V6, V4, V1, V5, V9, V7, V10, V11
10	1	0.29	62.81	V3, V7, V10, V4, V9, V8, V6, V11, V5, V1, V2
10	2	0.52	65.31	V2, V3, V11, V8, V7, V5, V10, V9, V4, V1, V6
10	4	0.90	66.88	V9, V5, V2, V10, V11, V7, V6, V4, V3, V1, V8
10	3	0.72	65.94	V11, V8, V10, V2, V7, V9, V5, V1, V6, V4, V3
25	1	0.63	66.25	V2, V5, V9, V7, V6, V3, V8, V10, V1, V4, V11
25	2	1.17	67.50	V2, V8, V7, V5, V10, V9, V11, V1, V3, V4, V6
25	4	2.00	65.62	V2, V10, V8, V7, V1, V11, V5, V9, V4, V3, V6
25	3	1.51	65.62	V8, V7, V5, V10, V3, V2, V9, V1, V11, V4, V6
50	1	1.20	68.44	V10, V5, V9, V2, V1, V3, V7, V11, V8, V4, V6
50	2	2.09	68.12	V7, V2, V5, V10, V9, V8, V11, V1, V3, V4, V6
50	4	4.01	65.62	V10, V2, V7, V8, V11, V5, V4, V9, V1, V3, V6
50	3	3.02	65.31	V2, V8, V10, V7, V1, V5, V11, V9, V4, V3, V6
75	1	1.66	64.38	V3, V7, V1, V5, V9, V10, V2, V8, V6, V4, V11
75	2	2.97	68.44	V7, V2, V10, V11, V8, V5, V1, V9, V3, V4, V6
75	4	5.53	65.94	V2, V7, V11, V10, V8, V5, V9, V1, V4, V3, V6
75	3	4.38	65.94	V2, V8, V11, V7, V10, V9, V5, V1, V6, V3, V4
100	1	2.38	65.62	V8, V11, V9, V7, V10, V3, V2, V4, V5, V6, V1
100	2	3.96	65.31	V2, V7, V8, V10, V11, V9 ,V1, V5, V3, V4, V6
100	4	7.38	66.25	V2, V10, V7, V11, V8, V5, V9, V1, V3, V4, V6
100	3	5.68	67.50	V7, V8, V2, V11, V10, V9, V5, V3, V1, V6, V4

Table 4 shows that the accuracies for the Random Forest experiments for the Wine dataset vary from $\approx 52\%$ to $\approx 68\%$, achieving 1% extra for the better performing one. For this dataset the differences between algorithm are not that remarkable. However, the training times are considerably lower. Surprisingly, the least important feature is not V11 like before, but V6 (free sulfur dioxide), which tells us the intrinsic differences between both algorithms.

4.3 Big dataset - Segment

The segment dataset comprises image segmentation data originating from the Vision Group at the University of Massachusetts in November 1990. It includes 2320 instances, with each instance representing a 3x3 region randomly sampled from a database of seven outdoor images. These images were manually segmented at the pixel level to assign classifications. The dataset encompasses 19 continuous attributes, featuring various measures such as centroid coordinates, pixel count, line density, edge contrast, and color-related statistics like intensity, RGB averages, and excess color measures. Notably, attributes like "value-mean," "saturation-mean," and "hue-mean" represent nonlinear transformations of RGB values. The dataset is devoid of missing attribute values and involves seven distinct classes for segmentation, including brickface, sky, foliage, cement, window, path, and grass, each with 330 instances.

4.3.1 Decision forest

Table 5: Performance Results for SEGMENT Dataset with Decision Forest

NT	F	Fit Time (s)	Accuracy (%)	Feature Importances
1	4	1.36	44.16	vedge-mean, hedge-sd, hedge-mean
1	9	2.41	93.07	rawred-mean, region-centroid-row, exblue-mean, ...
1	14	4.52	90.48	vegde-sd, region-centroid-col, rawgreen-mean, ...
1	runif	1.03	90.91	value-mean, region-centroid-row, hedge-mean, ...
10	4	2.01	94.16	saturation-mean, vegde-sd, exgreen-mean, ...
10	9	3.94	97.62	region-centroid-row, rawblue-mean, hedge-sd, ...
10	14	5.29	97.84	region-centroid-col, region-centroid-row, ...
10	runif	4.35	96.54	rawgreen-mean, hedge-mean, region-centroid-col, ...
25	4	4.08	95.02	vegde-sd, rawblue-mean, saturation-mean, ...
25	9	8.45	96.32	exred-mean, vegde-mean, hedge-mean, ...
25	14	11.95	98.27	region-centroid-row, hue-mean, region-centroid-col, ...
25	runif	7.30	95.67	exgreen-mean, vegde-sd, region-centroid-col, ...
50	4	7.97	93.72	region-centroid-col, hedge-mean, hedge-sd, ...
50	9	15.04	96.32	hue-mean, region-centroid-col, saturation-mean, ...
50	14	22.23	97.84	hue-mean, region-centroid-col, exgreen-mean, ...
50	runif	14.89	97.84	saturation-mean, hedge-sd, region-centroid-row, ...
75	4	12.84	94.16	vegde-sd, hedge-sd, value-mean, ...
75	9	25.84	97.62	hue-mean, region-centroid-row, region-centroid-col, ...
75	14	34.78	97.62	hue-mean, region-centroid-row, region-centroid-col, ...
75	runif	26.61	97.84	intensity-mean, hue-mean, rawblue-mean, ...
100	4	17.12	95.24	vegde-sd, exred-mean, region-centroid-col, ...
100	9	32.19	97.19	hue-mean, region-centroid-row, region-centroid-col, ...
100	14	45.60	97.84	hue-mean, region-centroid-col, region-centroid-row, ...
100	runif	32.07	98.05	vegde-sd, region-centroid-col, rawred-mean, ...

In Table 5 we can observe the executions for the Decision Forest for the Segment dataset. Here the accuracies do not vary a lot, ranging from $\approx$ 90% to $\approx$ 98% if we do not consider the best performing one with $\approx$ 44% when NT=1 and F=4. This tells us that this dataset is not that difficult to predict considering that it has 19 features and 7 classes. The trainign times are the highest of all the datasets, as expected. The most repeated important feature seems to be hue-mean, even though there is a wide variety of features at first position. The least decisive ones are short-line-density-2 and short-line-density-5, which cannot be seen due to a lack of space but are showed in the output of the experiment `/Data/output/segment_decision-forest_interpreter_output.txt`.

4.3.2 Random forest

Table 6: Performance Results for SEGMENT Dataset with Random Forest

NT	F	Fit Time (s)	Accuracy (%)	Feature Importances
1	1	0.14	67.10	rawblue-mean, vedge-mean, exgreen-mean, ...
1	2	0.45	90.48	vegde-sd, region-centroid-col, value-mean, ...
1	5	0.81	94.16	region-centroid-row, rawred-mean, exgreen-mean, ...
1	4	0.60	94.59	exgreen-mean, region-centroid-row, hue-mean, ...
10	1	0.35	90.69	hedge-mean, hedge-sd, vegde-sd, ...
10	2	0.73	96.32	hue-mean, region-centroid-row, intensity-mean, ...
10	5	1.33	96.75	hue-mean, region-centroid-row, exgreen-mean, ...
10	4	1.13	97.19	region-centroid-row, hue-mean, exred-mean, ...
25	1	0.66	92.86	hedge-mean, region-centroid-row, rawblue-mean, ...
25	2	1.44	96.75	region-centroid-row, hue-mean, saturation-mean, ...
25	5	3.31	97.84	region-centroid-row, hue-mean, region-centroid-col, ...
25	4	2.66	96.75	region-centroid-row, hue-mean, region-centroid-col, ...
50	1	1.31	94.81	hedge-sd, intensity-mean, hedge-mean, ...
50	2	2.82	96.75	region-centroid-row, hue-mean, exgreen-mean, ...
50	5	5.86	97.62	region-centroid-row, hue-mean, exgreen-mean, ...
50	4	4.89	97.62	region-centroid-row, hue-mean, region-centroid-col, ...
75	1	1.66	93.94	exred-mean, hedge-sd, region-centroid-col, ...
75	2	3.98	96.75	region-centroid-row, exgreen-mean, hue-mean, ...
75	5	8.90	98.05	region-centroid-row, hue-mean, region-centroid-col, ...
75	4	7.00	98.05	region-centroid-row, hue-mean, saturation-mean, ...
100	1	2.40	93.29	vegde-sd, exblue-mean, hedge-mean, ...
100	2	5.41	97.19	region-centroid-row, hue-mean, exred-mean, ...
100	5	10.89	97.84	region-centroid-row, hue-mean, region-centroid-col, ...
100	4	9.92	97.40	region-centroid-row, hue-mean, exred-mean, ...

Finally, Table 6 shows the executions for the Segment dataset and Random Forests. The accuracies range from $\approx$ 90% to $\approx$ 98%, the same as the Decision Forest. Nevertheless, the lower accuracy improves being $\approx$ 67% when NT=1 and F=1. These similarities in performance may be due to the ease of predicting dataset. Once more, the training times are considerably lower compared to the ones of the Decision Forest. As for feature importance, here we can see a clear winner, region-centroid-row. The least decisive ones are the same ones as before, short-line-density-2 and short-line-density-5, as it can be seen in `/Data/output/segment_random-forest_interpreter_output.txt`.

5 Code execution

In order to execute the code it is recommended to use a Python (version >= 3.8) environment with the packages installed from the `requirements.txt` file inside the `Source/` directory. Once having the python environment activated just run the command

```
pip install -r requirements.txt
```

The Decision/Random forest interpreter can be executed by running the script `main.py` from

the `Source/` directory. It shows a menu to choose which algorithm/dataset pair to process and then saves the results in `Data/output` and prints them. On the other hand, in order to preprocess the datasets one can run the `process_datasets.py` from `Source/` and choose from the menu what dataset to preprocess.

6 Conclusions

To summarize, random forests outperform decision forests due to their enhanced capabilities and superior outcomes. This could be attributed to several factors. Firstly, random forests employ bagging (bootstrap aggregating) individually for each tree, leading to greater specialization in various instances and thereby offering more diverse criteria for classifying new instances. Additionally, while both types of forests incorporate randomness, random forests inherently exhibit more randomness as they repeatedly select which features to consider at each node. As far as I understand, increased randomness in model training reduces the likelihood of overfitting to dataset noise, thereby enhancing generalization.

Based on both experimental results and personal insight, I would opt to use half the number of features in the dataset when selecting hyperparameters. This encourages more diversity in choices, as the best feature might not always be available. Regarding the number of trees, I would recommend training forests comprising a minimum of 20 trees, ranging up to 100 (more consistent results and the impact in training time is low). This approach ensures that any underperforming or poorly initialized trees have minimal impact on overall performance, thereby enhancing the robustness of the forest.