1 INTRODUCTION

A decision tree and a random forest have been implemented from scratch to classify mushrooms as edible (e) or poisonous (p), using customizable impurity criteria and cross-validation techniques to select the best hyperparameters. The goal is to build an interpretable and effective model capable of accurately distinguishing edible mushrooms from potentially dangerous ones based on their characteristics.

The dataset used for this task contains various features describing mushroom characteristics, including cap diameter, gill color, stem width, and more. The implementation involves a data cleaning phase, feature selection, and the development of decision trees using different impurity functions, such as Gini, scaled entropy, and a custom impurity measure.

To evaluate model performance and avoid overfitting, k-fold cross-validation is applied, testing multiple hyperparameter combinations. The decision tree serves as a baseline model, while the random forest, as an ensemble method, leverages multiple trees to improve prediction stability and accuracy.

2 DATASET DESCRIPTION

The dataset consists of 61,069 rows, and each row is labeled as either edible (e) or poisonous (p). There are 20 features used to describe each mushroom. Of these, 17 are nominal (categorical), and 3 are metrical (numerical). The variables are listed in Table 1.

Table 1: Description of Mushroom Dataset Variables

Variable	**Description**	**Values**
cap-diameter	Diameter of the mushroom cap in centimeters	Continuous
cap-shape	Shape of the mushroom cap	bell (b), conical (c), convex (x), flat (f), sunken (s), spherical (p), other (o)
cap-surface	Surface texture of the cap	fibrous (f), grooved (g), scaly (y), smooth (s), shiny (h), leathery (l), silky (k), sticky (t), wrinkled (w), fleshy (e)
cap-color	Color of the cap	brown (n), buff (b), gray (g), green (r), pink (p), purple (u), red (e), white (w), yellow (y), blue (l), orange (o), black (k)
does-bruise-bleed	Indicates if the mushroom bruises or bleeds	t = yes, f = no
gill-attachment	How the gills attach to the stem	adnate (a), adnexed (x), decurrent (d), free (e), sinuate (s), pores (p), none (n)
gill-spacing	Spacing of the gills	close (c), distant (d), or none (f)
gill-color	Color of the gills	Same as cap-color + none (f)
stem-height	Height of the mushroom stem	Continuous
stem-width	Width of the stem	Continuous
stem-root	Shape of the root at the base of the stem	bulbous (b), swollen (s), club (c), cup (u), equal (e), rooted (r), rhizomorphs (z)
stem-surface	Surface texture of the stem	Same as cap-surface + none (f)
stem-color	Color of the stem	Same as cap-color + none (f)
veil-type	Type of veil	partial (p), universal (u)
veil-color	Color of the veil	Same as cap-color + none (f)
has-ring	Presence of a ring around the stem	ring (t), none (f)

Variable	**Description**	**Values**
ring-type	Type of ring present	cobwebby (c), evanescent (e), flaring (r), grooved (g), large (l), pendant (p), sheathing (s), zone (z), scaly (y), movable (m), none (f), unknown (?)
spore-print-color	Color of the spore print	Same as cap-color
habitat	Habitat where the mushroom is found	grasses (g), leaves (l), meadows (m), paths (p), heaths (h), urban (u), waste (w), woods (d)
season	Season when the mushroom grows	spring (s), summer (u), autumn (a), winter (w)

3 DATA CLEANING AND DATA PREPARATION

The following procedures were adopted for cleaning and preparing the dataset:

1. The dataset was split into a training set (80%) and a test set (20%) to evaluate the model's performance in generalization, and the split was performed randomly using a fixed seed to ensure reproducibility of results.
2. The dataset was inspected to assess the presence of missing values and duplicate rows, and the data cleaning was performed separately on the training and test sets, rather than on the entire dataset:
   (a) For each column the percentage of missing values was calculated, and all the columns with a percentage higher than 40% were dropped (6 columns were dropped).
   (b) After the critical column were eliminate, all rows containing null values in the remaining features were removed (rows dropped from the training set were 19,343, while from the test set 4,746).
   (c) Duplicate rows were removed after handling missing values (there were 146 duplicate rows).
3. The variables were divided into categorical and numerical based on their data type: a data structure was built to map each feature to its type, essential for implementing the decision tree splitting algorithm.
4. The dataset was divided into independent features (**X**) and the target variable (**y**) required for training the classification model:
   (a) The `class` column was converted into a binary variable, with the value "1" for poisonous mushrooms (p) and "0" for edible ones (e).
   (b) The training (X_{train}, y_{train}) and test (X_{test}, y_{test}) sets were defined.

At the end of the cleaning phase, the dataset underwent a reduction in the number of rows and columns, and the new dimensions are:

- **Training set**: (29,596, 15).
- **Test set**: (7,384, 15).

4 DECISION TREE IMPLEMENTATION

To build the decision tree, two classes have been implemented: `Tree_Node` and `Decision_Tree`.

4.1 TREE_NODE CLASS

The decision tree is built using a recursive approach, where each node plays a fundamental role in partitioning the dataset and guiding the classification process. Each node can either be an internal node or a leaf node:

- An *internal* node is responsible for making a decision based on a feature and a threshold.
- A *leaf* node represents a final classification outcome, in this case if the mushroom is poisonous or not.

The node structure consists of the following key elements:

1. *Impurity Function*

   Impurity functions measure the homogeneity of a node in the decision tree and help determine the optimal splitting criterion for the dataset. The following impurity functions are implemented:

   - *Gini function*:
   $$H(p) = 1 - \sum_i p_i^2$$
   - *Scaled entropy*:
   $$H(p) = -\frac{p}{2}\log_2(p) - \frac{(1-p)}{2}\log_2(1-p)$$
   - *Custom impurity*:
   $$H(p) = \sqrt{p(1-p)}$$

   The function `select_impurity_function` allows for the selection of the desired impurity function based on input parameter, meaning that if `impurity_type = 'gini'`, it will return the `gini_impurity` function.

2. *Splitting Criterion* `(split_dataset)`

   Each internal node selects a feature and a threshold to divide (split) the dataset into two subsets. The process varies depending on the type of feature:

   - For *numerical features*, data is split at a threshold value, where:
     - The left node contains samples with values $\leq$ to the threshold.
     - The right node contains samples $>$ than the threshold.
   - For *categorical features*, data is split based on equality:
     - The left node contains samples that match the threshold category.
     - The right node contains the remaining samples.

   This selection is based on an impurity function, which determines the quality of a split (the split that minimizes the impurity is chosen).

3. *Best split* `(best_split)`

   To determine the best feature and threshold, the function `best_split` evaluates multiple potential splits. So, the function:

   (a) Iterates over all features in the dataset.

   (b) Determines possible thresholds:
   - For numerical features, it uses percentiles (25th, 50th, 75th) to define split points.
   - For categorical features, it considers unique values as possible thresholds.

   (c) Then the function splits the dataset using `split_dataset` and calculates the impurity for each split.

   (d) Computes the weighted impurity for left and right nodes.

   (e) Selects the split that minimizes impurity, storing the best feature and threshold along with the resulting subsets.

This recursive process continues until a stopping condition is met: a node becomes a leaf when no further meaningful splits can be made, and this can happen when:

- All samples in a node belong to the same class.
- A predefined maximum depth is reached.
- The number of samples in a node falls below a minimum threshold.

4.2 DECISION_TREE CLASS

The decision tree structure consists of the following key elements:

1. *Class initialization*

   The model is initialized with several key parameters:

   (a) *max_depth*, that specifies the maximum depth of the tree: setting a maximum value prevents the tree from growing excessively, reducing the risk of overfitting.

   (b) *min_samples_split*, that indicates the minimum number of samples required to split a node: if a node contains fewer samples than this threshold, it becomes a leaf.

   (c) *impurity_type*, that determines which impurity function will be used to evaluate the quality of node splits.

2. *Training process* `(fit)`

   The `fit` function takes the dataset X (containing features) and the vector y (containing class labels, poisonous or edible) as input. The tree is constructed recursively using the `_build_tree` function, which starts from the root and progressively splits the dataset until a stopping condition is met.

3. *Tree construction* `(_build_tree)`

   Before splitting the dataset, the function checks whether it is necessary to stop building the tree. The main stopping conditions are:

   (a) *Node purity*, if all instances in the node belong to the same class, the node becomes a leaf.

   (b) *Minimum number of samples*, if the number of samples in the node is below `min_samples_split`, the splitting stops, and the node becomes a leaf.

   (c) *Maximum depth reached*, if the depth value is equal to or greater than `max_depth`, the node becomes a leaf.

   If any of these conditions are met, the function creates a leaf node and assigns as its prediction the most frequent class among the samples present. If stopping conditions are not met, the best feature and threshold for splitting the data are selected, using the `best_split` function. Once the best split is determined, the data is separated into two subsets:

   (a) *Left node*, that contains samples with values $\leq$ to the threshold.

   (b) *Right node*, that contains samples with values $>$ than the threshold.

   The process continues recursively for both nodes until a stopping condition is met.

4. *Prediction* `(predict)`

   The `predict` function takes a dataset X as input and applies the `_predict_single` function to each observation. If the node is a leaf, the function returns the predicted class, while if it is an internal node:

   (a) If the feature is numerical, it checks if the value is $\leq$ or $>$ than the threshold.

   (b) If the feature is categorical, it verifies if value = threshold.

   The process continues until a leaf is reached, providing the final classification.

4.3 K-FOLD AND CROSS VALIDATION

The process of training and evaluating a decision tree involves carefully selecting hyperparameters that influence the tree's depth, splitting criteria, and impurity measures. To ensure robust evaluation and minimize biases, it was implemented a K-Fold with Cross-Validation approach, which systematically partitions the dataset while preserving class distributions: this method allows to assess different configurations of the decision tree and select the optimal combination of parameters.

The function `stratified_kfold_split` is responsible for dividing the dataset into k equally sized folds, ensuring that each fold maintains the same proportion of class labels as the original dataset: this is crucial when dealing with imbalanced data, as it prevents one class from being overrepresented in some folds while underrepresented in others. The function first extracts class information by retrieving the unique class labels in `y` and their corresponding frequencies. Then, to ensure that each fold maintains a balanced distribution of class labels, the process of distributing samples across folds follows a systematic approach:

1. For each unique class label in the dataset, the indices of all samples belonging to that class are extracted, making sure that the splitting process is aware of the different categories present in the dataset. The extracted indices are then shuffled randomly, since datasets might have structured ordering.
2. The indices are assigned cyclically to the k different folds, ensuring that each fold receives an approximately equal number of samples from each class, preserving class balance across all training and test sets.

At the end, the function returns a list where each element contains the indices corresponding to one of the k folds.

Once the dataset is split into k folds using the `stratified_kfold_split`, the `cross_validation` function systematically evaluates different hyperparameter combinations to identify the best-performing decision tree. The function explores all combinations of the following hyperparameters:

1. *max_depth*, for controlling overfitting, with the values of $[5, 10, 15]$.
2. *min_samples_split*, that regulates the robustness of the tree. It defines the minimum number of samples required to perform a split at a node and the values of $[2, 5, 10]$ were tested.
3. *impurity*, which specifies the impurity criterion used to split tree nodes. The criteria used were ["*gini*", "*scaled_entropy*", "*custom_impurity*"].

For each hyperparameter combination, the function trains a decision tree on `X_train`, `y_train` with the selected parameters, computes accuracy and error on both the training and test sets, and stores the results for each fold. At the end, for each hyperparameter combination, the function calculates the average across folds for the following metrics: mean_train_accuracy, mean_test_accuracy, mean_train_error and mean_test_error (the average error has been computed with a membership test: $1 -$ `mean_accuracy`).

4.4 RESULTS

Table 2: Hyperparameter Tuning with Cross-Validation

max depth	**min samples split**	**impurity**	**mean train accuracy**	**mean test accuracy**	**mean train error**	**mean test error**
15	5	gini	0.995025	0.993648	0.004975	0.006352
15	10	gini	0.994788	0.993580	0.005212	0.006420
15	2	gini	0.995244	0.993580	0.004756	0.006420
15	5	scaled_entropy	0.964987	0.963238	0.035013	0.036762

max depth	**min samples split**	**impurity**	**mean train accuracy**	**mean test accuracy**	**mean train error**	**mean test error**
15	2	scaled_entropy	0.965038	0.963238	0.034962	0.036762
15	10	scaled_entropy	0.964725	0.963103	0.035275	0.036897
10	5	gini	0.954690	0.953507	0.045310	0.046493
10	2	gini	0.954698	0.953507	0.045302	0.046493
10	10	gini	0.954588	0.953474	0.045412	0.046526
15	10	custom_impurity	0.945136	0.944790	0.054864	0.055210
15	5	custom_impurity	0.945254	0.944688	0.054746	0.055312
15	2	custom_impurity	0.945271	0.944688	0.054729	0.055312
10	5	scaled_entropy	0.885652	0.884141	0.114348	0.115859
10	2	scaled_entropy	0.885652	0.884141	0.114348	0.115859
10	10	scaled_entropy	0.885567	0.884073	0.114433	0.115927
10	5	custom_impurity	0.850385	0.851062	0.149615	0.148938
10	2	custom_impurity	0.850385	0.851062	0.149615	0.148938
10	10	custom_impurity	0.850385	0.851062	0.149615	0.148938
5	10	gini	0.801248	0.800446	0.198752	0.199554
5	5	gini	0.801248	0.800446	0.198752	0.199554
5	2	gini	0.801248	0.800446	0.198752	0.199554
5	2	scaled_entropy	0.740345	0.739695	0.259655	0.260305
5	10	scaled_entropy	0.740345	0.739695	0.259655	0.260305
5	5	scaled_entropy	0.740345	0.739695	0.259655	0.260305
5	10	custom_impurity	0.739855	0.738985	0.260145	0.261015
5	5	custom_impurity	0.739855	0.738985	0.260145	0.261015
5	2	custom_impurity	0.739855	0.738985	0.260145	0.261015

The results show that models with a maximum depth of 15 (`max_depth = 15`) and a minimum number of samples per split (`min_samples_split`) of 5 or 10 achieve the best performance, with a train accuracy of approximately 99.5% and a test accuracy of around 99.3%: the model effectively learns the relationships within the data without exhibiting excessive overfitting.

An interesting aspect concerns the choice of the impurity function: the Gini criterion proved to be the most effective, delivering better results than `scaled_entropy` and `custom_impurity`, indicating that Gini is the most suitable function for this dataset.

Regarding `min_samples_split`, we observe that excessively low values (`min_samples_split = 2`) tend to make the tree more sensitive to noise, while the optimal value appears to be 5, which balances model flexibility and robustness.

Based on these results, the best model has the following hyperparameters:

- max_depth = 15
- min_samples_split = 5
- impurity = gini
- mean_train_accuracy = 0.995244
- mean_test_accuracy = 0.993580
- mean_train_error = 0.004756

- mean_test_error = 0.006420

The Decision Tree proved to be a highly effective model for classifying this dataset, achieving high performance while maintaining good interpretability.

5 RANDOM FOREST IMPLEMENTATION

The Random Forest algorithm is an ensemble learning method based on decision trees, designed to enhance the robustness and accuracy of predictions compared to a single decision tree. The `Random_Forest` class implements this model by constructing multiple decision trees on randomly selected subsets of the dataset and combining their predictions using a majority voting mechanism:

1. *Class Initialization*
   The model is initialized with several key parameters:
   - *num_trees*: specifies the number of decision trees to be built in the forest; a higher number can improve the model's stability but increases computational cost.
   - *max_depth*: maximum depth of each tree; a very high value may lead to overfitting, while a low value might result in underfitting.
   - *min_samples_split*: the minimum number of samples required to split an internal node, this helps control overfitting.
   - *min_samples_leaf*: refers to the minimum number of samples required to be in a leaf node.
   - *num_features*: determines how many features are randomly selected at each split when constructing a tree.
   - *max_features*: controls the maximum number of features used when searching for the best split.
   - *impurity*: criterion used to measure node impurity.
2. *Training the model* `(fit)`
   The `fit` method is responsible for training the random forest: it begins by initializing an empty list that will store the trained decision trees. The algorithm then iterates a predefined number of times, corresponding to the number of trees in the ensemble. During each iteration, a new instance of the `Decision_Tree` class is created, configured with the specified parameters. To introduce variability, a bootstrap sample of the dataset is generated by randomly selecting rows with replacement. This ensures that each tree is trained on a slightly different subset of data. Additionally, to enhance the model's diversity and reduce overfitting, a random subset of features is selected for training each tree. This process limits the number of attributes considered at each split, encouraging different trees to focus on different aspects of the data. Once trained, the decision tree is added to the ensemble, along with the indices of the features used for training, ensuring that each tree can later be applied consistently to new data.
3. *Bootstrapping and feature selection* `(bootstrap_samples)`
   The `bootstrap_samples` method is used to generate random subsets of the dataset for training individual trees. It randomly selects rows from the dataset with replacement, maintaining the same number of samples as the original dataset. At each node split within the trees, a random subset of features is selected: by limiting the number of features considered at each split, the algorithm introduces additional randomness, making individual trees more diverse and reducing correlation between them.
4. *Making predictions* `(predict, majority_vote)`
   This process consists of two main steps: obtaining individual tree predictions and aggregating them through majority voting.

   First, the `predict` method is responsible for generating predictions for a given dataset. Each trained decision tree is applied to the input data but only using the subset of features that were selected during the tree's training. For each tree, the model extracts the relevant features from the dataset and then calls the tree's `predict` method to generate predictions. These predictions are stored in a list and then converted into a matrix, where each row corresponds to a sample and each column represents the prediction from a different tree.

Once all trees have made their predictions, the final classification is determined using the `majority_vote` method: this function processes the collected predictions by evaluating, for each sample, the most frequently occurring class label among the individual tree predictions by identifying the unique values in the prediction set and counting their occurrences. The class with the highest count is selected as the final prediction for that sample.

5.1 K-FOLD AND CROSS-VALIDATION

The function `cross_validation_forest` performs k-fold stratified cross-validation to evaluate and optimize the hyperparameters of a random forest classifier. This process systematically tests different combinations of hyperparameters to determine the best configuration for maximizing accuracy while maintaining generalization.

The function begins by calling `stratified_kfold_split`, which splits the dataset into k stratified folds (same function used in the decision tree). This ensures that each fold maintains the same proportion of class labels as in the original dataset. The output *folds* is a list where each element contains the indices of the samples assigned to a specific test fold, while the remaining samples form the corresponding training set. The function systematically explores different values for key hyperparameters of the random forest model:

1. *num_trees*: defines the number of decision trees in the forest and the values that were tested are [5].
2. *max_depth*: defines the maximum depth of each tree and the values that were tested are $[10, 15]$.
3. *min_samples_split*: the minimum number of samples required to split a node, with tested values $[2, 5]$.
4. *max_features*: corresponds to the fraction of features randomly selected at each split and the values that were tested are $[None, 3]$.
5. *impurity*: used for splitting {`gini`, `scaled_entropy`, `custom_impurity`}.

The function iterates through all possible combinations of these hyperparameters to test different random forest configurations.

For each combination of hyperparameters, the function performs k-fold cross-validation, where the dataset is divided into k folds. The model is trained on $k-1$ folds and tested on the remaining fold. This process is repeated k times, ensuring that each fold serves as the test set once.

During each iteration of the cross-validation process, the function first selects the training and test indices from the predefined folds. Using these indices, it extracts the corresponding subsets of `X_train`, `y_train`, `X_test`, and `y_test`, ensuring that the training and test sets are correctly partitioned for the current fold.

Next, a new instance of the `Random_Forest` model is created, initialized with the specific combination of hyperparameters being tested. The model is then trained using `X_train`, leveraging the provided `feature_types` to correctly handle different variable types.

Once training is complete, predictions are generated for both the training and test sets. The function then evaluates the model's performance by calculating both training and test key metrics, such as accuracy and error $(1 - accuracy)$.

Once all folds have been evaluated for a given hyperparameter combination, the mean accuracy and error rates across all folds are computed.

5.2 RESULTS

num trees	**max depth**	**min samples split**	**max features**	**impurity**	**mean train accuracy**	**mean test accuracy**	**mean train error**	**mean test error**
5	15	2	NaN	scaled_entropy	0.916399	0.878262	0.083601	0.121738
5	15	5	3.0	scaled_entropy	0.908163	0.875492	0.091837	0.124508

num trees	max depth	min samples split	max features	impurity	mean train accuracy	mean test accuracy	mean train error	mean test error
5	15	5	NaN	scaled_entropy	0.894014	0.867382	0.105986	0.132618
5	15	5	NaN	gini	0.899936	0.865591	0.100064	0.134409
5	15	2	NaN	custom_impurity	0.909675	0.861130	0.090325	0.138870
5	15	2	3.0	scaled_entropy	0.897841	0.858968	0.102159	0.141032
5	15	2	3.0	gini	0.890424	0.856501	0.109576	0.143499
5	10	2	3.0	custom_impurity	0.854811	0.848359	0.145189	0.151641
5	15	2	NaN	gini	0.875295	0.847515	0.124705	0.152485
5	15	5	3.0	gini	0.873487	0.845251	0.126513	0.154749
5	15	5	NaN	custom_impurity	0.874197	0.844303	0.125803	0.155697
5	10	2	NaN	scaled_entropy	0.851356	0.843360	0.148644	0.156640
5	10	2	3.0	gini	0.846626	0.842005	0.153374	0.157995
5	15	5	3.0	custom_impurity	0.875769	0.841329	0.124231	0.158671
5	10	5	3.0	scaled_entropy	0.849059	0.841059	0.150941	0.158941
5	10	2	3.0	scaled_entropy	0.850351	0.838188	0.149649	0.161812
5	10	5	NaN	gini	0.847260	0.837242	0.152740	0.162758
5	15	2	3.0	custom_impurity	0.857118	0.835044	0.142882	0.164956
5	10	2	NaN	gini	0.831599	0.822544	0.168401	0.177456
5	10	5	3.0	custom_impurity	0.826911	0.819030	0.173089	0.180970
5	10	5	NaN	scaled_entropy	0.827815	0.816427	0.172185	0.183573
5	10	5	NaN	custom_impurity	0.818886	0.807608	0.181114	0.192392
5	10	5	3.0	gini	0.813446	0.804974	0.186554	0.195026
5	10	2	NaN	custom_impurity	0.815398	0.804701	0.184602	0.195299

Table 3: Hyperparameter Tuning for Random Forest with Cross-Validation

The best-performing model has the following hyperparameters:

- *Training accuracy:* 0.9164
- *Test accuracy:* 0.8783
- *Impurity function:* scaled_entropy
- *min_samples_split:* 2
- *max_features:* NaN (automatic feature selection)
- *max_depth*: 15
- *num_trees*: 5

The impurity function significantly impacts performance: among the three tested (scaled_entropy, gini, and custom_impurity), *scaled_entropy* consistently produces the highest test accuracy and the best balance between training and test performance.

The choice of *min_samples_split* is also crucial. Models with *min_samples_split* = 2 achieve the highest accuracy. In contrast, models with *min_samples_split* = 5 maintain high accuracy with improved generalization.

The *max_features* parameter, which controls the number of features considered at each split, also plays a significant role. Limiting the number of features (*max_features* = 3.0) improves test accuracy and reduces overfitting compared to using all available features (*max_features* = NaN).