# Statistical method for machine learning

Project 2: Tree predictors for binary classification

Alessandro Di Liberti

August 2024

## Contents

# 1 Introduction

In the field of machine learning, decision trees stand out as a fundamental tool for classification tasks. They offer a combination of interpretability, flexibility, and efficiency, making them an invaluable asset in the machine learning practitioner's toolkit. This project focuses on the construction and evaluation of binary decision tree predictors, with an emphasis on implementing these models from scratch, using single-feature binary tests as decision criteria at each internal node.

The objective of this project is to develop a system for classifying the edibility of mushrooms based on a set of predefined features. The decision tree model is particularly well suited for binary classification due to its simple structure, where each internal node represents a decision rule based on a single feature, and each leaf node corresponds to a class label. The hierarchical nature of decision trees allows them to capture complex patterns in the data by recursively partitioning the feature space. This project aims to explore and optimize various aspects of decision tree construction, including the selection of partitioning criteria and the implementation of effective stopping conditions.

# 2 Dataset

The dataset used for this project is 'Mushrooms' from UC Irvine Machine Learning Repository. Some general informations about this data are:

- **Number of attributes**: 20, divided into
  - **Categorical**: 17
  - **Numerical**: 3
- **Number of examples**: 61069
- **Classes**: two main classes, *edible* and *poisounous*

Since the classifier used will be a decision tree, there is no need to apply normalization to the data. However, some caution was necessary for the purpose of good results.

There are two main problems that need to be solved: **the continuous data** and **the missing values**.

The continuous data were handled by discretizing the values in thresholds, enable binary decision-making at each node. This discretization is achieved by dividing the space into bins of the same size, and then using the thresholds dividing the bins in the splitting node. In this implementation **50 bins** where used. This value could be implemented as an adjustable hyperparameter during training, but in order to limit training time, it was chosen to keep it fixed. With this solution, the continuous data in the training set are treated as categorical, and once the predictor is trained, a comparison operation ($>$ or $\leq$) is used to make the decision.

The second problem concerns missing values: there are some examples that have NaNs on attributes. This problem is more complex to handle since it involves both

training and actual data that have never been seen (since they may have missing values). During training, when searching for the best feature and threshold, the examples with missing values on the feature under analysis are ignored; when the calculations are done and the best split is found, the previously ignored examples all go to the node which already has the biggest number of instances. Regarding the use of the predictor with data never seen, in case of missing value for the split, the example will result as false.

# 3 Building the tree

When constructing a binary decision tree classifier, the primary goal is to create a model that systematically partitions the data based on certain conditions. At each node within the tree, a decision criterion is posed in the form of a binary question or condition. The dataset is then partitioned based on the outcome of that condition. If the condition is true, the corresponding data examples are directed to the right subtree. Conversely, if the condition is false, the examples are directed to the left subtree. This recursive splitting process continues, with each node refining the data set into more homogeneous subsets, until the data is sufficiently partitioned.

The construction of the tree follows a recursive approach that ends when a stopping criteria occurs. The idea behind it is that when a node is created it becomes a leaf if a stopping condition occurs, and the label attached to it is the most common label of the examples routed to it; in the opposite case, the node becomes an internal one, and generates a left and a right node. The examples that verify the split function on that node will fall to the right, the others to the left. The operation is repeated for each node until all recursions end.

## 3.1 Stopping Criteria

The stopping criteria are the hyperparameters of the model that regulate the growing of the tree. This helps prevent the tree from growing unnecessarily complex and reduces the risk of overfitting. Three stopping criteria were implemented in this project:

- **max_depth**: this criteria exploits the concept of tree depth. When a new node is created, its depth value is checked: if this is $\geq$ than the hyperparameter entered during training, automatically the node becomes a leaf that has as its associated label the most common label among the examples routed to the node.

- **min_samples_split**: it sets the minimum number of data samples required at a node for it to be considered for splitting, if the number of examples is less than the threshold, the node is set as a leaf.

- **min_impurity_decrease**: is a stopping criterion that controls tree growth by setting a threshold for the reduction in impurity needed to make a split. If the decrease in impurity from a potential split is less than this value, the split is not made.

## 3.2 Implementation

Two classes are used to construct the classifier: **Decision_tree** and **Node**. Decision_tree represents the classifier, while Node represents the individual nodes that make up the classifier

### 3.2.1 Node

An object of type Node represents a leaf or an internal node, there is a method that offers the ability to know what state it is in. If a node is leaf then it will have an associated "value" representing the label routed to the node. In the case of internal node, the node will have a reference to the feature and threshold on which to split, and also a reference to the left and right children.

### 3.2.2 Decision Tree

The main part of the project is based on the Decision_tree class. This class only exposes the methods fit() and predict(). The former is needed to train the tree, the latter to use it with the test set and real data. The use of these two functions will be explained later.

One functionality of this class is to compute the training error. Since this is based on zero-one loss, when a leaf is created it counts how many examples are not of the same class as the most common label (the one the leaf is bound to), when the tree is finished building, this number it is divided by the number of training examples. So the training error is computed as follows:

$$\ell_S(h) = \frac{1}{m}\sum_{t=1}^{m} \ell(y_t, h(x_t)) \qquad \text{with } \ell = \begin{cases} 0 & \text{if } y_t = h(x_t) \\ 1 & \text{otherwise} \end{cases} \tag{1}$$

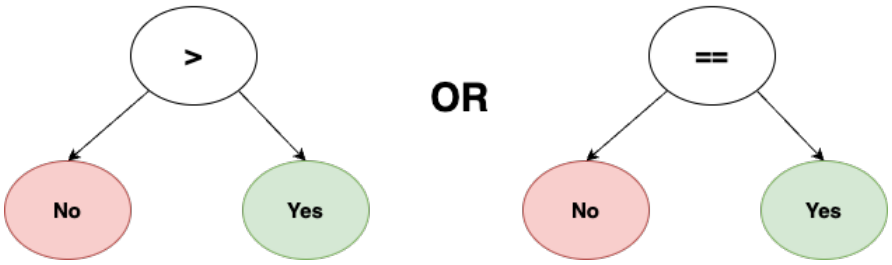
### 3.2.3 Training

The algorithm to build the tree is based on the following functions:

- ***grow_tree()***: recursive function that is called at the begginning of the training. It returns a Node as a leaf if a stopping criteria occurs, or it calls itself to generate left and right children, and returns an internal node.
- ***best_split()***: this function is used to find the best best feature, the best threshold and the information gain obtained.
- ***information_gain()***: this function is used to compute the information gain given a feature and a threshold. The gain is obtained by first computing the purity of the parent (that is the purity of the labels on the node before the split), then the split is applied and the purity of each childer is computed. The last step is to subtract the weighted purity of the children from the parent.

$$\textbf{Information\_Gain} = E(parent) - p_{left}E(left) - p_{right}E(right) \tag{2}$$

- ***split()***: returns the examples that do not verify the split condition (those $\leq$ than the threshold in the case of continuous values, or $\neq$ in the case of categorical values) and the examples that verify the condition; respectively, the examples that will go into the left child and those that will go into the right child.




- ***criterion()***: this function is used to compute the purity of a given set of labels. This function compute purity based on the hyperparameter chosen before training. There are three measure of purity to choose:
  - entropy $= -\frac{p}{2}\log_2(p) - \frac{1-p}{2}\log_2(1-p)$
  - gini $= 2p(1-p)$
  - standard deviation $= \sqrt{p*(1-p)}$

  Where $p$ is the probability of the element of positive class (edible).

The general idea behind the algorithm is the following:
*Given a training set S, with X the examples and y the labels*

1. Call grow_tree(X,y)
2. If one of the following stopping criteria is verified, return Node(most_common_label).
   - depth $\geq$ max_depth
   - samples $<$ min_samples_split
   - all the samples are of the same label
3. $\forall$ features and $\forall$ thresholds
4. Split $X$ on feature with threshold, and get $y_{left}$ and $y_{right}$
5. Compute information gain with 2 with the criterion set by the hyperparameter
6. Find best_feature and best_threshold as the ones having the highest information gain
7. Split on $X$ with best_feature and best_threshold to get: $X_{left}$, $X_{right}$, $y_{left}$ and $y_{right}$
8. Compute:
   - left_child = grow_tree($X_{left}$,$y_{left}$)
   - right_child = grow_tree($X_{right}$,$y_{right}$)
9. Return Node(left_child, right_child, best_feature, best_threshold)

### 3.2.4 Classification

When training is complete, the tree is ready to be used for predicting. To do this, it exposes the predict() method, which takes a list of examples and returns the list of corresponding predictions for each element.

The implementation of this function is done by the recursive traverse_tree() method. It takes as parameters a single example to analyze and a node. The first call passes the root of the tree as the node. For each node, it checks if the example satisfies the split condition; if so, it calls traverse_tree() again, but passes the right child as a parameter; if the split condition is not satisfied, it passes the left child. The recursion ends when the node reached by the example is a leaf, and the value associated with it is then returned.

As mentioned before, data with one or more features with NaN values require special treatment; in fact, it is first checked if the example to be predicted has a missing value on the feature associated with the node, if it does, this will result as a split condition whose outcome is false, thus calling traverse_tree() with the left child node as parameter.

# 4 Hyperparameter tuning

As mentioned earlier, the model uses 4 different hyperparameters to construct the tree: max_depth, min_samples_split, min_impurity_decrease and criterion. Proper selection of these hyperparameters significantly increases the quality of the model.

If you had a single hyperparameter, choosing its value would be much easier; you would scroll linearly among the possible values and choose the one with a better evaluation metric. In this case, when you have multiple hyperparameters to set, you have to consider combinations of them. In the early stages of implementing the tree, it quickly became apparent that there are hyperparameters that are more relevant than others (more details in the results section), but for fine-tuning the model, one must consider the combination of all four parameters.

To tackle this problem a robust hypeparameter tuning methodology was used: **Grid Search**. Grid Search systematically evaluates a specified set of hyperparameter values by exhaustively training and validating the model on all possible combinations. For each combination, cross-validation is used to evaluate the model's performance, ensuring that the selected hyperparameters generalize well to unseen data. The result is a set of hyperparameters that yield the best model performance, improving predictive accuracy and reducing the risk of overfitting. While computationally intensive, Grid Search is a powerful technique for finding the optimal settings for a decision tree classifier to ensure its best performance.

Considering that the model performed very well even without complex tuning, the reasons for choosing this technique are:

- **Easy implementation**
- **Comprehensive exploration**

- **Cross validation integration**

### 4.1 Implementation

The implementation is very simple: a list of values is used for each parameter; for training time concern the number of values for each hyperparameter is 2 or 3. This is because for each possible combination (in this implementation there are 54 possible combinations) a cross-validation has to be done, which increases the training time considerably. In any case, it is easily possible to increase the number of values by adding them to the list.

To be more precise here is the list of the values for each hyperparameter:

- **max_depth**: [25,27,30] *it was chosen to use higher value because from preliminary test it performs much better in this way.*
- **min_samples_split**: [1,2,5]
- **criterion**: [gini, entropy, std_dev]
- **min_impurity_decrease**: [0.001, 0.0005]

For each combination a model is created, and then cross validation is applied.

## 5 Validation

As mentioned before, finding the best combination of hyperparameters requires validating the model and evaluating its performance. For each possible parameter combination, a tree (yet to be trained) is created and then a train and test is run to evaluate its performance. To have higher quality of evaluation metrics, cross-validation is used. This reduces any imbalances in the train and test set, which could lead to more or less "efficacy" trains, making the values of the evaluation metrics more realistic.

In order to compute the metrics, **Strtified K-Fold Cross Validation** is used. Stratified K-Fold Cross Validation is a robust and widely used technique in machine learning and statistical modeling, particularly when dealing with imbalanced datasets. It extends the basic concept of K-Fold Cross Validation by ensuring that each fold is representative of the entire dataset's class distribution.

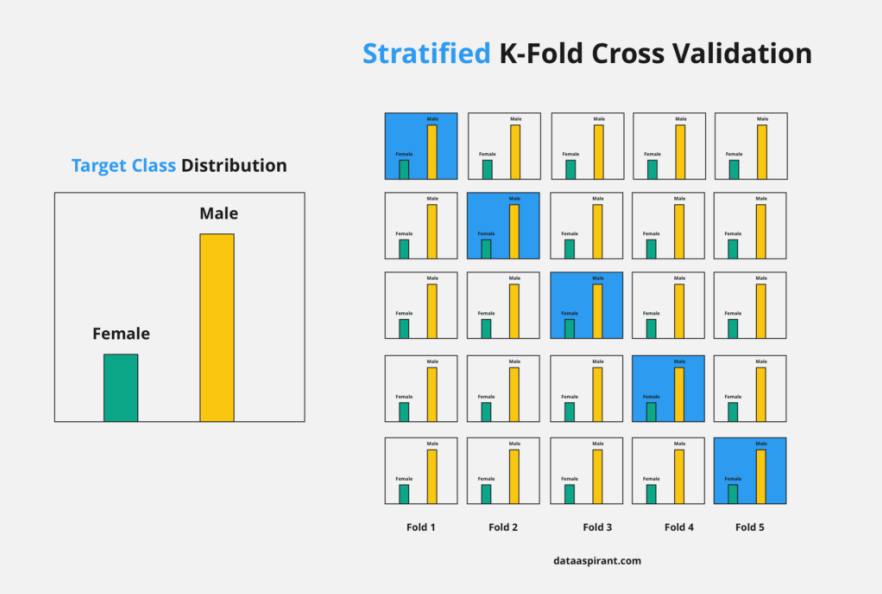



Figure 1: **Source**: https://dataaspirant.com/cross-validation/

The idea behind this method is similar to K-Fold: dividing the datase in $k$ folds, train the model with $k-1$ folds and testing the model on the remaining fold. Repeat this process for each fold (every time the "testing fold" is different) and finally averaging the metrics from all the tests. The metric obtained will be closer to the actual metric than the one calculated by a simple train-test split. The main difference with K-Fold is the fact that the distribution of the label from the original dataset is preserved during the creations of the fold. This is done in order to handle an unbalanced dataset.

In this case the dataset is made by 33888 **poisounous** mushrooms and 27181 **edible** mushrooms.

## 5.1 Metrics

Following here there is a list of all four metrics used to evaluate the model. It's important to remember that an element is considered to be of a positive class if it is labeled as 'e', i.e. it's edible.

- **Accuracy**: is the proportion of all classifications that were correct, whether positive or negative. It is defined as:

$$\textbf{Accuracy} = \frac{\text{correct classification}}{\text{total classification}} \tag{3}$$

- **Precision**: is the proportion of all the model's positive classifications that are actually positive. It is defined as:

$$\textbf{Precision} = \frac{\text{correct positive classification}}{\text{everything classified as positive}} \tag{4}$$

- **Recall**: is the proportion of all actual positives that were classified correctly as positives. It is defined as:

$$\textbf{Recall} = \frac{\text{correct positive classification}}{\text{actual positive}} \tag{5}$$

- **F1**: is the harmonic mean of precision and recall, providing a single measure that balances both the model's ability to correctly identify positive instances (precision) and its ability to capture all relevant instances (recall). It is defined as:

$$\mathbf{F1} = 2 * \frac{\text{Precision} * \text{Recall}}{\text{Precision} + \text{Recall}} \tag{6}$$

To evaluate the best combination of hyperparameter the accuracy metrics is used, but the recall, precision and f1 score are also computed and saved for later (more details on Score history 5.3).

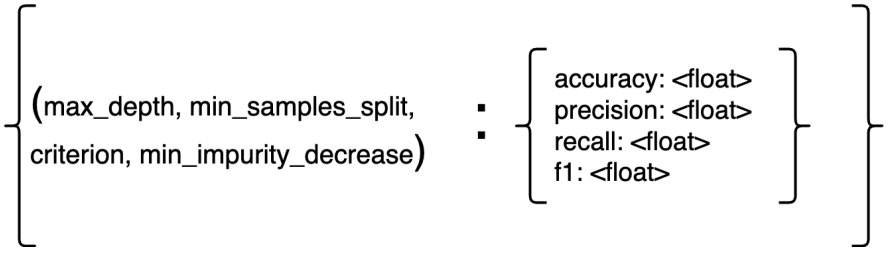
## 5.2 Implementation

The implementation of model evaluation is closely related to cross-validation and it is done in two steps. The first operation is a train-test split on the original data set with shuffle and test_size of 15%. The resulting test set is used at the end for a final evaluation. This first train set is what is used to cross-validate the model; using Strafied K-Fold cross-validation, with $k = 5$, the model is trained with the $k - 1 = 4$ folds and accuracy, precision, recall and f1 are calculated. At the end of the k trainings/tests, the average is taken for each metric and the accuracy and hyperparameter combinations are tracked. If the accuracy associated with a combination is better than best_accuracy, then best_accuracy and best_combo are updated. The process ends when all combinations have been tested.

The last step is to train the model with the entire train set obtained from the first split and evaluate it with the test_set set aside at the beginning.

## 5.3 Score history

As mentioned before, when calculating the metrics, in addition to accuracy; precision, recall and f1 are also calculated. Nevertheless, the last three are not used to find the best predictor. However, these metrics are not ignored, they are actually stored in a structure that keeps track of all scores for every possible combination of hyperparameters. The structure looks like this:




*The implementation of this structure is saved in the .pkl file named "metrics.pkl", and a short script to analyze this file is metrics_analysis.py*

There are many reasons why it was chosen to keep track of all the metrics obtained during cross validation, one of them is being to be able to visualize the training trend

and observe which hyperparameters are more relevant than others. The main reason, however, lies in the way the best predictor is selected: the metric used is accuracy, which, although it is the most intuitive and generally the one most used may not be the best in this context.

We must keep in mind that the model is used to predict the edibility of mushrooms, so it is much more important that the predictor does not classify a mushroom as edible when it is not. This is precisely why precision and recall come into play. In a model predicting the edibility of mushrooms, precision is more important than recall.

Here's why:

- **Precision** measures the proportion of mushrooms classified as edible that are actually edible. High precision is crucial here because it reduces the risk of classifying a poisonous mushroom as edible (false positive). In this context, it's crucial to minimize the risk of misclassifying an edible mushroom as poisonous, as this could lead to serious harm.
- **Recall** measures how well the model identifies all edible mushrooms. So with an higher precision there are more mushrooms predicted as edible, but within this there may be ones that are not actually edible.

Maximizing precision helps to avoid potentially dangerous mistakes, which is critical when human health is at stake. So it is batter to waste some mushrooms rather than eat a poisounous one.

In the end, having an history of all the metrics computed for each combination it is a faster way to find another predictor which best suits the purpose, without having to re-do the whole training process.

# 6 Results

After the whole process of training and evaluation, the predictor with the highest accuracy was found, and it consists of the following hyperparameters

- **max_depth**: 27
- **min_samples_split**: 1
- **criterion**: entropy
- **min_impurity_decrease**: 0.0005

This predictor obtained the following scores in the metrics seen above:

| Accuracy | Precision | Recall | F1 |
|---|---|---|---|
| 0.9995 | 0.99951 | 0.99951 | 0.99951 |

This excact predictor also scores the highest precision and f1 score (by looking at the score history).

The predictor having the highest recall differs from the one above just by the max_depth, which is 25.

# 7 Conclusion

The results obtained are very satisfactory and show that a model based on a tree predictor is very effective for binary classification. The reasons for this are that the model was trained on a lot of data and that the available features are significant for this type of classification.

The points to be made about situations that arose during development are:

- **most relevant hyperparamete**: this is definitely max_depth, in fact, by varying it the performance of the model varies significantly.
- **overfitting**: another observation is that the model suffers from a slight overfitting for high values of max_depth, this is demonstrated by the fact that although max_depth = 30 was present in the grid search parameters, the best model is the one with max_depth = 27. It is observed that by increasing this value, the training error tends to zero (in some training it reached zero), but the accuracy decreases.
- **choice of criterion**: although entropy is more effective, the gap with gini is practically zero, making both methods valid. The case is different for the standard deviation, which shows a slightly lower performance.

# 8 Random forest

Random Forest is a popular machine learning algorithm that constructs multiple decision trees in different ways, to improve accuracy and reduce overfitting. The random forest introduces a random component using a different dataset for each tree, and a different attribute pool for each tree node. This voluntary introduction of noise, coupled with ensemble, results in some of the highest performance learning models.

## 8.1 Building the forest

The goal of random forest is to have many trees that have noise inside them and to use the concept of majority voting to express the final decision. An element will be classified by each tree in the forest, and the class predicted by the forest will be the one most present in the tree predictions.

The noise component is inserted through two techniques:

- **Bagging**: This technique involves training each tree with a different dataset. The feature is that each dataset is generated from the original one by performing $m$ extractions with replacement (where $m$ is the size of the dataset). In this way, each tree will have $\sim 1 - \frac{1}{e}$ unique elements, and the rest will be repetitions.
- **Attribute sampling**: This technique works at the feature level during the split. In the case of a classic training of a decision tree, when choosing the attribute and the threshold on which to split, all possible features are used. With attribute sampling the pool that can be used to search for the best split is reduced, using $\sqrt{d}$ features (where $d$ represents the total number of features). This reduction of

features is done at the level of each node, and each node will be able to choose the best split from a different set of attributes.

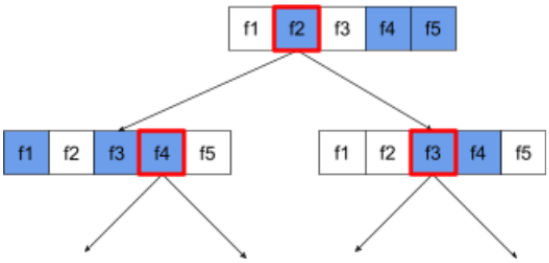



Figure 2: **Source**: https://developers.google.com/machine-learning/decision-forests/random-forests

The process for building the tree is the same as the one described before in section 3 "Building the tree". The main difference is that the trees we are going to train are slightly overfitting, this behavior is then corrected by the sctructure of the ensemble.

### 8.1.1 Evaluation

To evaluate the model, in addition to a classic train test split of the original dataset, a specific evaluation score for the random forest was used: OOB score.

The **Out-of-Bag (OOB)** evaluation is a method used in Random Forests to estimate the model's performance without needing a separate validation set. Since each tree in the Random Forest is trained on a bootstrap sample of the data, roughly one-third of the data is not used during the training of any given tree. This "left-out" data for each tree is referred to as Out-of-Bag data. Each data is then predicted by the tree/trees in which is not part of the training set (so it is an OOB for that/those tree/trees), majority voting is applied and then the final prediction is made. The score is the accuracy of the predicted label compared to the true label.

Once the training is over, accuracy, precision, recall and f1 are computed on the test set.

### 8.1.2 Implementation

To implement this model, Decision_tree and Random_Forest classes were used.

In Decision_tree there is now a flag (is_forest), which if activated selects $\sim \sqrt{d}$ random features, on which the best split is then selected. Obviously the trees generated inside the forest will have the flag active, while the implementation of a single Decision_tree will have this parameter set to False

The main class is Random_Forest, the hyperparameter is *n_trees* that represents the number of tree inside the forest. In this predictor this parameter it has been set to 29, this is because after several tests this value is the one that has shown the best results with respect to training times.

The implementation of the forest is based on these following methods:

- ***build_forest()***: this function generates *n_trees* trees and put them in a list. It is also responsible to identify the OOB data for each tree.
- ***bootstrap_sample()***: it generates *n_trees* bootstrap samples and return them as a list.
- ***oob_evaluation()***: this function is called after the train is done and performs the prediction for the OOB data, and then computes the OOB_score

Each tree in the forest it has been implemented in a way that overfits. We previously saw in section 7 "Conclusion" that with a higher max_depth the tree tends to overfitting. So each tree is trained with max_depth = 30.

### 8.1.3 Reproducibility

To allow reproducibility of the experiment an editable seed is implemented. This seed is used in the Random_Forest class and it is used to generate random seeds for each tree. This is done to avoid that each tree exctract the same feature for each node, thus having *n_trees* identical trees.

## 8.2 Results and conclusion

The results of this model are as expected. With 29 slightly overfitting trees, we get better results than training a single tree. The difference in performance between the two models is minimal because both achieved a very high degree in every metric. The characteristic of Random_forest is that it was able to obtain a perfect score for precision, which as we said previously, is the most relevant metric in this context.

The only parameter worth of tuning was the number of feature to use during the attribute sampling. Since we have 20 feature, the model was trained first with four feature, and then with five. As we can see in the following tables, it performs better with five feature.

With 29 trees and 4 feature for subsampling the results are:

- **OOB score**: 0.9998

| Accuracy | Precision | Recall | F1 |
|---|---|---|---|
| 0.9998 | 1.0 | 0.9994 | 0.9997 |

With 29 trees and 5 feature for subsampling the results are:

- **OOB score**: 0.9999

| Accuracy | Precision | Recall | F1 |
|---|---|---|---|
| 0.9999 | 1.0 | 0.9998 | 0.9999 |