# Machine Learning Project : classification of edible mushrooms

**Abstract**

This document is the report of a project taking place during the course of Statistical Methods for Machine Learning conducted by prof. Cesa Bianchi in the Università degli Studi di Milano Statale. This lecture aims to develop some method of machine learning using statisticals tools. You can find the repository of the project at this link : `https://github.com/theot-student/MachineLearning_Project`.

## 1 Introduction

The goal of this project is to classify mushrooms in two categories: if they are edible or if they are poisonous. The solution to this problem that is explored in this project is to use machine learning methods. In particular, we will explore the solution of using decision tree (and possibly random forest) to predict if a mushroom is edible or not.

## 2 Dataset

This project will use the "Secondary Mushrooms" Dataset : Wagner,Dennis, Heider,D., and Hattab,Georges. (2023). Secondary Mushroom. UCI Machine Learning Repository. This dataset can be find here. This dataset contains 61068 exemples of mushrooms lebelled if they are edible or not. Every mushroom is represented by 20 features either categorical or continuous. Some features have missing values, but it's not a real problem in decision tree, because we can simply chose a hypothese such that None values are always considered as False for the decisions in our nodes in our tree (see later for explanation of decision tree). You can find a description of the 20 features in the website of the dataset.

To import the data in the jupyter notebook, we use a the method proposed in the same website of the data which give a panda dataframe of the data.

# 3 Recall of decision tree

In this part, I will recall you what is a decision tree in theory, and recall the different parameters used to change the learning.

## 3.1 What is a decision Tree

### 3.1.1 Idea of decision tree

A decision tree, is a tree where each inner node correspond to a criterion and each leaf correspond to a decision. This can be use to find class of element, or to make decision corresponding to a situation. In fact, to do so, we start from the root and use criterion to descend the tree. When we reach a leaf, we have then the class or the decision corresponding on the element or the situation. For exemple, considering animal, we can retrive animal with the tree in *Figure 1.*

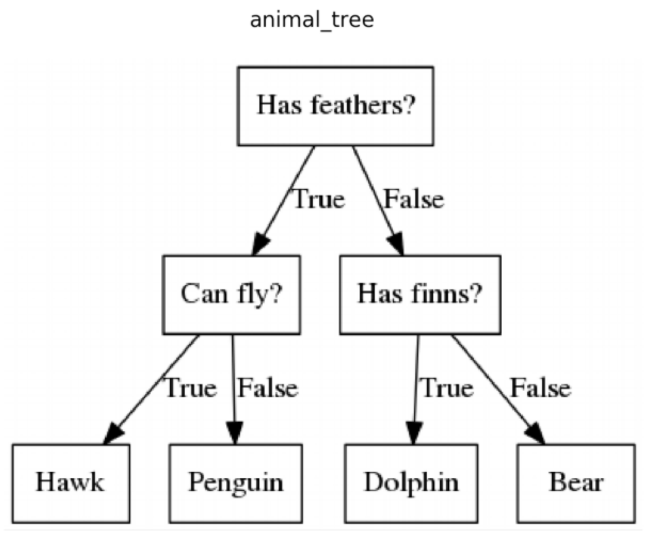

Figure 1: simple decision tree to find the race of an animal.

## 3.2 Decision Tree in Machine Learning

In Machine Learning, a decision tree is a model which given a training set S of labelled data $\{(x_1, y_1), ..., (x_n, y_n)\}$ can construct a decision tree to retrieve the label y of an exemple x.

To create the tree, the idea is the following :

1. we create a root, which label is the majority of the label
2. we chose a leaf (usually by greedy algorithm) considering a spliting criterion and a set of feature criterion
3. we split this leaf in two new leaf and chose the label with majority in these leaves
4. we repeat the 2. and 3. step until a stopping criterion

Then we need to discuss what are all the criteria used in the algorithm :

### 3.2.1 Feature criterion

these are the criteria that can be used on the feature of the data. For exemple if the feature is categorical, a criterion could be to check if this feature of a data point correspond to a category. This correspond to our criteria in the node in the first figure. We will discuss of which criterion to use in the implementation part of the project.

### 3.2.2 Splitting criterion

this criterion is used to calculate wheter or not a split will drop the training error. Let's recall theoriticaly the hypotheses for this. Considering a leaf that we split with a specific feature criterion. For a leaf l, let $N_l$ the number of exemple rooted to this node and $N_l^+$ the number of right exemple (i.e which label is the same label of the leaf) rooted to this node. Then, some theoritical analysis give us this formula :

$$l_S(h_T) = \frac{1}{m}\sum_l \psi(\frac{N_l^+}{N_l})N_l$$

with $l_S(h_T)$ the training error of $h_T$ a tree predictor, m the number of leaves in $h_T$ and $\psi$ the splitting criterion. Then, as we use greedy algorithm, we want to maximize the dropout of the training error. Then considering a leaf l and a feature criterion c, we denote by $l_R$ and $l_L$ the right and left children creating by splitting l with criterion c. Then, to dropout the training error we want to find the solution to :

$$\max_{l,c} \psi(\frac{N_l^+}{N_l})N_l - \psi(\frac{N_{l_R}^+}{N_{l_R}})N_{l_R} - \psi(\frac{N_{l_L}^+}{N_{l_L}})N_{l_L}$$

in all leaves l of our tree and in all criterion c in our features criterion. To ensure that a splitting is reducing the training error we need that the splitting criterion is concave. Here are some exemple of splitting criteria that can be used :

- **Gini Index** : $\psi(p) = 2p(1-p)$
- **Scaled Entropy** : $\psi(p) = -\frac{p}{2}\lg_2(p) - \frac{1-p}{2}\lg_2(1-p)$

### 3.2.3 Stoping criterion

This criterion is used to stop the construction of the tree. In fact, we don't want a full tree with all possible splits because it would be to long for computation and it would more likely to overfit the training set. Then we impose a stop condition of growing the tree. Here are some exemples :

- **Max depth** : we can limit the maximum depth of the tree in order to keep it quite simple.
- **Homogeneity** : another choice is to stop growing the tree when there are not enough exemples in the new leaves created.

**Remark :** The idea of this algorithm is quite easy but, since it is a greedy algorithm, it may take some time to be computed. We will see that in this project.

## 4 Implementation

In this part we will see how the decision tree are implemented in this project. First, I want to recall that this project is coded in python3 on a jupyter notebook available here. We will need the classical libraries such as numpy and pandas.

### 4.1 Implementation of Node

First step is to implement a class for a Node of the tree. This is done by creating a classical recurcive class in object oriented programming. In fact, each node has a right and left children that are also nodes. Here is the class of the node in the jupyter notebook:

```
class Node:

  def __init__(...):
    self.rightChildren = rightChildren
    self.leftChildren = leftChildren
    self.isALeaf = isALeaf
    self.label = label
    self.criterion = criterion
    self.criterionAlreadyUsed = criterionAlreadyUsed
    self.errorVariationDict = errorVariationDict
```

where :

- *rightChildren* and *leftChildren* are the recurcive part and denote the childrens nodes
- *isALeaf* is a boolean checking if the node is a leaf
- *label* is the label of the leaf

- *criterion* is the criterion used to split the node.
- *criterionAlreadyUsed* are the criteria already used before in the tree
- *errorVariationDict* is a dictionnary keeping the variation error (i.e $\psi(\frac{N_l^+}{N_l})N_l - \psi(\frac{N_{l_R}^+}{N_{l_R}})N_{l_R} - \psi(\frac{N_{l_L}^+}{N_{l_L}})N_{l_L}$) for every possible splitting of this leaf considering all criteria not already used before

When the tree is finished, it is easy to predict a label because we just have to descent among our tree by using the criterion of each node to chose which way to take. **Important remark :** in this project, every criterion node conduct to the right children when the result of the criterion is true. What I mean is that when an exemple is checked with a criterion , if the check is true, the exemple is sent to the right children of the node, and if it is false, it is sent to the left children.

## 4.2 Implementation of tree classifier

In this part, we will see how the tree classifier class is implemented and the most important method in this class.

### 4.2.1 class variables

First let see which are the variables of a tree classifier :

```
class BinaryTreePredictor:

  def __init__(self, decisionCriterionSet, splittingCriterion, stoppingCriterion):
    self.decisionCriterionSet = decisionCriterionSet
    self.splittingCriterion = splittingCriterion
    self.stoppingCriterion = stoppingCriterion
    self.labels = None
    self.dictLeaves = None
    self.nbExemples = None
    self.tree = None
```

where :

- *decisionCriterionList* is the set of criterion possible over our features. Need to define what is a criterion, probably a function
- *splittingCriterion* is a function to compute the training error and chose the best splitting reducing training error (i.e $\psi$ function)
- *stoppingCriterion* is a function that stop the grown of the tree (for exemple when the depth max is reached)
- *labels* is an array containing all the possible labels

- *dictLeaves* is a dictionnary containing all the leaves as keys and the set of exemples rooted to this leaf as value
- *nbExemples* is an integer of the number of exemples
- *tree* is the root of the tree classifier created by the algorithm

#### 4.2.2 training the tree classifier

Now, we can focus on how a tree is really created. In first step, we need to create a new *BinaryTreePredictor* object with a *decisionCriterionList,* a *splittingCriterion* and a *stoppingCriterion.* After that, we can call the method train of the *BinaryTreePredictor* passing the training features and training targets in a numpy array and the method we want to use (**explained in a later part**):

```
def train(self, training_features, training_target, method = 'greedy'):
```

this method will create and train the decision tree for the given training set. Let's have a quick look to how works this method :

First, we need to create all the missing variables of our class :

- the labels are creating by counting the uniques label in target set
- the number of exemples is easily computed by getting the length of the training set
- the tree is created as a leaf whose label is the majority label over the training target
- the dictionary of leaves add the tree as the only leaf with all training pointsnode

Then, we need to call the method *calculVariationErrorDict* which, given a leaf, compute the dictionnary of the variation error considering the *decisionCriterionList.* We will see how this method works after. Now that everything is initialized, we need to run the loop to grow the tree :

```
while not(self.stoppingCriterion(self.tree, self.dictLeaves))
                and (variationError > 0):
        if method == 'greedy':
           variationError = self.growTree()
        if method == 'greedyAllLeaves':
           variationError = self.growAllLeaves()
        trainingError = self.computeTrainingError()
```

the methods *growTree* and *growAllLeaves* are used to train the tree by splitting leaves as we have seen before. Then we calculate the training error as an indicator of advancement of the training. This loop stop either because of the stop criterion or because the *variationError* for the split is lower than 0. Recall that the variation error is the value $\psi(\frac{N_l^+}{N_l})N_l - \psi(\frac{N_{l_R}^+}{N_{l_R}})N_{l_R} - \psi(\frac{N_{l_L}^+}{N_{l_L}})N_{l_L}$ for the chosen split. Therefore, if the variation error is below 0 then the training error can not be reduce anymore (but it should not happen since we never construct the whole tree).

***calculVariationErrorDict*** Now let's check how the *calculVariationErrorDict works :*

```
def calculVariationErrorDict(self,leaf, points = None):
    errorVariationDict = dict()
    for criterion in self.decisionCriterionSet:
      if not(criterion in leaf.criterionAlreadyUsed):
        errorVariationDict[criterion] = self.testSplit(leaf, criterion, points)
      else:
        errorVariationDict[criterion] = -100
    return errorVariationDict
```

This function is just a loop over the decision criteria in order to get the variation error created using this criterion to split the given leaf. Since we only need variation error from criterion not already used, we just put absurd value in criterion already used (negative because we are interested in the maximum of variation error). The *testSplit* method is "simulating" the splitting of the leaf with the given criterion and compute the variation error of this split.

Then this function returns a dictionnary with all criteria and variation error for these criteria.

***growTree* or *growAllLeaves*** Let's talk about the difference of the two method to grow the tree.

The first one, *growtree*, is the usual greedy method for splitting leaves. It will check the maximum variation error over all the leaves and the criteria and will split considering this maximum. Notice that this maximum is not really hard to find because all the variation errors of leaves are stored in their *errorVariationDict* variable.

The second one, *growAllLeaves* represents the almost the same algorithm, but instead of splitting the leaves one by one, we split all the leaves of the same depth. Then the depth is increased by 1 at each step. This method was usefull in term of time of computation in the beginning of the project because the *errorVariationDict* did not exist. But now, it is not as usefull as before. However it is still a good function to avoid some problem of computation time but then we take the risk of overfitting by splitting too many leaf.

In both of the algorithm, we need to perform the real split of a leaf.

***splitLeaf*** This function is use to split a leaf into 2 new leaves considering a given criterion :

```
def splitLeaf(self, leaf, criterion):
```

The idea is just to get all exemples rooted to the old leaf (stored in *dictLeaves*) and to check the criterion on these exemples. Then we create 2 new nodes getting the new labels by the majority in these leaves. We actualize the exemples in *dictLeaves.* Remember that the true checked exemples are always rooted to the right leaf. Then it remains the huge part which is to compute the *errorVariationDict* for the 2 new leaves.

And that's it. Now we have a functionnal algorithm to grow tree classifier.

## 5 Experiments

In this part we will discuss about some experiments conducted with the tree classifier on our data set of edible/poisonuous mushrooms. We will also talk about tuning the hyperparameters.

The experiments are conduct on google colab using my computer which has a processor Intel® Core™ i5-7200U CPU @ 2.50GHz × 4 and 12,0 Gio of RAM. I used exactly the same jupyter notebook as present joined to this report or in the github depository.

**Remark** The computation of the learning of a treeclassifier is quite long (at least on my computer), so the experiments conducted here are limited in order to keep a reasonable computation time. Feel free to try others parameters if you have time and a good computer.

### 5.1 Recall of the hyperparameters

First let's recall what are the hyperparameters in the tree classifier algorithm (see section 3.2 for more explanation):

- **Splitting criterion**
- **Stopping criterion**
- **Set of decision criteria**

One can argue that the set of decision criteria is not a real hyper parameters, since by doing a greedy approach we would need all the possible criteria on the features. There are 2 reasons why it's not possible in reality :

1. First, some features are continuous, so we need at least to separate in a finite number of case these features. Moreover a too precise criterion on continuous feature can conduct to a large overfitting.
2. Secondly, if we take all the possible criteria for categorical features, the size of the set will become too large. Recall that we need to check all the criterion over all the leaves. So we can't (at least on my computer) deal with too many criteria in a reasonnable time of computation

Let's go over the choices we have for our hyperparameters :

**Splitting criterion** : as we have seen before we can use either the **Giny Index** or the **Scaled Entropy**

**Stopping criterion** : as seen before, we can use the treshold over the depth of the tree or the homogeneity of the leaves

**Set of decision criteria** : for this hyperparameters we have plenty of choices. Here are the ones tested in this project :

- a single criterion for each categorical features wich checked if the exemple is in the first or the second half of the category
- a criterion for each category, i.e one criterion check if the feature of an exemple is in a specific category
- a criterion for each pair of category
- a criterion for continuous features which check if the exemple is between two constant or not

## 5.2 experiment conducted

Let's see some experiments conducted in this project over the hyperparameters. All the test are evaluated with the accuracy over the test set, i.e the proportion of good predictions of the classifier over the test set. Moreover, the training error is computed and printed at each step of the growth of the tree in order to check how the training error is decreasing during the training.

### 5.2.1 A dumb test

This was the first experiments I tried. We have

- maximum depth = 5
- splitting critarion = gini Index
- for the Criterions of features, we use 1 criterion for each categorical feature and no continuous criterion

this is really not good because of the use of only 1 criterion for each feature. Then we are surely not getting a all the possbility of decision in our tree. And therefore, computing the accuracy, we get an accuracy of 38%. This is really bad, because it's worth than pickin randomly a label for each exemple (around 50% of accuracy, like flipping a coin). Then, it show that chosing good criterions is really important to create a good tree.

### 5.2.2 Splitting criterion test

We now want to compare the the two splitting criterion test. To do so, we use :

- maximum depth = 4
- splitting critarion = gini Index or scaled entropy
- for the Criterions of features, we use all pair of category in each categorical feature

by computing this, we see that the training error is different for the 2 case. However, the 2 accuracy computed at the end are equal. That means that, in our case the two trees created are the same. Therefore, we can say that in our case, the splitting criterion does not change anything in the computation. The difference resides only in the complexity of calculation of the $\psi$ function. Then, for the later experiments, we chose to usually use the giny index because it is easier to compute.

### 5.2.3 Adding continuous criterion

We now add the continuous criterion to our features criterions :

- maximum depth = 4
- splitting critarion = gini Index
- for the Criterions of features, we use all single category in each categorical feature, and continuous criterion are splitted in 4 subset

We also use the *greedyAllLeaves* method to grow the tree. Now, the accuracy is increased to 60% which is better than a random guess but it's still not really good.

### 5.2.4 Change of stop criterion (and comparing method of growing)

This experiment we try to use the homogeneity stop criterion :

- minimum homogeneity = 50
- splitting critarion = gini Index
- for the Criterions of features, we use all single category in each categorical feature, and continuous criterion are splitted in 4 subset
- *growTree* and *greedyAllLeaves* methods are used

The change of the stopping criterion, does not change a lot of thing because we now have an accuracy of 44%. This accuracy is not really good, but I don't exactly know why. It may be criterions that are not chosen well or the stop criterion which is not good (but 50 for homogeneity is already small). Anyway, the interesting part is that we see that the computation of the *greedyAllLeaves* is faster than the usual greedy method. Moreover, the two final tree are the same in this case. This is not really logic, because by growing all the leaves at the same time, it is more probable to reduce too muche the homogeneity of the leaves and to stop the growth of the tree earlier with *greedyAllLeaves* method.

### 5.2.5 Classic test

This test is a regular growing tree in a reasonable amount of time with reasonable parameters :

- maximum depth = 8
- splitting critarion = gini Index
- for the Criterions of features, we use every single criterion for each categorical feature, and 2 criterion for continuous features
- greedyAllLeaves

This test give a quite good accuracy with 71% of accuracy. I think this a good result of accuracy for a regular tree classifier without too much time of computation.

### 5.2.6 Best test

This is the best test I have conducted so far :

- maximum depth = 20
- splitting critarion = gini Index
- for the Criterions of features, we use every single criterion for each categorical feature, and 2 criterion for continuous features
- greedyAllLeaves

This computation takes some times, but at the end, we get an accuracy of 94%. This is really good, but I am a bit surprised that it avoids overfitting.

### 5.2.7 Some remark

I would have liked to perform a cross validation over the training set in order to tune the hyper-parameters. However, since the computation time of training a single tree is already quite long, I would not had time to compute the cross validation.

Same if I had time it could have been interesting to create a method that create all possible criteria for categorical features and to try to compute a small tree with these criteria.

Another remark on the data used. I'm not qualified in mushrooms topic, but since a only some species of mushrooms are edible, it might have been easier to predict the species of the mushroom and then check if the species is edible. In fact, all the mushroom of a same species have quite the same charasteristics over the features. This is not the case for all the edible mushrooms. Then it is probably easier to separate mushrooms by species than by edibility.

## 5.3 Random Forest

In this project, I also tried to implement an algorithm to create a random forest using the previous algorithm of decision tree. The idea is quite simple :

1. we create some subset of our training set by sampling over the rows and the columns (we remove some feature to construct each tree)
2. we construct each tree train tree over all this subset and do a majority vote

If you want further explanation of the idea of random Forest, I let you see here.

For this algorithm, we chose to use for each tree a subset of 67% of the training set sample at random with replacement. On top of that, we remove randomly between 2 and 10 features to construct each tree. This two computation makes the tree more likely to be independants, which is an assumption of bagging method.

#### 5.3.1 Test

Here, I tried to use the random forest algorithm with the following parameters :

- maximum depth = 5
- splitting critarion = gini Index
- for the Criterions of features, we use every single criterion for each categorical feature, and 2 criterion for continuous features
- greedyAllLeaves

for each tree and I compute 21 trees. Then a prediction is computed by computing the prediction for all the tree and taking the majority vote over the trees. The accuracy here is not really good : 55%. I do not really know why, but maybe I took a too restrictive stopping criterion for each tree.

## 6 Conclusion

To conclude this project, we have seen how to construct a tree classifier from scratch. We have also seen what are the hyperparameters of decisions tree and have seen the change they induce in the learning of a tree predictor. Finally, we also implement a algorithm to create random forest. Now, a further implementation in this project would be to perform a cross validation in order to really tune the hyperparameters.

## 7 Bibliography

- Lecture of *Statistical methods for machine learning*, conducted by N. Cesa-Bianchi In the Università degli Studi di Milano Statale during 2023/2024 year.
- Decision ForestsDecision Forests course by Google for Developers
- Chapter 18 of the book “Understanding Machine Learning: From Theory to Algorithms” by Shalev-Shwartz and Ben-David