MACHINE LEARNING AND STATISTICAL LEARNING

EXPERIMENTAL PROJECT REPORT
TREE PREDICTORS FOR BINARY CLASSIFICATION

A.Y. 2023/2024
INSTRUCTOR/DOCENTE: Nicolò Cesa-Bianchi
TAs: Roberto Colomboni and Emmanuel Esposito

Student: Timur Rezepov
Course: B79 (DSE)
Matriculation number: 34177A

Introduction

The goal of this project is the implementation of the tree predictors for binary classification from scratch in Python3. Tree predictor fitting and evaluation is performed using the Mushroom dataset.

This is a descriptive part of the report. The implementation of the tree predictor is described in the *Tree Predictor Implementation* section. Model training and evaluation on the Mushroom dataset was performed in Jupiter notebook in the core directory.

1. Tree Predictor Implementation

The core code contains several modules responsible for different procedures of tree predictor fitting and evaluation:

tree.py

The *Tree Class* implements methods for the construction of the tree and predicting the examples' labels.

splitter.py

The *Splitter Class* is responsible for dataset splitting: finding best split, train/ test, k-folds generating.

entropy.py, impurity.py

Contain classes for entropy and impurity estimation of the data provided.

evaluator.py

Evaluator implements estimation of the model performance metrics: accuracy, precision and recall.

Tree features:

- Tree has a root initialised at start; after the root the child nodes are created recursively if split condition is found.
- Single-feature binary tests are used in each node to asses potential splits.
- For selecting the best split a splitting criterion is used: *Entropy*, *Gini impurity* or *Scaled impurity*.
- Two split conditions are implemented: *Threshold* and *In-set (membership).*
- On each split a stopping criterion is controlled: minimum gain or maximum depth. If the stopping condition is reached the node is converted to a leaf with target label probabilities.

2. Data analysis and preprocessing

The Mushroom Dataset describes physical characteristics of mushrooms and contains binary classification in poisonous or edible.

The primary data set contains descriptions of 173 mushroom species as entries. It can be used to simulate hypothetical mushrooms. In the project the secondary dataset was used, which is a product of such simulation and contains 61,069 hypothetical mushrooms. **class** is the target feature of the dataset, which shows is the example poisonous (p) or edibile (e).

The exploratory data analysis (EDA) consisted of the following steps:

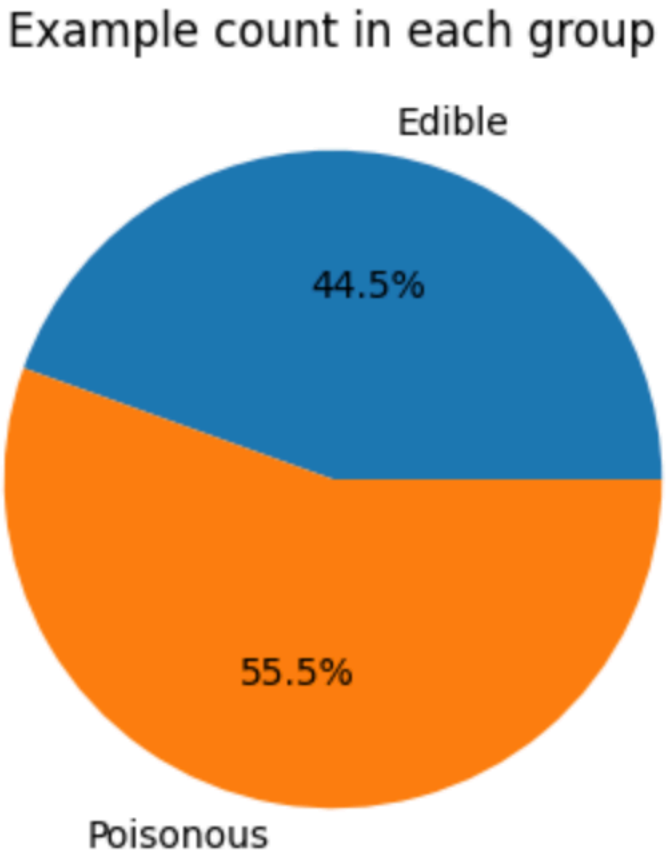
1. Data validation
2. Target feature analysis
3. Missing values analysis
4. Outlier identification for numerical features

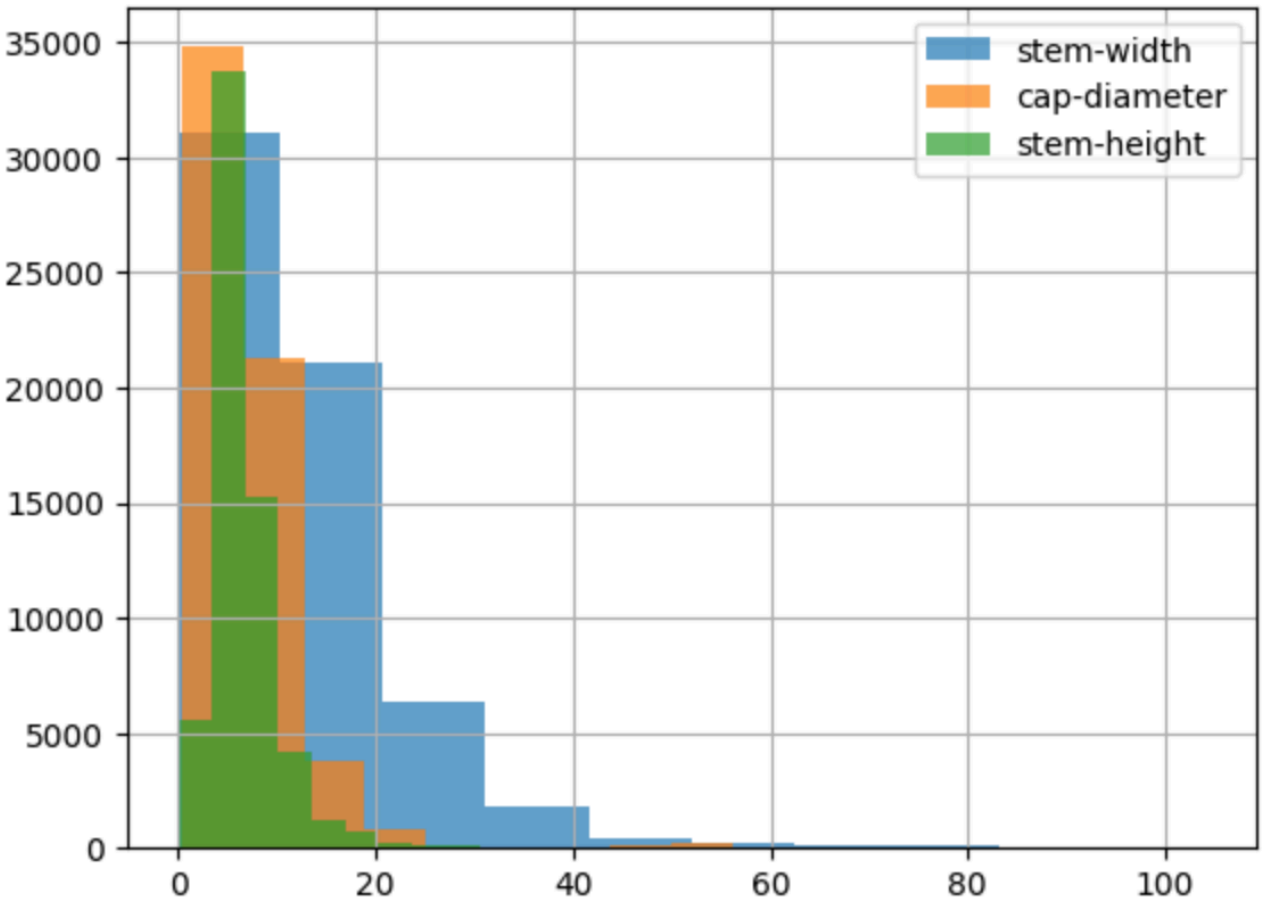
The EDA showed that the dataset is applicable for tree predictor fitting:

- the dataset is of sufficient size and target feature balance
- there are no missing values.

At last, dummy variables were created, which are numerical representation of categorical data, and the train and test splits were created.

```
Data columns (total 21 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   class                 61069 non-null  object
 1   cap-diameter          61069 non-null  float64
 2   cap-shape             61069 non-null  object
 3   cap-surface           46949 non-null  object
 4   cap-color             61069 non-null  object
 5   does-bruise-or-bleed  61069 non-null  object
 6   gill-attachment       51185 non-null  object
 7   gill-spacing          36006 non-null  object
 8   gill-color            61069 non-null  object
 9   stem-height           61069 non-null  float64
 10  stem-width            61069 non-null  float64
 11  stem-root             9531 non-null   object
 12  stem-surface          22945 non-null  object
 13  stem-color            61069 non-null  object
 14  veil-type             3177 non-null   object
 15  veil-color            7413 non-null   object
 16  has-ring              61069 non-null  object
 17  ring-type             58598 non-null  object
 18  spore-print-color     6354 non-null   object
 19  habitat               61069 non-null  object
 20  season                61069 non-null  object
```


Example count in each group
Edible
44.5%
55.5%
Poisonous


35000
30000
25000
20000
15000
10000
5000
0
0
20
40
60
80
100
stem-width
cap-diameter
stem-height

3. Predictor training and evaluation

The quality of classification models can be evaluated using several metrics (all three metrics are implemented in the *evaluation* model):

- **accuracy** shows how often a classification ML model is correct overall.
- **precision** shows how often an ML model is correct when predicting the target class.
- **recall** shows whether an ML model can find all objects of the target class.

First of all, I wanted to decide, which splitting criterion to use: Entropy, Gini impurity or Scaled impurity. For this three tree predictors were trained and evaluated.

Predictor @splitting criterion	Accuracy (avg) train/test	Precision (avg) train/test	Recall (avg) train/test
Entropy	0.616/0.623	0.781/0.782	0.430/0.436
Gini Impurity	0.614/0.619	0.785/0.784	0.422/0.426
Scaled Impurity	0.647/0.651	0.784/0.783	0.505/0.507

There is no problem with overfitting, but looking at accuracy/precision/recall one can see that the model performs differently. When choosing the most suitable model performance metric (or weighted combination) the *cost of error* should be considered. As the goal of the project is to predict, whether the mushroom example is poisonous or not, I decided to move on with the *precision* score as the key model performance metric (we want to be sure that the poisonous label predicted by the model is reliable).

Next, the predictors' performance was assessed using 5-fold cross-validation (the results on the test split are provided). This time I also included precision score standard deviation to assess the *robustness* of the predictor:

Predictor @splitting criterion	Precision avg	Precision stdev
Entropy	0.77	0.02
Gini Impurity	0.78	0.02
Scaled Impurity	0.69	0.11

It can bee seen that the tree predictor created using Scaled Impurity turned out to be unstable (the precision score has high standard deviation). Tree predictor with Gini impurity as splitting criterion seems to be robust and has high precision, so I chose it to use in hyperparameter tuning.

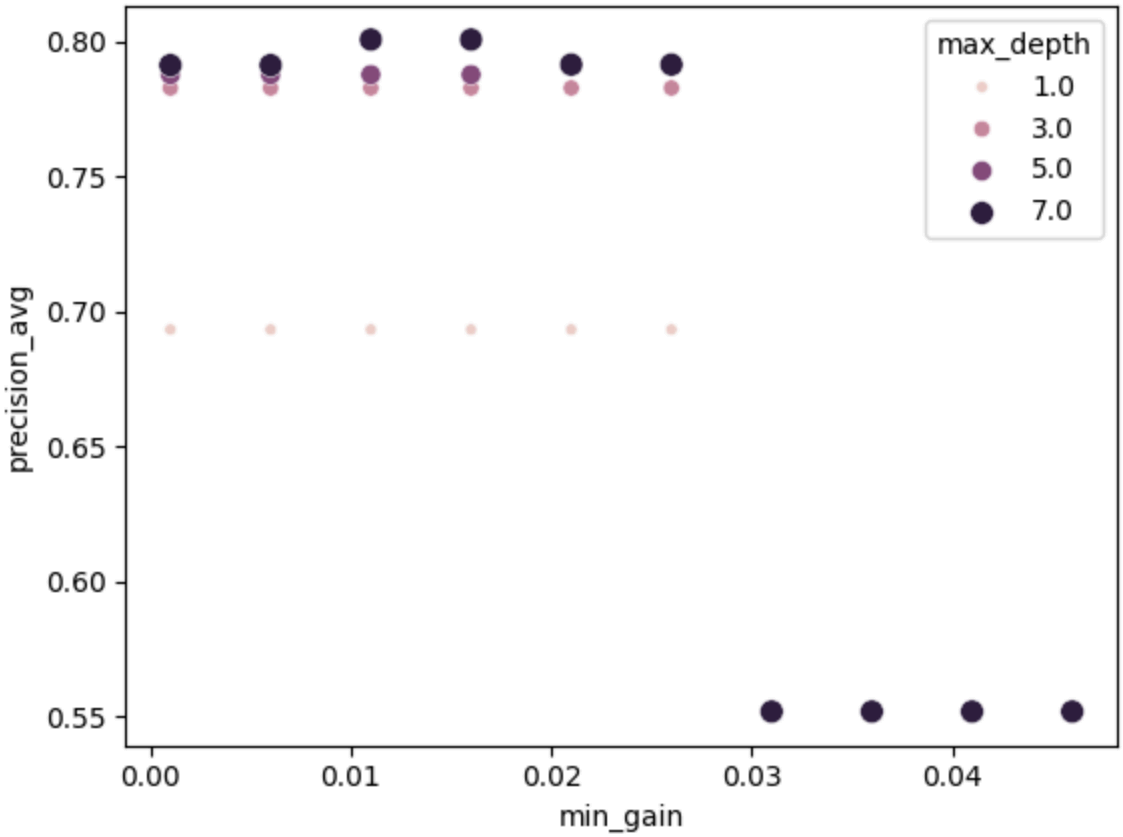
4. Hyperparameter tuning

A hyperparameter is non-learnable parameter, defined before the learning process. It helps to control aspects of the learning process and can be used for example to:

- improve the tree predictor performance
- reduce overfitting
- address class imbalance.

In this project I am using two hyperparameters: minimum gain ([0.001, …, 0.05]) and maximum tree depth ([1, …, 8]). The grid 'surface' of possible values of these parameters was created and the tree predictors were trained and evaluated in each point of this grid.

The chart below shows that minimum gain 0.015 and maximum tree depth 7 correspond to the highest precision. The hyperparameters' tuning resulted in the increase of precision from 0.78 to 0.80.



5. Takeaway Points

Some of this points are obvious, but I wanted to highlight the key points that can be taken away from the job done during the experimental project:

1) Data transformation can significantly reduce the time needed to fit a predictor.

   Introducing dummy variables instead of categorical features reduced the time needed to grow a tree by 6 times: 12 minutes vs 2 minutes. In case of using categorical variables for binary classification tree, one need to use *in-set* condition and all possible membership combinations need to be accounted for. That significantly increase the training time.

2) Model performance should be assessed using cross-validation. This technique not only gives a better performance estimation, but also helps to assess the robustness of the predictor.

3) Model performance should be evaluated strictly according to the business context.

   There is no ‘best metric’ to assess the quality of the model. Considering accuracy, precision and recall in complex with cost of error can lead to more correct decisions about the performance of the machine learning models.

4) Even simple implementation of hyper parameter tuning procedure can improve model performance.

Conclusion

The tree predictor for binary classification application was developed from scratch in Python3 without using any external libraries. Currently it contains all the needed modules and structures to initialise, train and evaluate the tree predictor.

The model was trained and evaluated on the Mushroom Dataset according to the commonly adopted methodology with exploratory data analysis and preprocessing, model performance evaluation, cross-validation and hyperparameter tuning steps. Although the precision (~0.80) and accuracy(~0.62) are good enough, some more algorithm training techniques (or more features) need to be implemented to increase the recall.

However, the implemented algorithm is resilient to overfitting: generated configurations of splitting criteria and hyper parameters didn’t introduce significant difference between model performance on train and test sets.