Robin Erd
Project I
University of Bergen – INF264
Autumn 2021

# Report – Project 1: Implementing decision tree

The goal of this report is to explain the chosen approach and design decisions made regarding the implementation itself as well as the insights gained during evaluation and testing. It is best understood when following the explanations with the code at hand.

## Implementation

This implementation of a decision tree learning algorithm is based on the approach shown to us during the lecture and explained on the project assignment sheet. The implementation and evaluation make use of the packages `numpy`, `pandas`, `seaborn`, `json`, `datetime`, `matplotlib` and `sklearn`. As usual the data is read from a `.csv` file and split into training, validation and test data. The first and most complex step is building the tree, making use of the training data.

### Building the tree

This functionality is implemented in the `buildTree()` function. The tree is stored in a dictionary, using array-like indices as keys. The root is saved to key 0 and its children according to *Formula 1*. This corresponds to the common way of saving binary trees in an array. It can however, due to the size of the tree, not be implemented using an array, as the array quickly grows too big with only small parts of the array with exponentially growing indices being used. The `buildTree()` function checks, implements and handles the three different cases outlined in the assignment Task 1.1 recursively. Therefore, it is always passed the set of datapoints, the target values, the current tree, the key (read: node) it is going to save its result to and the measure it should use to determine the best split (entropy or Gini index). The first call of `buildTree()` gets passed the complete array of datapoints, its target values vector Y, an empty tree (dictionary), a 0 as the key the first node should get saved to and the desired impurity measure.

Formula 1: Key calculation scheme

$$key(leftChild) = 2 * Key + 1$$
$$key(rightChild) = 2 * Key + 2$$

The different cases are implemented as follows:

#### Case 1

Using the numpy `unique()` function we can check how many different values (labels) the target data contains. If all data points have the same label the `unique()` function returns an array containing only this one label. We can then save this label into the dictionary (tree) using the key received as a parameter, creating a leaf. This case is an end-condition of the recursion.

#### Case 2

Again, using the numpy `unique()` function we can check how many different values the data contains. If all data points have identical feature values, we can use the counts per label returned by the `unique()` function to determine the most common label of these data points. We can then save this label into the dictionary (tree) using the key passed as a parameter, creating a leaf. This case is an end-condition of the recursion.

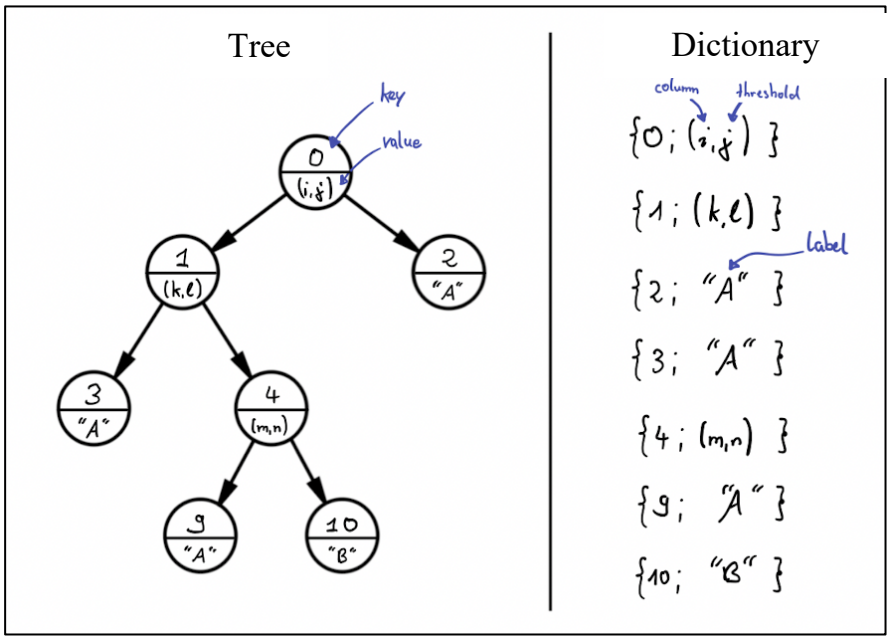
Robin Erd
Project I
University of Bergen – INF264
Autumn 2021

### CASE 3

This case gets all the work done and therefore is the most complex case. It handles the situation in which the data is too diverse to be assigned a fitting label and therefore must be split again. The overarching goal of this case (and function) is recursively splitting the data, thereby reducing the impurity, until every branch of the tree leads to a leaf with a label. To fulfill this goal, we must split in a way that decreases the impurity. Every split can be defined by the feature and the threshold the dataset is split by. We require the feature and threshold combination that maximizes information gain. In order to find the pair which maximizes the information gain, we must compute the impurity measure for every possible pair (split) and subtract it from the original impurity. The splitting itself only makes sense if we sort the data by the respective feature first. As the features and the targets must be in the same order (because this is how features are related to the target values) we must combine the features and targets first, sort them, then split them again using `splitByThreshold()`. Because the values of the features are continuous there is a possible split between every two adjacent datapoints and as we can split by every feature this leads to a nested for-loop. The outer for-loop iterates through the features (columns) and the inner for-loop iterates through all the possible splits for this feature (column). Having found the maximum information gain we save the feature (column) and the threshold used for this (optimal) split in the dictionary (tree). That way this split is reproducible when going through the tree for prediction. We then recursively call the `buildTree()` function for each of the two halves of the split, passing the new subsets of data, the modified tree, the new key (according to *Formula 1*) to save the column and threshold split by to, and the impurity measure to use.

This way, in the end, the tree contains a list of keys and values, where every key is the id of a node and the value is a tuple, containing the threshold and column to split by (see *Figure 1*).

Figure 1: Tree-Dictionary Mapping



Robin Erd
Project I
University of Bergen – INF264
Autumn 2021

## Reduced-error pruning

Before moving on to the prediction stage one can opt to prune the decision tree. Reduced-error pruning is a measure to reduce overfitting. It works by evaluating the accuracy of the original tree on a given data set (pruning data) and comparing it to the accuracy of a modified version of the tree on this same data set. The modification of the tree is the replacement of a subtree with the label that is most common in this subtree's leaves. For this procedure to work properly one must start from the bottom of the tree, with the smallest subtrees. The modification is only kept if the accuracy of the modified tree is higher or equal to the accuracy of the original tree.

This functionality is implemented (as requested in Task 1.3) as optional part of the `learn()` function in a function called `pruneTree()`.

The pruning itself is implemented as follows:

At first, we get a list of all the available nodes (keys) in the tree (dictionary). From those we discard the ones which are leaves (have labels as values). The rest of the keys can then be sorted in reverse order. Because of the way the nodes are stored in the dictionary a bigger key always corresponds to a node (subtree) on a lower (or equal) level. We can then iterate through this reversed list of nodes, which corresponds to iterating over the subtrees from the bottom up. For each subtree the ("original") accuracy (of the whole) tree on the pruning data set is computed and stored. Then the tree is modified (the most common label in this subtree is found and the value of the node is replaced with this most common label using `labelCounts()`) and the accuracy of the prediction made by this modified tree on the pruning data is stored. The comparison of this accuracy with the original accuracy on the pruning data then decides whether the modification will be kept or reversed. If the modification is kept the entries in the dictionary of this subtree can be deleted using `purgeSubtree()` (of course except for the root of the subtree now containing the leaf). Then the next for-loop begins, this time possibly with the modified tree as the "original" tree.

## Prediction

In order to walk through the tree to make predictions one starts at the key 0 and retrieves the value of this key from the dictionary (tree). If the value is a tuple (containing column and threshold), we have not yet reached a leaf and therefore must follow another branch. We do that by testing whether the feature in the given column of our given data point is smaller than or equal to the threshold. If this is the case, we must follow the left branch, which corresponds to doubling the key and adding **one**. Otherwise, we would have to multiply by two and add **two**. We then retrieve the value of this new key from the dictionary (tree). This repeats until the dictionary (tree) returns a label, not a tuple.

This label is the prediction for the given data point. This is implemented in a for-loop, iterating through all elements in the input data set, appending the predictions to a list which is, eventually, returned.

Robin Erd
Project I
University of Bergen – INF264
Autumn 2021

## Evaluation and model selection

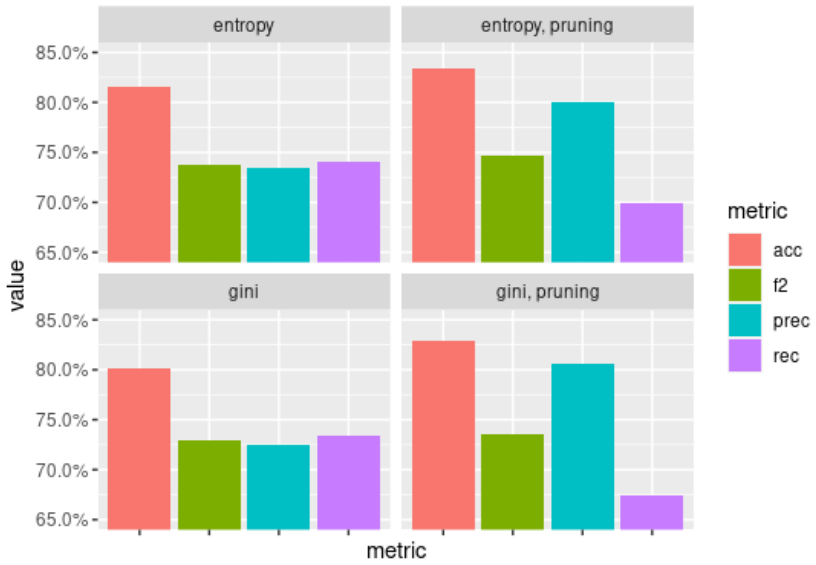
This paragraph evaluates the performance of the implemented decision tree classifier. The comparison of the predictions and the actual (observed) target values using various metrics shows how our algorithm performs with different configurations and allows the selection of the best performing configuration. This configuration can then be used to perform the final evaluation on the test data.

There are two parameters controlling how the tree is built (or: the algorithm learns). Pruning can be enabled or disabled and there are two different impurity measures (entropy or Gini index) to choose from. This leads to four different configurations to test on the MAGIC Gamma Telescope dataset[1] which is split into training data (11411 data points), validation data (3804 data points) and test data (3804 data points). Each of the results in *Table 1* was achieved after training the model on the training data and applying the learned model (tree) on the validation data.

Table 1: Performance on validation data

| Configuration | Time [s] [2] | Accuracy | F2 Score | Precision | Recall |
|---|---|---|---|---|---|
| Gini Index, pruning | 2669.41 | 0.82913 | 0.73491 | 0.80662 | 0.67491 |
| Gini index | 2609.78 | 0.80062 | 0.72917 | 0.72432 | 0.73408 |
| entropy, pruning | 2414.72 | 0.8336 | 0.74690 | 0.80103 | 0.69964 |
| entropy | 2286.26 | 0.81519 | 0.73759 | 0.73512 | 0.74007 |

Figure 2: Performance on validation data




The first thing to notice is that the pruning does not seem to have a significant impact on how long constructing the tree (learning the model) takes, as it only adds about 60 seconds to the process. The duration is affected by the impurity measure chosen, due to the formula for computing the Gini index being different to the formula for computing the entropy. Regarding

[1] https://archive.ics.uci.edu/ml/datasets/magic+gamma+telescope
[2] executed online on a Google Colaboratory machine equipped with an AMD EPYC 7B12

the prediction-performance of the algorithm the chosen configuration also makes a difference. Choosing entropy as the impurity measure results in slightly higher accuracy, f2 score and recall both with pruning enabled and disabled.

The performance measures also show that, independently of the chosen impurity measure, pruning affects the precision and recall values, increasing the precision and accuracy and in turn reducing the recall.

Between the two different entropy configurations I assume it is better to choose the version with pruning enabled, as a pruned model is simpler and therefore more general. In addition, the drop in recall caused by pruning is smaller than the gain in precision. Therefore, we can conclude that the best performance on the validation data can be achieved using entropy as an impurity measure with pruning enabled. I estimate the performance for the unseen data to be slightly worse than on the validation data due to the validation error systematically underestimating the test error (if validation data is used for model selection, then validation error is essentially a form of training error). Nevertheless, I believe one can reasonably assume that the variance in the performance of the different configurations provides a good orientation for how much the performance on the test data might vary from the performance of the chosen model on the validation data.

## Comparing to an existing implementation

The comparison of my implementation with the existing implementation in the `sklearn` package reveals interesting differences. While the `DecisionTreeClassifier` from `sklearn` requires only a fraction of the time my implementation does, it performs a little worse in terms of accuracy, f2 score and precision (see *Table 2*). Of course, the implementation used in `sklearn` is to be preferred as the trade-off is not very good and therefore does not make sense in practice – it requires 0.0001 times the time (although there probably is room for lots of improvements in my implementation) and only sacrifices ~1.314% of accuracy and ~5.237% precision for that and even achieves about 4% better recall (more uniform scores).

While I have not looked at the source code of the `DecisionTreeClassifier`, I can imagine that with a deeper understanding of the implementation of python standard library functions and the use of concurrency instead of recurrence a better performance can be achieved. In general, the price of the relative ease of implementation of recursive functions is that they pose an inefficient approach to most problems. One issue identified in my implementation is the `splitByThreshold()` function performing bad, with the `buildTree()` function spending most of the time executing the `learn()` function in this function.

Table 2: Compared performance on test data

| Configuration | Time [s] [3] | Accuracy | F2 Score | Precision | Recall |
|---|---|---|---|---|---|
| Sklearn | 0.23648 | 0.82124 | 0.75096 | 0.74763 | 0.75423 |
| Custom [Pruning, Entropy] | 2401.02 | 0.83438 | 0.75505 | 0.80049 | 0.71449 |

[3] executed online on a Google Colaboratory machine equipped with an AMD EPYC 7B12