Imperial College London

Coursework

Imperial College London

Department of Computing

CO395 - Decision Tree Coursework

Authors:
Vincent Jarasse
Alexis Laignelet
Shu Okabe

Date: February 11, 2019

1 Brief summary of the implementation details

In this section, we will describe the main parts of our implementation and detail specific parts. Following the provided specifications, our code is split into 5 main parts : the "`find_split`" function, the tree generation algorithm itself, the graphics part (bonus), the evaluation and metrics part and finally the pruning. We will explain the specificities of each of these parts hereafter.

1.1 Find Split function and associated sub functions

In the decision tree algorithm, the decisive point is where to split the dataset in order to maximise the gain as defined below:

$$Gain(S_{all}, S_{left}, S_{right}) = H(S_{all}) - Remainder(S_{left}, S_{right})$$

with:

$$Remainder(S_{left}, S_{right}) = \frac{\mid S_{left} \mid}{\mid S_{left} \mid + \mid S_{right} \mid} H(S_{left}) + \frac{\mid S_{right} \mid}{\mid S_{left} \mid + \mid S_{right} \mid} H(S_{right})$$

The `find_split` function studies every split possibility i.e. it tries every split for every feature (here the 7 WiFi emitters) and computes the associated gain. The algorithm returns the feature number (from 0 to 6) and the value associated with the best split.

Let's make a few notes on our implementation. First, we implemented 3 sub functions that will make the `find_split` code cleaner : `compute_entropy(dataset)` which gives $H($`dataset`$)$, `split_dataset(dataset, idx)` and `extract_from_array(full_array, start_index, length)` which combined together allow to simply extract one part of a 2 dimensional array and returns the extract as well as the 2 remainder parts merged together. Now about the `find_split` itself. Its behaviour is already described above, but there are a few specificities we would like to highlight. First, every time we consider a new feature, we sort the dataset based on the value present in that feature. To avoid useless computations, we also test (in the line "`if minval == maxval: continue`") if the feature only contains the same values, which is a tricky case where we do not want to split. Another specificity is the following part at the end of the function:

```
if sorted_dataset[idx, feature] == minval:
    # Avoid the case where the split point is at the edge of the dataset (and thus we could not split,
    best_value = sorted_dataset[idx, feature] + 0.5
elif sorted_dataset[idx, feature] == maxval:
    best_value = sorted_dataset[idx, feature] - 0.5
else:
    best_value = sorted_dataset[idx, feature]
```

Indeed, when the best split is achieved for a value that is at the extremity of the sorted dataset, we could encounter an issue when we actually split the data: if we take the convention to split for values $>$ of the split value, we will have a problem when the split value is the max. The inverse is true for the min. That is why when the split value is the min or the max, we slightly modify it to cope with that issue. This does nos affect the gain after the split, as the values of the dataset are integers, and we only add or subtract 0.5 to them.

1.2 Algorithm to generate a decision tree

Based on the previous functions, especially the `find_split` function, we can now implement the recursive function `decision_tree_learning` in order to create the decision trees. To do so, we use the given pseudo-code which is taken from the paper *Artificial Intelligence: A Modern Approach* and adapted to our case. The function itself takes a labelled dataset `data` and a `depth` argument and returns a tree in a form of a Python dictionary and its depth.

As a reminder, the general structure of a tree is: {`'attr', 'value', 'left', 'right'`}, where the attribute and the value are the name of the best feature and its best threshold value to split the dataset, according to the `find_split` function. The left and right keys store the left and right branches from the node; we chose to put in the left branch the sub tree corresponding to values below the threshold and in the right branch the one with values above. Finally, we chose to put the attribute `'leaf'` for the leaves with the corresponding class number in the `'value'` key.

The idea of the algorithm is to recursively find the best split of the given dataset, create a sub tree according to the split feature and its threshold value, separate them into two branches and execute the algorithm on the two new sub trees. When the given dataset has only one label (which is tested with the condition `len(np.unique(data[:, 7])) == 1`), the algorithm returns a leaf.

The `depth` element that is given as an output of the `decision_tree_learning` function is the depth of the whole tree, found by recursively taking the maximum of the depth of the right and left sub trees.

1.3 Graphics: decision tree visualisation

The aim of the graphical representation is to give a visual intuition of the overall tree as well as some characteristics of the nodes (split values, features, etc).

The main idea of the implementation is to explore recursively the tree.

- If the current node is a leaf, then a bubble text with the main info is displayed.
- If the current node has children, a bubble text is shown and two segments are also displayed below the current node to link it with its children. Then, the function is applied to the left child and the right child.

The position of the nodes is calculated with two values: `delta_x` and `delta_y` used to adjust the position of the children given the position of a node. As the tree is binomial, it makes sense to divide by two this `delta_x` each time we go one layer deeper in the tree. As for `delta_y`, this parameter can be constant.

The overall representation of the tree is this one:

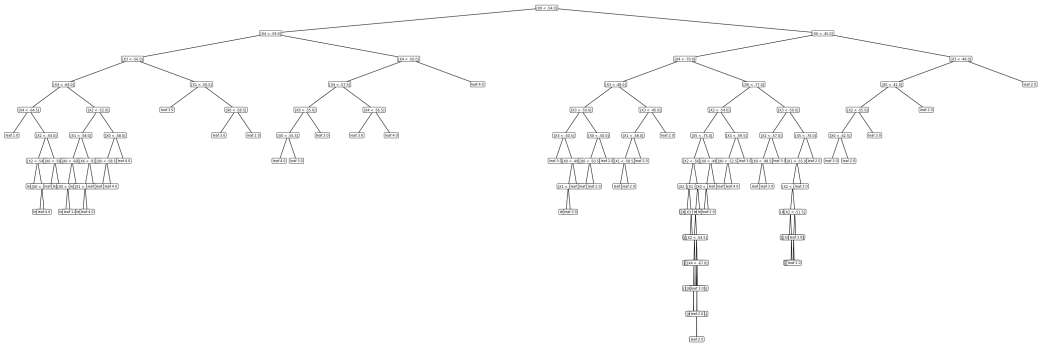

Figure 1: Representation of the whole decision tree

It gives a general view of the tree, but it can be hard to read for huge trees. This is why the plot function has an optional parameter to indicate the desired maximum depth.

For example, with a `max_depth` of 3:

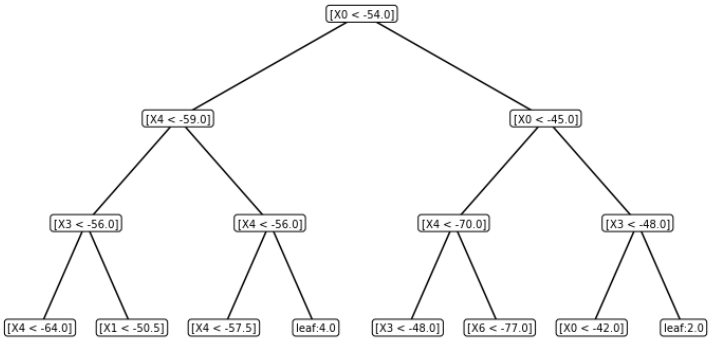

Figure 2: Representation until the depth 3 of the decision tree

1.4 Evaluation and metrics, including cross-validation

This section is composed of 4 methods: the evaluation function, the cross-validation and three sub functions. The evaluation function `evaluate(test_db, trained_tree)` takes a test dataset and an already-trained tree and outputs the achieved accuracy, as well as the predicted and true labels for the test dataset, which will be useful inputs for the other metrics functions. There are three of them: `confusion_matrix(true_labels, predicted_labels)`, `recall_precision(cm)` and `f1_measure(recall, precision)`. These functions are quite simple and do not require further explanations, apart from the fact for the last two, they output averages on each label class (rooms 1 to 4). These functions will be called in the cross-validation algorithm as explained below.

The 10-fold cross validation algorithm for this part (before pruning) works as follows:

- Shuffles the dataset and splits it into 10 folds (tuples of the form: (training set, validation set)).
- For each fold:
  - build the tree on the base of the training dataset
  - evaluate the performance of the model on the validation dataset, by outputting the following metrics: the accuracy, the confusion matrix, the recall, precision and F1-measure.

Warning: in this version, there is no differentiation between validation and test sets, because we do not tune any parameter based on the result of the validation dataset: the validation dataset is actually a test dataset. In the pruning section, we define a second version of this cross-validation where such a difference is made.

The 10-fold cross validation outputs a bunch of results that can be analysed by another function, `analyse_metrics(cv_results, folds)` that takes the raw output of the cross-validation and digests it to give the average of every metric : `mean_accuracy`, `mean_cm`, `mean_recall`, `mean_precision`, `mean_f1`, `sum_cm`. The difference between the two confusion matrices is if we want to have the overall count (`sum_cm`) or just a percentage(`mean_cm`).

1.5 Pruning

1.5.1 Pruning algorithm

In order to prevent overfitting, and hence improve the accuracy of the decision tree on an unseen test dataset, we can implement a pruning algorithm. The basic idea of pruning is to find a node with two children which are leaves and see if by merging them the performance of the new tree is better than the original one. If it is better we continue the algorithm with the new version of the tree.

To implement such a function, we first need two sub functions: `merge_some_leaves(tree, rev_path, value)` and `compare_trees(test_db, original_tree, new_tree)`. The aim of the first function is to have a new tree where two leaves are merged. Indeed, it takes an already-trained tree, a path to access the node with two leaves and a class value for the future leaf, in order to output the new tree. A path here is a list of tuples with the following structure: `(attribute, value, direction)`, where the attribute and the value are the name and value of the node, and the direction is a string stating if the next node is the right or left branch. In the list, the tuples are in the root-to-leaf order i.e. the first element is the root and the last is the node which actually has two-leaved children.
The second function is here to compare the performance of two trees on a test/validation dataset and returns the best tree. To compare the performance, we chose to use the classification error with the help of the `evaluate` function created in the previous step.

Now that the sub functions have been explained, we can implement the pruning algorithm; we chose to do it recursively like in a depth-first search algorithm i.e. it traverses the tree

from the root to the leaf and from left to right. The function `pruning_aux(node, path, whole_tree, test_data)` takes indeed the current node, the path to access it, the state of the whole tree, and the testing dataset to compare the possible new trees and outputs a possibly-updated version of the node, the path and the whole tree. The idea of the algorithm is as follows:

- If the current node is a leaf, we go back to the parent node. This means that if it is a left leaf, we have to check if the right one is also a leaf. If it is a right leaf, we have to check if pruning the tree at the parent node is better for the accuracy on the validation dataset.

- If the current node is a "normal" node, we look first look at the left branch and update it with the `pruning_aux` function while tracking the path. Then, we update similarly the right branch. This means that:

  - If the node has at least one child which is not a leaf, we check the next nodes recursively to update the children branches. This notably enables to test pruning when the children nodes were just pruned and that the parent node might satisfy the pruning conditions.

  - If the node has two direct leaves as children after looking at the two branches, we have to see if pruning i.e. transforming the current node into a leaf with one of the two possible classes from the children is better for the accuracy on the test dataset. This is the case tested under the condition `if (updated_node['left']['attr'] == 'leaf') and (updated_node['right']['attr'] == 'leaf')`.
    First, we store the two possible label values and create two corresponding new trees with the `merge_some_leaves` function.
    Then, we compare both trees with the current original tree, and take the best among those three trees with the `compare_trees` function.
    If the best tree is a new one, we update the whole tree with the best new tree and transform the current node into a leaf with the best label.

Finally, the final `pruning` function uses the previous recursive function and only needs the already-trained tree and the testing dataset, and returns the new pruned decision tree.

1.5.2 Cross validation version 2 for pruning

The pruning consists of tuning the decision tree according to the accuracy on the validation dataset. Thus, in order to provide unbiased test results, we need to clearly differentiate the training, validation and test datasets. This is done in the sub function `separate_dataset(dataset, folds)` which outputs a (10, 2) matrix. The first dimension represents the 10 test datasets, and the second dimension contains the corresponding 9 sets of training/validation datasets.
Note that the `cross_validation_pruning(dataset, folds, all_combination = True)` contains a parameter `all_combination` which, when set to False, only perform the algorithm on the first of the 10 folds, thus averaging on 9 results on the test dataset.

2 Comments on the results of part 1

Note: all of the results below were obtained before pruning.

2.1 Raw results of part 1

The interpretations of the following results are in the next subsection.
Please find below the "average" (actually sum of 10 confusion matrices) confusion matrices for the training and evaluation on the clean dataset and on the noisy dataset respectively.

Class	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Actual 1	493	0	4	3
Actual 2	0	484	16	0
Actual 3	2	15	478	5
Actual 4	5	0	4	491

Class	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Actual 1	391	26	30	43
Actual 2	30	414	34	19
Actual 3	35	38	408	34
Actual 4	46	27	35	390

The corresponding confusion matrices expressed in percentage are also available :

Class	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Actual 1	24.65	0	0.2	0.15
Actual 2	0	24.2	0.8	0
Actual 3	0.1	0.75	23.9	0.25
Actual 4	0.25	0.	0.2	24.55

Class	Predicted 1	Predicted 2	Predicted 3	Predicted 4
Actual 1	19.55	1.3	1.5	2.15
Actual 2	1.5	20.7	1.7	0.95
Actual 3	1.57	1.9	20.4	1.7
Actual 4	2.3	1.35	1.75	19.5

From the confusion matrices above, we can derive the average recall and precision for clean and noisy datasets respectively:

Recall	Room 1	Room 2	Room 3	Room 4
Clean	0.9855	0.9669	0.9565	0.9815
Noisy	0.7999	0.8337	0.7960	0.7821

Precision	Room 1	Room 2	Room 3	Room 4
Clean	0.9861	0.9712	0.9522	0.9830
Noisy	0.7816	0.8206	0.8069	0.7995

And finally the average F1-measure per class on the two datasets:

F1-measure	Room 1	Room 2	Room 3	Room 4
Clean	0.9856	0.9688	0.9539	0.9822
Noisy	0.7901	0.8250	0.7995	0.7879

Finally the average classification rate achieved on the two datasets :

Dataset	Classification rate
Clean	0.973
Noisy	0.824

2.2 Interpretation of these results

The metrics presented above clearly show two trends. The first one, which was expected, is the importance of the quality of the data for the machine learning model to work properly. This point will be discussed more deeply in section 3.1. The second remark concerns the behaviour of the model with respect to the different classes. Indeed, the confusion matrices show that rooms 2 and 3 are harder to distinguish than the two other rooms. To be more specific, we even see a discrepancy between the recall/precision of the clean and noisy datasets. Indeed, the recall and precision of rooms 2 and 3 are higher than the two other rooms, whereas it is the contrary in the clean dataset. This would mean that the clean dataset tends to predict more rooms 1 and 4 than the noisy dataset, proportionally to its classification rate of course. Finally, we could formulate the possibility that the shape of the apartment, and more specifically the symmetry of rooms 2 and 3 wrt the WiFi emitters could be the root cause of this confusion between these two rooms.

3 Answers to the questions of part 2

3.1 Noisy-Clean Datasets Question

First, there is a clear difference of prediction accuracy between the clean and the noisy dataset: appr. 80% for the noisy dataset compared to appr. 97% for the clean dataset. This is a good example of the importance of the data quality for a machine learning problem: all come from the data, so the better the data, the better the model. Secondly, there is a little difference of behaviour in the model predictions between the two datasets, as described in section 2.2. Finally, the pruning does not have the same effect on the two datasets. While on the clean dataset, in does not improve that much the already -very- good accuracy, it does improve a bit (by appr. 2%) on the noisy dataset. However, the fact that the data is noisy may have the same effect as a regularisation, and thus already prevents overfitting.

3.2 Pruning Question

The pruning function has been detailed in the pruning algorithm description here.

When it comes to the performance we can see an improvement of the accuracy thanks to pruning. For the clean dataset, before the pruning we have an average accuracy of 81.06

and after pruning we have an average accuracy of 81.96. For the noisy dataset, before the pruning we have an average accuracy of 80.98 and after pruning we have an average accuracy of 82.04. Hence, we can see that on the noisy dataset, we obtain a 2% improvement on the accuracy with the pruning, while on the clean one, we have an improvement of less than 1% on the accuracy.

We can explain that this slight change in accuracy for the clean dataset is due to the quality of the dataset that leads to a good decision tree classifier; improving it with pruning is thus more difficult than for the noisy dataset one.

3.3 Depth Question

For the clean dataset, before the pruning we have an average maximal depth of 12.89 and after pruning we have an average depth of 12.78. For the noisy dataset, before the pruning we have an average maximal depth of 19.33 and after pruning we also have an average depth of 19.33. First we can see that when the dataset is clean, the maximum depth of the tree is lower than when the dataset is noisy. Moreover, after pruning we can see that the maximum depth is less than before the pruning.

If we also take into account the accuracy of the tree for both datasets before and after pruning, we can see that the a deeper tree does not necessarily mean a better prediction accuracy. Indeed, deeper trees are sometimes a result of overfitting on the training dataset, leading to an excessive split in the tree. There thus should be a compromise in the maximal depth: if it is too shallow it is not strong enough to have a good prediction accuracy, but if it is too deep it will overfit.