Introduction to ML - Decision Tree Coursework

Kevin Landert, Constantin Eulenstein, Daphne Demekas, Nasma Dasser

October 29, 2020

1 Introduction

Decision tree learning is a powerful predictive modelling technique used in machine learning for classification and regression. This report documents our python implementation of the Iterative Dichtomiser 3 (ID3) algorithm developed by by Ross Quinlan [2]. Our algorithm aims to predict the room in which a user is located based on seven WiFi signal strengths that are captured from the user's mobile phone. To train and test the algorithm, we use two separate "clean" and "noisy" datasets, (Figure 1), both containing 2000 samples of seven continuous variables (signal strength from seven emitters) and one categorical variable (the room in which the user is located).

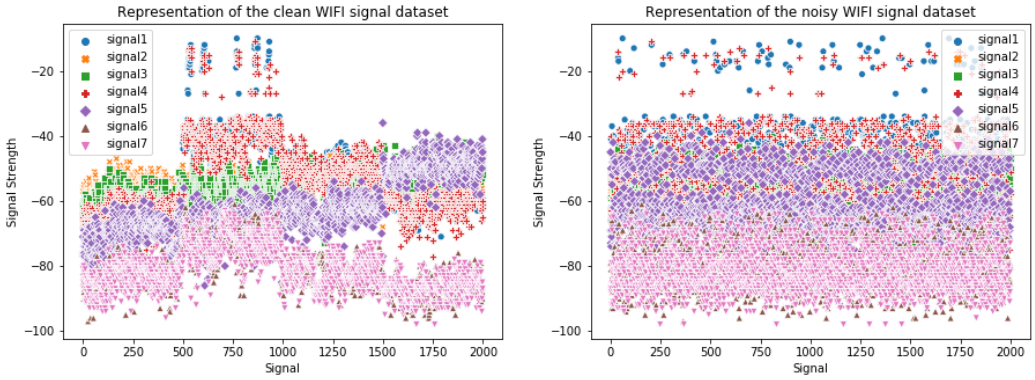


Figure 1: Scatter plot of the dataset to classify, (left) clean dataset, (right) noisy dataset.

We begin by highlighting the mathematical background necessary to solve this problem. Next, we briefly explain our implementation. We then show how we build our decision tree, evaluate our tree on test data and how to avoid over-fitting using pruning. Finally, we discuss the performance of the algorithm on the two datasets.

2 Theoretical Background

The ID3 algorithm starts at the root node of the tree with the full dataset considered. It then repeatedly partitions the data into subsets that achieve the highest information gain according to information theory [3]. If we consider two subsets of a dataset S with K different labels as (S_{left}, S_{right}), we calculate the Entropy H as

$$H(S) = -\sum_{k=1}^{k=K} p_k \log_2(p_k), \tag{1}$$

where p_k is the probability of the label k, i.e. the ratio of the number of samples with the label k to the total number of samples in S. The Information Gain IG is then defined as

$$IG(S, S_{left}, S_{right}) = H(S) - Remainder(S_{left}, S_{right}), \tag{2}$$

where the *Remainder* is equal to

$$Remainder(S_{left}, S_{right}) = \frac{|S_{left}|}{|S_{right}| + |S_{left}|} H(S_{left}) + \frac{|S_{right}|}{|S_{right|} + |S_{left}|} H(S_{right}). \tag{3}$$

Once the split points in the attribute resulting in the highest information gain are found, we iterate this procedure for every subset, thereby creating a tree, where every internal node corresponds to a split point. This recursive procedure stops once all samples have the same label, indicated by a "leaf" in the tree. A node is defined as the point of split between two values of one attribute.

3 Algorithm Implementation

The implementation of our algorithm is split into three python files that can be found at [1] in the `src` directory. The `DecisionTree.py` file contains all major functionality of the decision tree itself, the `validation.py` file contains the code to performs cross validation and the `utils.py` contains several utility functions.

3.1 Building a Decision Tree

We first defined a class called `DecisionTree` which is used to build, train and evaluate a decision tree. The tree can be created by calling the build `buildTree` function. The training data can either be passed to this function or to the constructor of the class. In the second case, the constructor implicitly calls the `buildTree` function. Internally, the tree is represented as a nested dictionary (Figure 2). Each dictionary contains an attribute, value and a left and right branch consisting of a subtree. The structure of the dictionary looks like this:

```
node = { attribute: int or str,
         value    : float,
         lbranch  : dict node,
         rbranch  : dict node},
```

where `attribute` is either an integer (index of the signal on which we split) or the string 'leaf', signifying if the node is a terminal node. If the node is not a leaf, `value` is the split value of a particular node and `lbranch` and `rbranch` hold the two subtrees. If a node is a leaf, `value` indicates the label assigned to this partition of the data. The `buildTree` function recursively builds a tree by partitioning the data into two partitions as explained in Section 2. Details of the implementation of `buildTree` closely follow the specifications described in Algorithm 1 of the Coursework.

```
{'attribute': 2,
 'val': -52.0,
 'lbranch': {'attribute': 0,
  'val': -43.0,
  'lbranch': {'attribute': 2,
   'val': -54.0,
   'lbranch': {'attribute': 0,
    'val': -44.0,
    'lbranch': {'attribute': 'leaf',
     'val': 1.0,
     'lbranch': None,
     'rbranch': None},
    'rbranch': {'attribute': 'leaf',
     'val': 2.0,
     'lbranch': None,
     'rbranch': None}},
   'rbranch': {'attribute': 'leaf',
    'val': 3.0,
    'lbranch': None,
    'rbranch': None}},
  'rbranch': {'attribute': 'leaf',
   'val': 2.0,
   'lbranch': None,
   'rbranch': None}},
 'rbranch': {'attribute': 'leaf',
  'val': 3.0,
  'lbranch': None,
  'rbranch': None}}
```

Figure 2: Example of an internal representation of the tree as nested dictionary.

3.1.1 Finding the best split

To find the best partition that results in the highest information gain, we implemented a function `findSplit`. It takes a subset of the data we want to partition and finds the best split according to Equations 1,2 and 3 in Section 2. The function iterates over all signals and every potential split `value` and computes the information gain (Equation 2). After splitting the data where the highest information gain is achieved, we call the `partitionData` function. Given the best split point as input this function returns two subsets according to this split rule. We then recursively call the `buildTree` function on these subsets until all elements of a partition have the same label.

3.2 Evaluation of a Single Tree

To evaluate if our tree is successful at predicting labels we call the `evaluate` function with unseen test data. We traverses the tree for a new input data point by calling the internal `predict` function. This function returns our best guess for the sample. The `predict` function takes the signal strengths for a single point and recursively traverses our tree until we reach a leaf node. This node then contains the value of our best guess based on the previously learned training data. The `evaluate` function loops over the test set and compares every prediction - the return value from the `predict` function - to the true label for every data point. Based on this prediction and the true label we fill a confusion matrix. We then use this confusion matrix to compute the standard evaluation metrics: *precision*, *recall*, $F1$ and *accuracy* (Appendix A). The `evaluate` function returns all these metrics as a dictionary in the following form:

```
statistics = {'confusionmatrix': float #shape (4,4)
              'precision'      : float #shape (4,),
              'recall'         : float #shape (4,),
              'F1score'        : float #shape (4,).
              'posClassRate'   : float} #shape (1,).
```

3.3 Evaluation of our Implementation

The evaluation methodology in section 3.2 can be used to test a specific tree. However, as a single tree is highly dependent on the training and test data size, we perform a 10-fold cross validation on our decision tree algorithm for both the clean and noisy dataset to evaluate its performance. We begin by calling the function `crossValidation_split` which randomly splits our data into K = 10 folds. We save the indices of every fold in a list. The function `cross_val` then performs K loop iterations, setting aside a dedicated test fold for every iteration. From the remaining K - 1 folds one is additionally saved for validation during the pruning step and the other K - 2 folds are used to train our tree. Once again we loop over our K - 1 remaining folds and choose a different fold for validation every round. During every loop iteration a new tree is built with K - 2 folds as training data and one fold is used for pruning, which will be discussed in the next section. We evaluate both the unpruned and pruned tree with the help of our dedicated test fold by calling the `evaluate` function. We average the evaluation metrics for every loop iteration for both the unpruned (Figure 3) and pruned tree separately with the help of the `average_statistics` function. The averaged results are then returned in a dictionary of the same form as described in section 3.2.

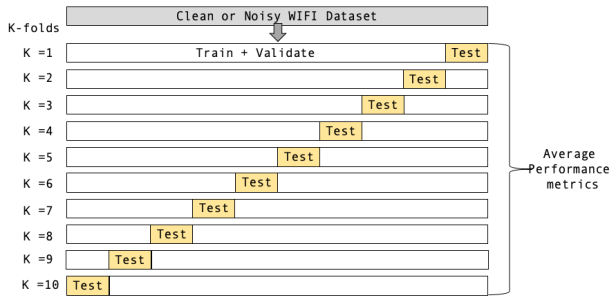


Figure 3: Implementation of the 10-Fold cross validation.

3.4 Pruning

To avoid overfitting in the case of a noisy dataset, we implement a pruning function in our decision tree class. The `prune` function receives the training and validation data as input and calls the internal `pruneNode` (Algorithm 1) function. This function then recursively tries to prune our tree wherever a node is connected to two leafs. The `pruneNode` functions saves the state of the current node and recursively calls `pruneNode` on both branches if it is not a leaf. It then checks if its connected to two leafs. If this is the case, it computes the pre-pruning accuracy score. Then it replaces the current node by a leaf and recomputes the accuracy.

If the new tree has a higher accuracy score, we keep the pruned tree, otherwise we restore the current node to its old state. This implementation makes sure that the tree is pruned bottom up, to ensure that a node is only checked after its children. This makes the implementation very efficient as we only check every node once.

Algorithm 1 pruneNode

Input: node, training data,validation data
Output: depth d of pruned tree
node[*attribute*], *node*[*value*], *node*[*lbranch*], *node*[*rbranch*] in current node

```
if node[attribute] is a leaf then
 | return depth d =1
else
 |  ldepth ← pruneNode(node[lbranch], lbranch data)
 |  rdepth ← pruneNode(node[rbranch], rbranch data)
 |  if lbranch[attribute] and rbranch[attribute] are leaves then
 |  |  calculate pre-prune accuracy A_old
 |  |  delete node[lbranch]
 |  |  delete node[rbranch]
 |  |  node[attribute] ← leaf
 |  |  node[value] ← max(count(training data[values]))
 |  |  x calculate post-prune accuracy A_new
 |  |  if A_new > A_old then
 |  |  |  keep pruned tree
 |  |  |  return depth d =1
 |  |  else
 |  |  |  restore old tree
 |  |  |
 |  |  end
 |  else
 |  |  return max(ldepth,rdepth) + 1
 |  end
end
```

3.5 Visualizing the Decision Tree

To visualize a decision tree, we have created the `visualize` function. This function recursively traverses the tree and builds a visualization similar to the unix file-system tree. This function calls the function `formatNode` which formats every node dictionary into a nice string and then builds our tree with String concatenations. Calling this function on both the unpruned and pruned tree nicely illustrates how pruning reduces the complexity of our tree on a noisy dataset.

3.6 Robustness of the Implementation

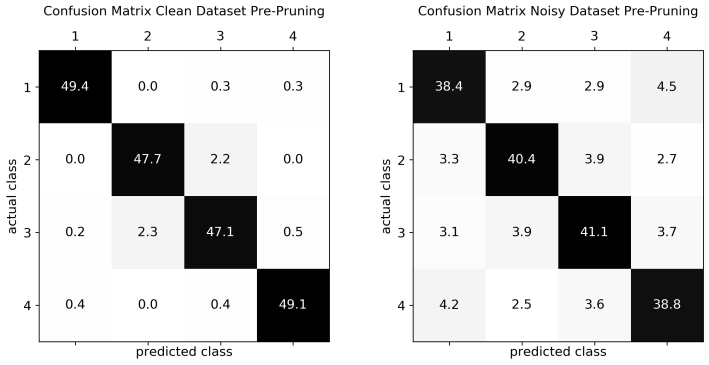
In our `buildTree` we considered additional edge cases. First, the algorithm was designed to ensure it is robust against a change in the number of rooms (classes). It works for more or less than 4 rooms both during training and testing phase. Second, it can also work with more or less than 7 attributes. This ensures that the algorithm generalizes well to alternative datasets capturing potential edge-cases.

4 Algorithm Evaluation and Results

The following section outlines the results of our decision tree learning algorithm, presenting the metrics explained above and investigating the affect of pruning. We note that we have a balanced class distribution in the noisy and clean datasets with class 1 represented 490 times (24.5%), class 2 represented 497 times (24.85%), class 3 represented 515 times (25.75%) and class 4 represented 498 times (24.9%).

4.1 Before Pruning Results

First, we present the results for the clean and noisy datasets pre-pruning (Figure 4 and Table 2). When comparing the confusion matrix in Figure 4a and Figure 4b we observe right away that the decision tree learning before pruning performed better on the clean dataset, with less misclassifications, than on the noisy dataset. More precisely, the clean dataset has a 17.6% higher *accuracy* than then noisy dataset.



(a) (b)

Figure 4: Confusion matrix for (a) clean and (b) noisy datasets after cross validation pre-pruning.

	Clean Dataset				Noisy Dataset			
	1	2	3	4	1	2	3	4
Precision	**0.988**	0.957	0.942	0.984	0.788	**0.806**	0.793	0.788
Recall	**0.989**	0.955	0.941	0.982	0.785	**0.813**	0.798	0.778
$F1$ *score*	**0.988**	0.956	0.941	0.983	0.785	**0.808**	0.794	0.781
Classification Rate	0.967				0.794			

Table 1: Results per class after 10-fold cross validation before pruning.

In the clean dataset, we observe that class 1 has the highest *precision*, *recall* and $F1$ *score*, followed by class 4 and both are more likely to be classified correctly than class 2 and 3 which are more likely to be confused with each other. If we compare the best and worst classes by metrics in the clean dataset, we observe that class 1 has a 4.6% higher *precision*, 4.8% higher *recall* and 4.7% higher $F1$ *score* than class 3. The average *precision* indicates that 98.8% of class 1 versus 94.3% of class 3 predicted labels are accurately predicted. And the $F1$ *score* which can be interpreted as the measure of the test accuracy is also high, with class 1 having the highest score, followed by class 4, class 2 and class 3. Interestingly, in the noisy dataset, class 2 and class 3 have the highest *precision*, *recall* and $F1$ *score* and are more likely to be classified correctly than class 1 and class 4 which are more likely to be confused as each other. We observe that class 2 has a 1.8% higher *precision*, 3.5% higher *recall* and 2.7% higher $F1$ *score* than class 4. The confusion matrix shows additionally that the predicted class 1 is the least precise.

4.1.1 Noisy vs Clean Dataset Analysis

There is a significant difference in performance when using the clean and noisy dataset (17.6% higher accuracy for the clean dataset). Results showed additionally that the decision tree performed better on the clean

dataset than on the noisy dataset, since precision, recall, F1 score and classification rate are higher for both datasets as well as by class. See Table 2 for the average scores over all classes. The clean dataset has on average for all classes a 17.5% higher precision, 17.3% higher recall and 17.5% higher F1 score than for the noisy dataset. In addition, we observe that in the clean dataset confusion matrix (Figure 4a) most of the values ended up being correctly classified (along the diagonal) except for on average 2.2 misclassifications of room 3 as room 2 and room 2 as room 3. In contrast, the confusion matrix of the noisy dataset has many more misclassifications, shown as positive numbers on the upper and lower triangle (Figure 4b). This makes sense, because the noise in the data prevents the decision tree from predicting the correct label - the more noise, the harder the prediction is.

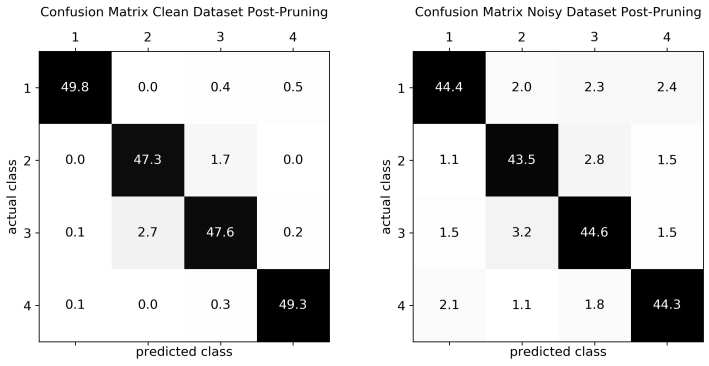
	Clean Dataset	Noisy Dataset
Precision	0.968	0.793
Recall	0.967	0.794
F1 score	0.967	0.792
Accuracy	0.967	0.794

Table 2: Average scores over all classes for the unpruned trees.

Upon further inspection of the datasets, we see that the clean dataset has integer values and is not sampled with any noise, while the noisy dataset has floating point values and has been sampled with noise, indicating that the signals provided are not always representative of their true values. This is a common problem when working with noisy data, because it can have a strong impact on the accuracy of an algorithm. Training can take longer and there is a danger of overfitting to the noisy data. This effect was also seen while building and pruning the tree, as it took significantly longer for the noisy than for the clean dataset for the same amount of training samples. This effect is due to the fact that the algorithm splits the tree in many more partitions due to the noise, resulting in both smaller partitions at the leafs and a greater overall depth of the tree.

4.2 Post-Pruning Results

After pruning the tree we compute the same evaluation metrics as before (Figure 5 and Table 3). We find that performing pruning on our tree significantly improves the performance of our tree on our noisy dataset, while the results from the clean dataset improved only slightly. More precisely, the *accuracy* of the pruned clean data increased by 0.3% while for the noisy dataset it increased from 79.4% before pruning to 88.4% thanks to pruning.



(a) (b)

Figure 5: Confusion matrix for (a) clean and (b) noisy datasets after cross validation post-pruning.

Between the noisy and clean dataset post-pruning, we observe a 9.9% higher *classification rate*. Pruning the tree leads to a larger increase in accuracy for the noisy dataset. The average *precision* between both datasets is higher by 10.1% for the clean dataset, indicating an increase of 11.2% in *precision* for the noisy

	Clean Dataset				Noisy Dataset			
	1	2	3	4	1	2	3	4
Precision	0.983	0.968	0.940	**0.992**	0.870	0.889	0.879	**0.894**
Recall	**0.996**	0.946	0.951	0.985	**0.906**	0.875	0.866	0.887
F1 score	**0.990**	0.956	0.945	0.988	0.887	0.881	0.871	**0.890**
Accuracy	0.970				0.884			

Table 3: Results per class after 10-fold cross validation and after pruning.

dataset pre- and post-pruning. The clean dataset *recall* is 9.8% higher than for the noisy dataset. After pruning, *recall* for the noisy dataset increased from an average of 79.4% before pruning to 88.4% by 11.3% and the *F1 score* saw a similar increase post pruning by 11.4%. In the confusion matrix (Figure 5), we still see a higher misclassification rate for the noisy dataset, with class 3 in the clean dataset more likely to be confused. The average metrics over all classes after pruning can be found in Table 4.

	Clean Dataset	Noisy Dataset
Precision	0.971	0.883
Recall	0.970	0.883
F1 score	0.970	0.883
Accuracy	0.970	0.884

Table 4: Average scores over all classes for the pruned trees.

4.2.1 Post-Pruning Analysis

For convenience, we attached in Appendix B a table comparing the metrics between the pruned and unpruned results. Pruning has significantly increased the performance on our noisy dataset, while for the clean dataset it remained almost constant. During the pruning part of our algorithm, we are iteratively checking whether removing leaves improves the accuracy of the tree on the evaluation set. Before performing the pruning algorithm, the tree has most likely overfitted to the training data. Pruning reduces this overfitting significantly: without pruning, our tree fits the training data very well, but does not generalize well to unseen data. Pruning the tree also significantly reduces the complexity of the tree by making it both slimmer and less deep. For more information on the implementation of the pruning functions, we refer to Section 3.4.

For the clean dataset, comparing the results from pruned and unpruned tree shows a very small overall increase in accuracy (0.3%). However, when looking at the results on a by class level, we observe a small decrease by 0.5% in class one *precision* and a decrease of 1% in class two *recall*. As we can see in general pruning has made some metrics better and some metrics worse for the clean dataset. As there is no noise in the clean dataset, pruning potentially removes valuable data which can lead to worse performance. Depending on how our dataset was shuffled and split, certain samples can erroneously be detected as noisy by the pruning algorithm. However, it is important to note that this reduction does not happen very often. Every time we run the algorithm a new tree is created and our results are slightly different. As we are averaging results of many runs the erroneous pruning in the clean datasets affects our results only little. On the other hand, pruning has a substantial effect on the noisy dataset. Every single one of our evaluation metrics increased for the noisy dataset after pruning by a substantial amount. This suggests that the pruning eliminated a lot of the noise in the dataset - increasing correct classifications and preventing misclassifications. This brings the performance on the noisy dataset much closer to the performance on the clean data. However, the algorithm does not manage to classify the noisy data as well as the clean one.

4.3 Depth Results

The average maximal depth for the clean dataset before pruning is $d = 14$, which is reduced by 29% to $d = 10$ after pruning. For the noisy dataset, the average maximal depth is $d = 22$, which decreased by 32% to $d = 15$ after pruning.

4.3.1 Depth Analysis

Comparing the depth before pruning between both datasets, with a 57% larger depth for the noisy dataset, gives valuable insights to understand why the noisy dataset evaluation has an overall lower accuracy. In general, a very deep tree might lead to overfitting, while a very short tree might experience underfitting. This is the trade off between bias-and variance applied to decision trees. In our case, we used 10-fold cross validation with both training, validation and test set and experienced 9 different depth for each K-fold of the cross validation steps. Since for each fold the data is partitioned differently, we also experienced different depths per fold. The depth of $d = 22$ indicates that the very deep tree for the noisy unpruned dataset partitions the data unnecessarily often, trying to fit the noise as well, leading to overfitting in this case. In comparison, the tree trained on the clean dataset has a significantly lower depth (8 levels lower), which indicates a reduced tendency towards overfitting. By comparing the results before and after pruning, we see that pruning reduces the depth for both datasets. The reduction of 7 levels for the noisy dataset (which reduces the tree depth to a size smaller than the unpruned clean tree) leads to understand how much noise is being damped in the dataset by the pruning algorithm.

From these results, we can infer that the shortening maximal depth is related to the improvement in prediction accuracy due to pruning. In this case, a shorter depth often affords a higher prediction accuracy, because there is less noise. This relationship is not a fundamental rule, it is possible that a smaller depth tree can result in a worse accuracy, because it would be underfitting the data by not splitting enough. But once maximal depth goes beyond a certain threshold, there is a strong potential for overfitting by splitting too much. Overall, we view the reduction in depth of our pruned tree, accompanied by improving evaluation metrics on the noisy data to be an indicator that pruning is eliminating noise, and a smaller depth implies a better classification for the noisy dataset, in this particular case.

4.4 Diagram of the pruned tree

Figure 6 illustrates the visualization of our pruned decision tree. Each vertical line represents a split, and each horizontal line represents the node at the end of the split, which is either the beginning of the left or right branch, or it is a leaf. We only show one tree here to avoid filling this report with unnecessary diagrams. If the reader wants to generate additional trees we refer to the Jupyter Notebook *DecisionTree_Notebook*.

5 Conclusion

This report discussed our implementation of a decision tree algorithm analyzing two datasets of Wi-Fi signals measured from 4 different rooms. We split the tree based on the highest information gain, evaluated our tree on test data and managed to improve our trees with a pruning algorithm. We've shown that our algorithm works nicely for both clean and noisy data, clearly outperforming on the clean data. We've managed to improve our algorithm by adding a pruning function that reduces overfitting to noise in the data. Our implementation is easy to understand and uses a simple Python dictionary class to save the decision tree. We've analyzed the difference between the average depth of the trees before and after pruning. We conclude that our decision tree was successful for both the clean and noisy dataset, with an overall classification accuracy of 97 % on our clean dataset and 88% on the noisy dataset. Furthermore, we saw how pruning had little to no effect on the decision tree for the clean data, yet for the noisy dataset it was able to improve the classification accuracy by 11%, which suggests that pruning was able to eliminate a considerable amount of noise in the tree. Nevertheless pruning made both trees simpler which is generally good practice. Finally, we compared the depth of our trees before and after pruning, and we noticed that for the noisy dataset which was improved considerably by pruning, the maximal depth decreased by 7 layers. Thus we were able to infer that there is correlation between a shortening maximal depth and an improving accuracy through pruning.

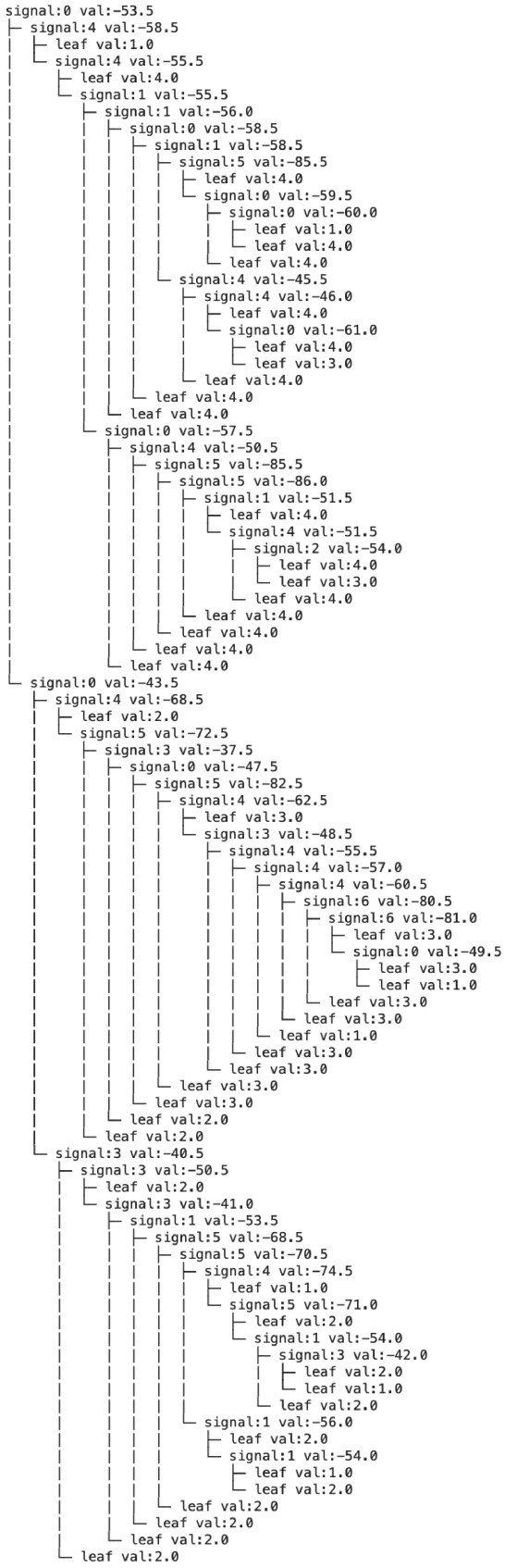


Figure 6: Visualization of a pruned decision tree.

References

[1] Nasma Dasser et al. *IML_DecisionTree*. Version 1.0. Nov. 2020. URL: `https://gitlab.doc.ic.ac.uk/kgl20/iml_decisiontree`.

[2] JR Quinlan. "Induction of Decision Trees. Mach. Learn". In: (1986).

[3] Stuart Russell and Peter Norvig. "Artificial intelligence: a modern approach". In: (2002), pp. 534–545.

Appendix A Evaluation Metrics

In order to evaluate the performance of the classification algorithm on a testing set, we define five metrics. First, the confusion matrix $C_{i,j}$ which represents in each row i instances of the predicted class and each column j instances of the actual class. In our case, a class is a "room" (Table 5). Each entry is an integer representing how many times the tree classified a node as true or false, and if false, which mis-classification it made. For example, we can immediately observe that along the diagonal, we can count the total number of true predictions that our tree made per class.

		predicted class			
		1	2	3	4
actual class	1	true	false	false	false
	2	false	true	false	false
	3	false	false	true	false
	4	false	false	false	true

Table 5: Confusion matrix $C_{i,j}$

The next metrics, follow directly from the $4x4$ confusion matrix. The *accuracy* is defined as the total number of correct predictions divided by the total number of predictions made :

$$accuracy = \frac{\sum_{i=1}^{N} \mathbf{C_{ii}}}{\sum_{i=1}^{N} \sum_{j=1}^{N} \mathbf{C_{ij}}}. \tag{4}$$

Precision is the number of correct positive predictions divided by the total number of positive predictions for each class $r_j, j = 1, 2, 3, 4$:

$$precision(r_j) = \frac{\mathbf{C_{jj}}}{\sum_{i=1}^{N} \mathbf{C_{ij}}}. \tag{5}$$

Recall or Sensitivity is the number of correct positive predictions divided by the total number of positives for each class :

$$recall(r_j) = \frac{\mathbf{C_{jj}}}{\sum_{i=1}^{N} \mathbf{C_{ji}}}. \tag{6}$$

And finally, we compute the $F1$ score, defined as the harmonic mean of the *precision* and *recall* as:

$$F_1(r_j) = \frac{2 \times precision(r_j) \times recall(r_j)}{precision(r_j) + recall(r_j)}. \tag{7}$$

Appendix B Table comparing the pruned and unpruned results

Table 6: Results per Class on both clean and noisy dataset before and after pruning.

	Clean Dataset				Noisy Dataset			
	1	2	3	4	1	2	3	4
Precision								
unpruned	**0.988**	0.957	**0.942**	0.984	0.788	0.806	0.793	0.788
pruned	0.983	**0.968**	0.940	**0.992**	**0.870**	**0.889**	**0.879**	**0.894**
Recall								
unpruned	0.989	**0.955**	0.941	0.982	0.785	0.813	0.798	0.778
pruned	**0.996**	0.946	**0.951**	**0.985**	**0.906**	**0.875**	**0.866**	**0.887**
F1 Score								
unpruned	0.988	**0.956**	0.941	0.983	0.785	0.808	0.794	0.781
pruned	**0.99**	**0.956**	**0.945**	**0.988**	**0.887**	**0.881**	**0.871**	**0.890**
Classification Rate								
unpruned			0.967				0.794	
pruned			**0.970**				**0.884**	