Imperial College
London

COURSEWORK

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

CBC Decision Trees

Authors:

Mehedi Hossain	CID: 01219041
Georgios Ioannides	CID: 01200726
Leszek Nowaczyk	CID: 01253901
Ruoyu Zhang	CID: 01209210

Date: February 11, 2019

Contents

1 Introduction

The purpose of this assignment was to implement a decision tree learning algorithm to predict the room in which a mobile phone was located. The dataset contains 7 Wi-Fi signal strength attributes, and the actual room location of the device. A clean and a noisy dataset were provided.

2 Implementation

2.1 Overview

The given datasets were relatively small in size, so the cross-validation method was used to train and validate the algorithm. The decision tree algorithm involves continuously splitting the dataset to find new nodes. The class Node() was created to represent a node in this tree. Since the attributes have continuous values (non-categorical), this will be a binary tree (value is $<$ or $>=$).

2.2 Generating iterations for Cross Validation

The k-fold cross validation method involves dividing the (shuffled) dataset into **k** equally sized folds, which are then used to create multiple sets of **training**, **validation** and **test** datasets. The function `split_dataset(dataset, k)` produces the above sets, for k-folds. One fold is allocated to each of the validation and test datasets, and **k-2** folds are allocated to the training dataset. For this assignment, k=10.
Firstly, one fold is chosen to be the test dataset (out of 10 possible permutations). From the remaining 9 folds, another one is chosen to be the validation dataset (out of 9 possible permutations). The remaining folds form the training dataset. This gives a total of 90 sets. `split_dataset()` returns these sets in the form of 3 lists, `training_sets, validation_sets, test_sets`, each of length 90.

2.3 Finding the optimal split point

The recursive function `decision_tree_learning()` is used to build a decision tree, returning the root node and the tree's maximal depth. The algorithm uses a `find_split()` function. This function finds the optimal split point, which maximises the information gain, by finding subsets with the lowest combined, weighted entropy (remainder). This optimal split point can exist in any one of the input features, so each feature must be checked. Whilst searching, the maximum reduction in entropy and the corresponding split point have to be tracked. The full pseudo-code for the algorithm used to find the split-point is outlined in Appendix A.

2.4 Training multiple trees in parallel

In order to speed up the training of 90 trees per dataset, the procedure was multi-threaded. The function `cross_validation_train()` is designed as a scheduling process, making use of a Python dictionary, from a "Manager" object from the "multiprocessing" library. Each thread uses one training set to build a decision tree by passing it to the `build_tree` function.

2.5 Computing average results

The constructor for the `Eval_Metrics()` class takes a 4x4 matrix as input, and computes the recall, precision and F1 measure for each class, as well as the tree's classification rate. Each tree is tested against it's associated test set to fill a confusion matrix, to then pass into an `Eval_Metrics()` object.
Whilst doing this for all trees, a list of all the metrics including the matrix entries are collected into `numpy` arrays. The average confusion matrix is computed using the `numpy.average` function, and is then used to instantiate a new `Eval_Metrics()` object, `avg_metrics`. A copy of this is made, and is used to store the standard deviations calculated using `numpy.std`.

2.6 Visualising the tree

The trees are visualized using Matplotlib Pyplot where each node is recursively plotted on an axis showing the label and the value that it was split on. The correct position vertically of each node is determined by the depth of that node in the tree and horizontally by the relative position to the parent node and depth in the tree. Appendix B contains a few example visualisations (Figures 4-7). The `visualise_tree(tree, file_name)` function takes a root node of a tree and the name of the file that the visualisation will be saved to.

3 Evaluation

3.1 Average Performance Metrics

3.1.1 Confusion Matrix

Columns are the Predicted classes and rows are the Actual classes.

	Predicted				
		Room 1	Room 2	Room 3	Room 4
Actual	Room 1	49.178	0.000	0.411	0.411
	Room 2	0.000	48.189	1.811	0.000
	Room 3	0.156	2.478	46.922	0.444
	Room 4	0.333	0.000	0.456	49.211

Table 1: Average Clean dataset

	Predicted				
		Room 1	Room 2	Room 3	Room 4
Actual	Room 1	38.911	2.933	2.944	4.211
	Room 2	2.933	40.022	4.111	2.633
	Room 3	2.733	4.078	40.967	3.722
	Room 4	4.044	2.833	3.067	39.856

Table 2: Average Noisy dataset

3.1.2 Derived Metrics

s.d. = standard deviation
Average classification rate for the clean dataset: 96.75%
s.d. for the average classification rate for the clean dataset: 1.29%

Average classification rate for the noisy dataset: 79.88%
s.d. for the average classification rate for the noisy dataset: 2.88%

	Room 1	Room 2	Room 3	Room 4
Avg	98.36%	96.38%	93.84%	98.42%
s.d.	1.52%	2.41%	3.58%	1.72%

Table 3: Recall rate per class for Clean Dataset

	Room 1	Room 2	Room 3	Room 4
Avg	99.02%	95.11%	94.60%	98.29%
s.d.	1.06%	2.98%	2.51%	2.09%

Table 4: Precision rate per class for Clean Dataset

	Room 1	Room 2	Room 3	Room 4
Avg	0.9868	0.9574	0.9422	0.9836%
s.d.	0.0097	0.0197	0.0234	0.0120

Table 5: F1 measure per class for Clean Dataset

	Room 1	Room 2	Room 3	Room 4
Avg	79.41%	80.53%	79.55%	80.03%
s.d.	6.21%	6.25%	7.50%	6.51%

Table 6: Recall rate per class for Noisy Dataset

	Room 1	Room 2	Room 3	Room 4
Avg	80.03%	80.26%	80.19%	79.04%
s.d.	5.57%	7.14%	5.73%	7.49%

Table 7: Precision rate per class for Noisy Dataset

	Room 1	Room 2	Room 3	Room 4
Avg	0.7972	0.8039	0.7987	0.7953
s.d.	0.0378	0.0502	0.0479	0.0503

Table 8: F1 measure per class for Noisy Dataset

3.2 Interpretation of Average Metrics

Normalization of the confusion matrix changes the values in the matrix from an absolute scale into a notionally common scale through adjustments of values. This allows the data to align on the same scale ($0 \leq x \leq 1$, where x is the matrix entry), making it easier to observe trends and identify any imbalances between classes.

	Predicted				
		Room 1	Room 2	Room 3	Room 4
Actual	Room 1	0.984	0.000	0.008	0.008
	Room 2	0.000	0.964	0.036	0.000
	Room 3	0.003	0.050	0.938	0.009
	Room 4	0.007	0.000	0.009	0.984

Table 9: Normalised clean dataset before pruning confusion matrix

	Predicted				
		Room 1	Room 2	Room 3	Room 4
Actual	Room 1	0.794	0.060	0.060	0.086
	Room 2	0.059	0.805	0.083	0.053
	Room 3	0.053	0.079	0.795	0.072
	Room 4	0.081	0.057	0.062	0.800

Table 10: Normalised noisy dataset before pruning confusion matrix

The diagonal elements of the matrix show the respective recall rates for each class, 1 to 4. The clean dataset's normalised confusion matrix shows that room 3 is the largest contributor to the lower recall rate for room 2 and vice-versa. For the noisy dataset, the matrix shows that the false negatives for each class, are equally spread across the other classes, unlike the clean dataset.

Recall Rate:

Recall Rate = $\frac{TP}{TP+FN}$

The recall rate is the probability of a correctly classified example given it is a positive example. Referencing tables 3 and 6, in section 3.1.2, the recall rate of every class is higher in the clean dataset than their respective counterparts in the noisy dataset. A high recall rate shows that a class is being correctly classified (because of the small number of False Negatives). Looking into the clean dataset's results, room 3's standard deviation is significantly higher and the average is significantly lower. This could be explained by there being a WiFi access point next to the wall between rooms 2 and 3, which makes it hard to determine whether the input should be classified as room 2 or room 3; this is supported by the lower performance for both these rooms.

Precision Rate:
Precision Rate $= \frac{TP}{TP+FP}$
The precision rate is the probability of a positive example given that the example is classified as positive. Precision would be the best performance metric when the application of interest has no high cost associated with false positives. Referencing tables 4 and 7, in section 3.1.2, the precision rate of every class is higher in the clean dataset than their respective counterparts in the noisy data set. A high precision rate shows that an example labeled as positive is indeed positive (small number of False Positives), giving confidence in the prediction made. In the clean dataset, confidence in the predictions made for rooms 1 and 4 are greater than that for rooms 2 and 3, due to the same reason for their lower recall rates. For the noisy dataset, the precision rates per class were inconsistent due to high standard deviations.

F1-measure:
$F_1 = 2 * \frac{Precision * Recall}{Precision + Recall}$
The F1 measure is a performance metric that assesses the performance of the algorithm by combining both the recall rate and the precision rate. Referencing tables 5 and 8 from section 3.1.2, the conclusions drawn for the recall rates and precision rates above apply here as well, as the F1-measure gives equal weighting to both recall and precision. In general, an F-measure score with $\alpha > 1$ would have given much more importance to the precision value. However, as the precision and recall rates were very similar for each class, the F-measure score would remain about the same.

Classification Rate:
Classification Rate $= \frac{TP+TN}{TP+TN+FP+FN}$
This gives an overview through the proportion of correctly classified examples out of total number of examples. Referencing 3.1.2, the average classification rate for the clean data set is much higher than in the noisy data set with a lower standard deviation as well. This observation is expected from the analysis of the previous sections where the clean data set measure has a higher average than the noisy data set average in every measure from recall to F1.

4 Answers to Part 2 Questions

4.1 Noisy-Clean Datasets Question

There is a considerable decrease in classification rate of the noisy dataset when compared with the one of the clean dataset. Noise originates from each WiFi signal strength that may have deviated from the true value. Let us define a *Signature* as the 7-tuple of WiFi signal strength (the feature space), associated with a room label. Noisy signatures paired with label A, can be of a similar profile to the expected signature of label B. This can lead the learning algorithm to predict label A when it should have predicted label B .
In decision trees, complexity refers to the total number of nodes in the tree. As the complexity in a decision tree increases, the model is able to classify the training data better. However, classification rate on the test set decreases because the model becomes specialized for classifying the noise in the training dataset. In the clean dataset, the decision tree learning algorithm generalises well, as the approximated target function selected from the hypothesis space is of appropriate complexity. Therefore the classification rate on the clean test set is higher than in the noisy test set.
Trees trained using the clean dataset had consistently better average recall, precision and F1-measure scores for rooms 1 and 4, over rooms 2 and 3, since they have a lower signature similarity. The noisy dataset had consistently worse recall, precision and F1-measure scores per room. However the average scores across all rooms was approximately equal. This is due the tree's learnt signature per room to deviate from the expected signature, as a result of the noise present in the training dataset. Therefore, the overall performance for each class deteriorates.

4.2 Pruning Question

4.2.1 Implementation

To prune a tree, the root node is passed into `prune_tree(val_dataset, root)`. This function acts as a wrapper for the `prune` function by continuously calling it until no nodes are pruned on the most recent call. `prune(val_dataset, root, parent, curr_node, left, count)` will recursively traverse the binary tree, finding nodes with both children as leaf nodes. When a node that satisfies this condition is found, the parent node is replaced with its children, one after the other, measuring the new classification rate against the validation test. If the classification rate is not negatively affected, the parent node is pruned - replaced by the leaf child, as the parent was not clearly relevant. However, if there was a decrease in the accuracy, the parent node is restored.
A more efficient method would have been to start from the left-most leaf node and traverse the tree from left to right. By doing this, the pruning algorithm avoids having to traverse the tree multiple times, as after pruning a node, the algorithm traverses up to the parent node, and then checks if the parent can be pruned. The more basic algorithm implemented does not check for this condition, and so the whole tree is traversed top-down again until no more nodes are pruned.

4.2.2 Influence on accuracy

	Pre-prune	Post-Prune
Avg	96.75	96.32
s.d.	1.29	1.22

Table 11: Accuracy for Clean dataset

	Pre-prune	Post-Prune
Avg	79.88	87.38
s.d.	2.88	2.75

Table 12: Accuracy for Noisy dataset

Table 11 shows that there was minimal decrease in average accuracy after pruning for the clean dataset. This could be due to nodes that were generally good to have for the test set, being removed by the pruning procedure to improve performance on the validation set. Since the training set was eight times the size of the validation set, it can be seen then that the model that was best for the larger training set, should have better accuracy on the test set.
However, for the noisy data in table 12, a significant improvement on the test dataset was observed (7.5%). Classication rate on the test set is initially low as the model became specialized for classifying the noise in the training dataset. The approximated target function selected from the hypothesis space was too complex, and so was overfitted. Pruning reduced the complexity, by removing the deepest leaf nodes in the tree. By exposing the model to the unseen validation dataset, overfitted branches of the trees were discovered and removed - leading to the increased classification rate on the disjoint test dataset.
Given the overall classification rate for the clean dataset remained the same after pruning, it was not surprising to see that the per class metrics also stayed the same. As for the noisy dataset, the high increase in accuracy after pruning meant that the per class metrics were more interesting. Tables 14 and 15 show that room 3 had the lowest recall and precision rate out of all the rooms. Table 13 shows that the

majority of the confusion for room 3 comes from room 2 and vice versa. In addition, the majority of the confusion for room 1 comes from room 4 and vice versa. This observation is similar to that found for the clean dataset (before and after pruning). Suggesting that the pruning algorithm had successfully reduced the effect of noise on the model, as the per room characteristics of the clean dataset appeared when re-evaluating the noisy trees with the test sets.

		Predicted			
		Room 1	Room 2	Room 3	Room 4
Actual	Room 1	0.897	0.022	0.034	0.046
	Room 2	0.039	0.870	0.065	0.027
	Room 3	0.045	0.065	0.846	0.044
	Room 4	0.044	0.030	0.043	0.883

Table 13: Normalised noisy dataset after pruning confusion matrix

	Room 1	Room 2	Room 3	Room 4
Avg	89.73%	89.70%	84.62%	88.33%
s.d.	4.00%	5.53%	8.44%	5.54%

Table 14: Recall rate per class for Noisy Dataset after pruning

	Room 1	Room 2	Room 3	Room 4
Avg	87.24%	87.91%	86.14%	88.25%
s.d.	5.19%	5.59%	4.40%	6.44%

Table 15: Precision rate per class for Noisy Dataset after pruning

	Room 1	Room 2	Room 3	Room 4
Avg	0.885	0.874	0.854	0.883
s.d.	0.027	0.042	0.049	0.050

Table 16: F1 measure per class for Noisy Dataset after pruning

4.3 Depth Question

	Pre-prune	Post-Prune
Max	14	11
Avg	10.93	5.96
s.d.	1.20	1.89

Table 17: Heights for Clean dataset

	Pre-prune	Post-Prune
Max	22	16
Avg	16.18	7.97
s.d.	1.64	3.04

Table 18: Heights for Noisy dataset

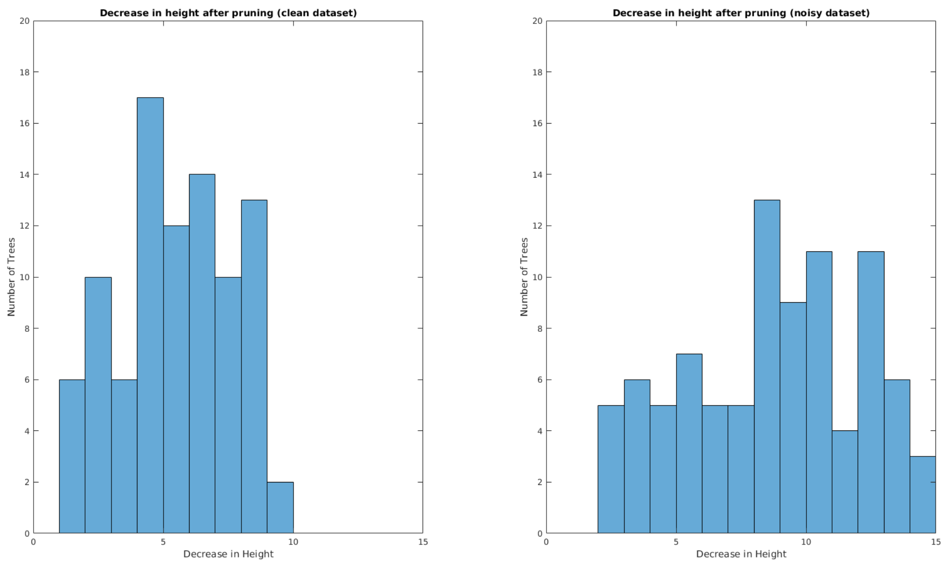

Figure 1: Change in height after pruning

The maximal depth (height) recorded across all trees after pruning had decreased by 3 for the clean dataset and by 6 for the noisy dataset. The exact amount slightly varied run-to-run, depending on the random shuffle of the datasets. After pruning, the average height for the noisy dataset greatly decreased, reducing the difference between average height for the two datasets, and showing that a decrease in height had a much larger impact on the noisy trees' accuracy on the test set. Figure 1 shows that for the noisy trees, the decrease in average height did **not** originate from only a few trees having unusually high decreases in height, showing that all of the noisy trees were too deep. The standard deviation of decrease in height, increased due to a balanced amount of trees having either a large or small decrease in height. The large decreases for certain trees were most likely caused by those trees having deep branches only classifying noise. Figures 6 and 7 in Appendix B display how removing these deep branches can significantly decrease the height, leading to the increase in prediction accuracy shown in Table 12.
Figure 3, also confirms that for the noisy trees, there was a high total number of nodes being pruned per tree, due to either a few deep branches extremely affected by noise, or many branches being equally affected by noise. Tree height determines how

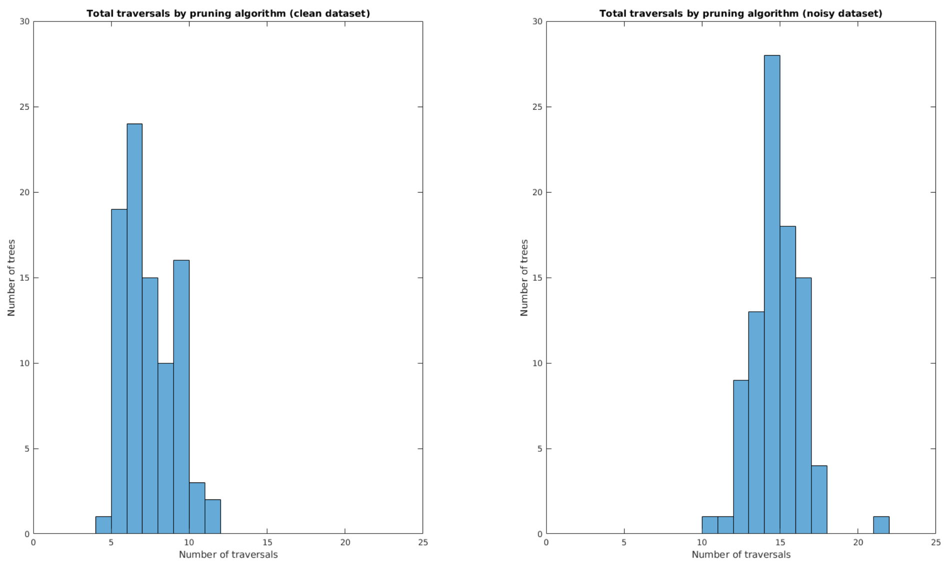

Figure 2: Number of traversals during pruning

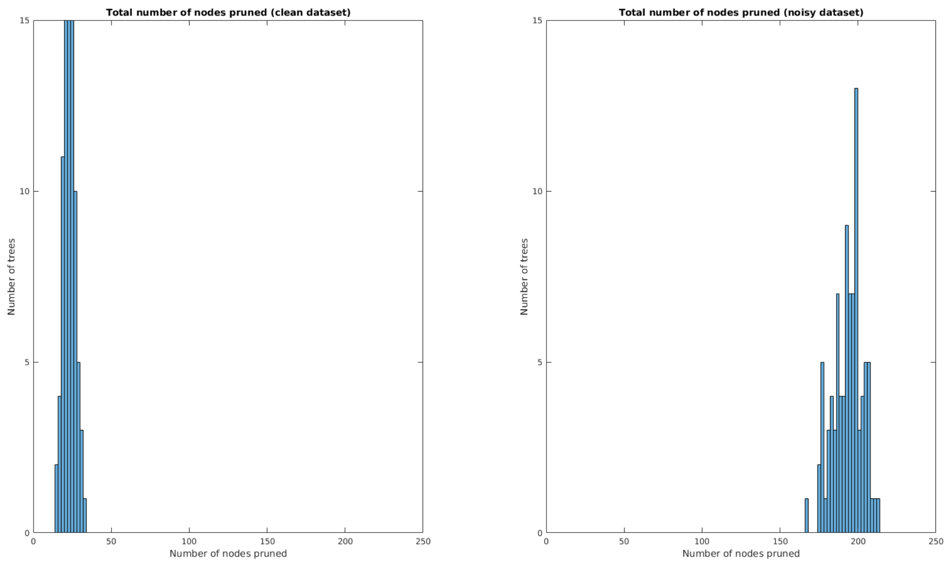

Figure 3: Total number of nodes pruned during pruning

complex the model is. The taller the tree, the more complicated the target function it can fit. Therefore, by increasing the height of the tree it should also increase the performance on the *training* set used. However, by increasing the complexity of the tree through increasing its height, it will also produce a greater tendency to overfitting. If this occurs, the prediction accuracy on the test set may decrease.

Appendices

A Find Split Algorithm

```
def find_split(training_dataset):
        min_remaining_entropy = entropy in training_dataset
        # keep track of split point that produced the max_reduction
        best_split = None

        for each feature:
                sort dataset by the current feature
                for each example:
                        if current_outcome != prev_outcome
                                # Potential split point
                                entropy_reduction = calc_remaining_entropy(this_split_point)
                                if entropy_reduction < min_remaining_entropy
                                        min_remaining_entropy = entropy_reduction
                                        update best_split

        return best_split
```

$$Information\ Gain = H(dataset) - Remainder(S_{left}, S_{right})$$
$$H(dataset) = \sum_{k=1}^{k=K} p_k * log_2(p_k)$$
$$Remainder(S_{left}, S_{right}) = \frac{|S_{left}|}{|S_{left}|+|S_{right}|}H(S_{left}) + \frac{|S_{right}|}{|S_{left}|+|S_{right}|}H(S_{right})$$

B Visualisation Figures

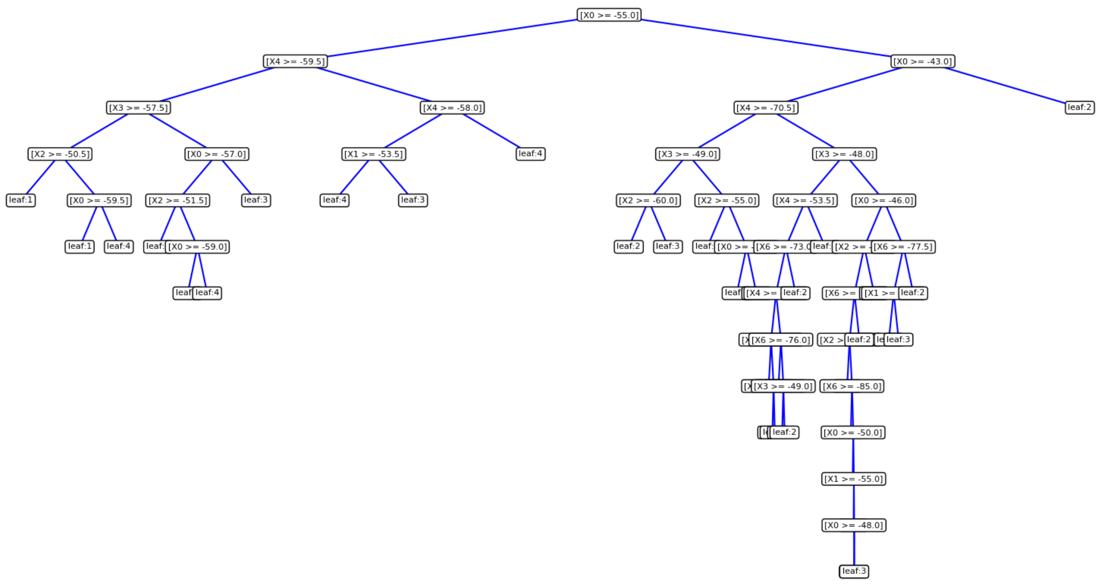

Figure 4: Pre-pruned decision Tree using clean dataset

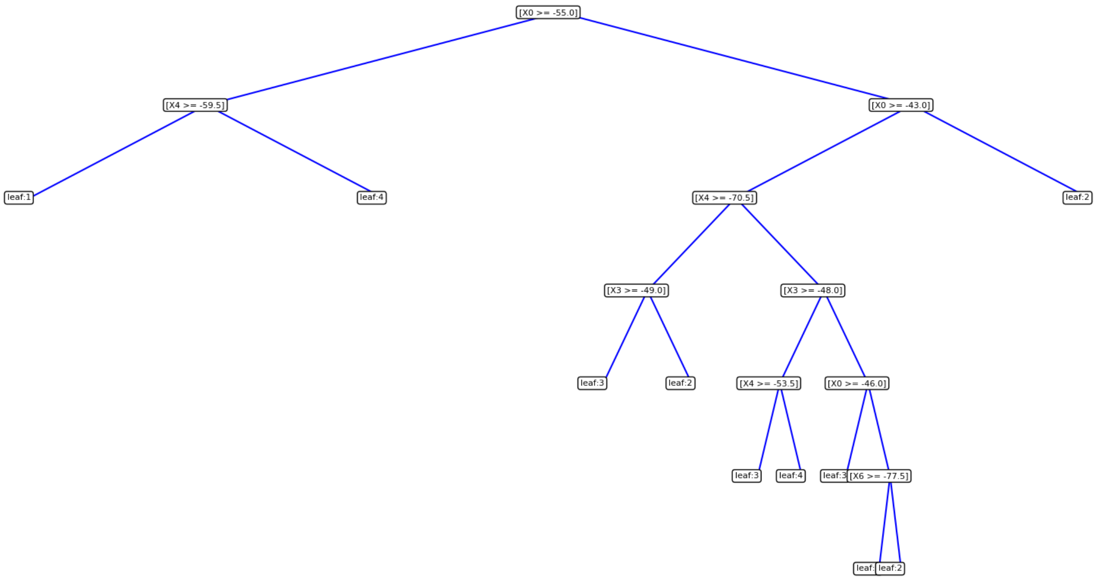

Figure 5: Pruned decision Tree using clean dataset

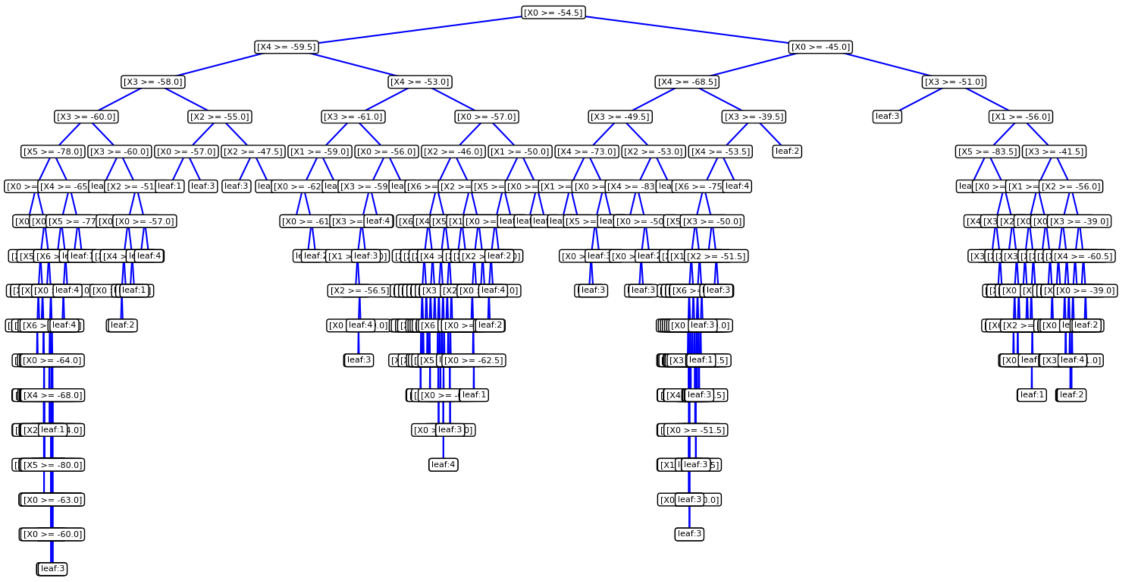

Figure 6: Pre-pruned decision Tree using noisy dataset

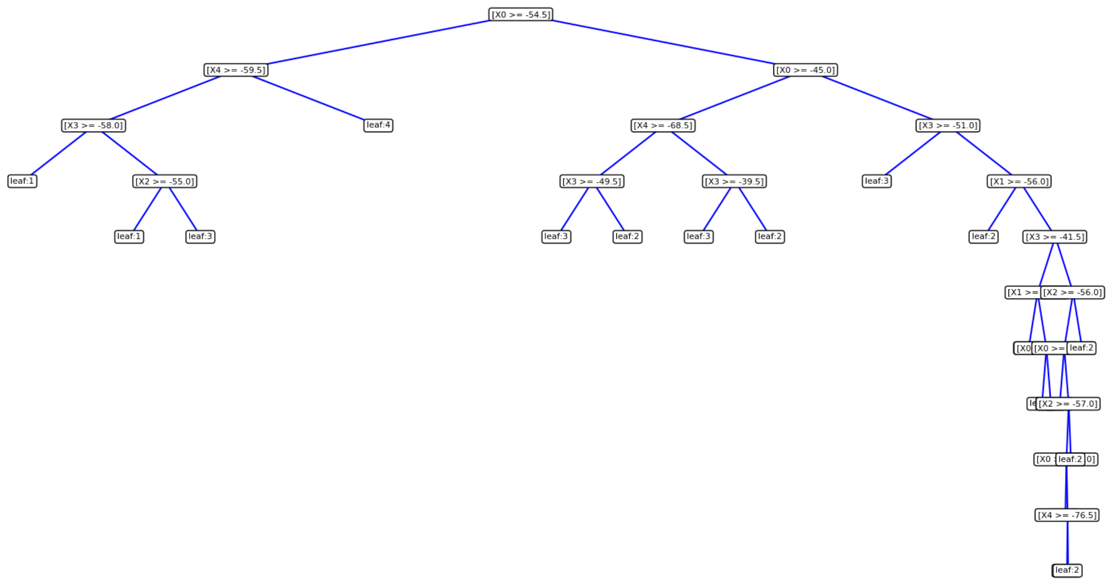

Figure 7: Pruned decision Tree using noisy dataset