C395
ML Report 1
Decision Trees
Group 68

Enda Mulville: em1716
Alessandro Serena: as6316
Zed Al-Aqqad: za716
David Rovick Arrojo: der115

Contents

1 Introduction

In this assignment, we were asked to implement a decision tree algorithm and use it to determine the location of a phone based on WIFI signal strengths collected from 7 different routers by that mobile phone. The data that the mobile phone provides is the strength of the 7 different WIFI signals.

Our decision tree will consist of a series of nodes, each node representing a single binary decision on one of the attributes (in this case if the strength of a WIFI signal is greater or lower than a certain value). Based on the result of that decision either another node is reached and another decision is taken or the data arrives at a leaf. Once the data reaches a leaf then the value at that leaf is taken to be the room that the decision tree has predicted the phone to be in.

2 Summary of Implementation Details

Our implementation is based on the basic decision tree algorithm used in lectures, this includes the gain function used to find the most optimal split attribute, pruning and cross validation as well as analysis of the confusion matrix to evaluate our results.

2.1 Splitting The Tree

To split the tree, we need to decide which attribute (column) as well as what value in that attribute we split the tree on. To do that, we have a function called `best_split`. This function uses a nested for loop to calculate the information gain on every possible split, updating a `'best_split'` (initialized at 0) data structure with information on the most optimal split found so far. The outer for loop loops on every column (this is looping over the attributes i.e. each wifi signal) in order from the 1st to the 7th. For each attribute, the function orders the training set in a descending order and looks for when the label changes, then it calculates the information gain we would achieve if this split happened at that value on that attribute. Because a split only represents a simple decision, essentially asking if an attribute is greater than or less than a value (with that value found by comparing information gains on potential splits as explained later in this section), we can meet the performance requirements of the algorithm while still checking all feasible splits in a reasonable length of time.

In order to evaluate each potential split that is found, the function carries out this split by considering each row and appending it onto either the `left_set` or `right_set` subsets based on the `value` and `attribute` of that split. Once the split has taken place, the `information_gain` from this split is calculated, this is based on the information gain formula seen here:

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

$$H(dataset) = -\sum_{k=1}^{k=K} p_k * \log_2(p_k)$$

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

Where $|S|$ represents the number of samples in subset S.

Whenever a split has a larger gain than the previous best value, it is set as the "best split" and after considering all possible splits, it the function implements the most optimal split. Finally the two new sets are passed recursively into the function again in order to calculate the decisions of the new left and right nodes.

A particular branch terminates and builds a leaf when the all of the data in the set that is passed into the function all has the same label, i.e. once the remaining data entries to be considered in the descendingly ordered data set all have the same label. The node is then set to a leaf with the value of the label of the data that was passed to it. This is simply done by updating the value of `leaf` in the node to the label it predicts, otherwise a `leaf` value of 0 represents that the node is not a leaf and requires further evaluation.

2.2 Evaluation

In order to evaluate the performance of our tree we created a function `evaluate` that takes as its arguments a tree and a data set that will be used to test that tree.

Firstly, we feed all of the data set into the tree, recording both the tree's predicted tag and the actual tag (ground truth) for that data point. From this data we can construct a confusion matrix to record all the combinations of ground truth and prediction. The confusion matrix records the predicted vs actual classifications of each data point, telling us the number of correct predictions, or 'hits', and incorrect predictions, or 'misses', for each possible classification. As each run of our implemented training, cross validation and testing algorithm creates a large number of decision trees and thus a large number of confusion matrices, the confusion matrices displayed in this report contain averages of all the corresponding entries for all the created trees.

While the confusion matrix is a good visual indicator of performance, having derived metrics to represent the performance has its benefits. That is why the function not only returns the confusion matrix, but also four other values. These are the `recall`, `precision`, `F-score` and `classification rate`.

The `classification rate`, also known as the `accuracy`, is the ratio of correctly classified data points with respect to the total number of data points. Note that if the spread of classes in the actual data set was not roughly uniform, then the classification rate would be flawed. To understand this, consider a data set where 90% of data points belong to one class. The classification rate in this case would only be giving us information about the success of predicting that one class, and would not tell us anything about the success of the others. Thankfully, both our clean and noisy data sets are uniformly distributed, meaning that the `classification rate` can be used as an informative and reliable performance measure without having to perform any manipulation, or "balancing", on the test set.

For `recall` and `precision`, it is necessary to first note that these are metrics that are associated with and calculated for each separate class in the confusion matrix, and that the figures we report are unweighted mean averages across the derived metrics for each class across every decision tree built. Thus we can explain that the `recall` is the number of correctly classified data points of a given class divided by the total number of actual data points for that class, shown here as a percentage.
Similarly, the `precision` is simply the number of correctly classified data points of a given class divided by the total number of data points predicted to be a member that class.

The `F-value` or `F-score` is calculated as

$$F_1 = 2 * \frac{precision * recall}{precision + recall}$$

.

Shown in this report as a percentage, this is essentially the harmonic average of the precision and recall. Here, a score of 100% indicates perfect precision and recall, with worse performance indicated by a lower percentage.

2.3 Pruning

Regarding our pruning implementation, we have created a function called `prune` that takes as parameters a decision tree and a validation set. Once called, the function calls `recursive_prune`, which recursively traverses the tree until it reaches a node where both children are leaf nodes. If both children of the node are not leaves, the function will call `recursive` prune on those children.

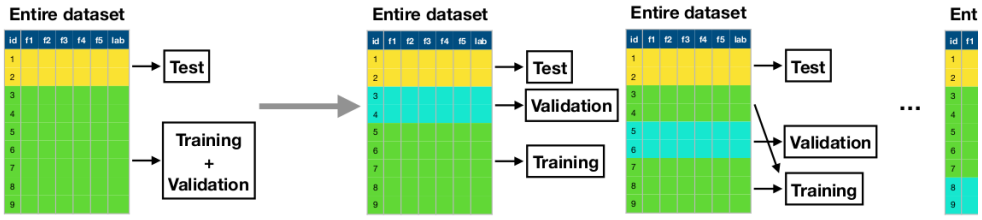
If `current_node` is a node where both its children are leaves, we evaluate the performance of the current tree using the `evaluate` function and the validation set and use these values as a benchmark for any changes. Then we replace the current node with its left child and then its right child, evaluating the performance of the tree for each of the three variations of the tree using the same `evaluate` function and the validation set. The function then compares the performance of the three options and keeps the best performing one, either turning `current_node` into a leaf or reverting any changes made.

As the recursion is done before checking if both children are leaves, the function can deal with the case when a node is pruned, which results in its parent now being a candidate for pruning. In this case the function will check for any improvements as if the node had had two leaf children all along.

2.4 Validation and Testing

In order to test the tree we implemented a cross validation function `crossValidate` taking the target data set as a single parameter. The cross validation is done by splitting the data set into testing, validation, and training sets, using a respective 10%/10%/80% split. For each 10th of the data set chosen for testing, both building and evaluation of a new decision tree are performed on each of the nine possible spits between training and validation sets. Our code uses what is essentially a 10 by 9 nested for loop to build and evaluate each of the 90 possible trees we can construct using the above ratio. Splitting out the testing data set is performed easily by `numpy.split`, then `numpy.concatenate` gathers the remaining data together so the inner loop can cycle through and select subsequent splits between training and validation data with each iteration.

During each inner loop iteration the function extracts the separate `validation` and `training` data ready for use building and pruning the tree. Firstly an initial tree is build using the training data. The `evaluate` function is then called on this tree, this evaluation uses the test data set and the resulting metrics are stored in the `unpruned_results` data structure. The `prune` function is then called using the validation set in order to prune the tree, and the results of `evaluate` using the same test data on the pruned tree is stored in `pruned_results`. The final part of each inner nested for loop iteration simply calculates what percentage of the way through execution the for loop is currently at, as well as the time elapsed since the beginning of the outer for loop. This information does not affect any execution of the algorithm, and is printed neatly on the screen purely for the benefit of the user.



Lastly, once all 90 trees have been build, pruned and tested the final step is to take the average results from all of the tests. This is done by calling `average_metrics` that takes in a 10 by 9 matrix of metrics and returns a single average set of metrics. Both average metrics from before and after pruning are then printed so that the user can clearly see the results of testing on the data set.

3 Evaluation of Results

Our main method of evaluating our results is to test the performance of our generated trees under numerous different circumstances. As stated earlier, a confusion matrix is generated for each tree, of which the following metrics are generated: Recall, Precision, F score and Classification rate.

3.1 Performance: Clean vs Noisy Data

The initial test is on how the quality of the data affects the performance of the tree. To do this we have created decision trees under the following four different cases (shown with their corresponding confusion matrix [1]):

1. Tree trained on clean data, tested using clean data.

	$P1$	$P2$	$P3$	$P4$
$T1$	40	0	0	0
$T2$	0	52	1	0
$T3$	0	2	44	1
$T4$	0	0	0	60

2. Tree trained on clean data, tested using noisy data.

	$P1$	$P2$	$P3$	$P4$
$T1$	34	2	2	2
$T2$	1	46	1	0
$T3$	1	1	48	3
$T4$	1	0	1	56

3. Tree trained on noisy data, tested using clean data.

	$P1$	$P2$	$P3$	$P4$
$T1$	39	0	0	1
$T2$	2	48	0	3
$T3$	1	2	43	1
$T4$	1	2	4	53

4. Tree trained on noisy data, tested using noisy data.

	$P1$	$P2$	$P3$	$P4$
$T1$	26	3	3	8
$T2$	4	37	4	3
$T3$	1	3	44	5
$T4$	5	5	7	42

This ensures that we know the exact effect of the quality of the data.
Having seen the confusion matrix of each case, the four metrics are calculated, they are summarized in the following table:

Case	Recall(%)	Precision(%)	F Score(%)	Classification rate(%)
1	97.933	98.109	98.021	98.000
2	91.579	91.866	91.722	92.000
3	91.972	91.469	91.720	91.500
4	74.072	74.395	74.233	74.500

[1] All confusion matrices have the "Predicted" classes labels on the x-axis and the "True" classes labels on the y-axis

As shown by the results, the best performance was that of case 1, this is expected as both the training data and the test data are clean, giving the tree the best chance of correct classification. Also as expected, the worse performance was that of case 4 (about 20% degradation) as both training and test data sets are noisy, making correct classification difficult due to it's proneness to errors.

A surprising aspect of these results is the relatively good performances of cases 2 and 3; they performed much better than case 4 and they were also quite similar to each other. This at first seemed odd given that both were dealing with a noisy data set (either the training or test data). However, after giving it more thought it made sense; since only half of the data sets are noisy at a time, in most cases the trees will still manage to correctly classify the data, as opposed to both data sets being noisy.

3.2 Performance: non-pruning vs pruning

As you can see from these results:

Results before pruning:

Confusion Matrix:

	$P1$	$P2$	$P3$	$P4$
$T1$	49.1889	0	0.4333	0.3778
$T2$	0	47.8444	2.1556	0
$T3$	0.2444	2.1222	47.3	0.3333
$T4$	0.4111	0	0.2778	49.3111

Recall: 96.7375 %
Precision: 96.8657 %
F Score: 96.8015 %
Classification Rate: 96.8222 %

Results after pruning:

Confusion Matrix:

	$P1$	$P2$	$P3$	$P4$
$T1$	49.1889	0	0.4556	0.3556
$T2$	0	47.8667	2.1333	0
$T3$	0.2333	2.0556	47.4	0.3111
$T4$	0.4111	0	0.2778	49.3111

Recall: 96.7919 %
Precision: 96.9358 %
F Score: 96.8637 %
Classification Rate: 96.8833 %

The pruning has only minimally affected the performance of our decision-tree on the clean data set. This is reasonable because the clean data set has no noise at all, thus every value used to build the tree is correct and should affect our prediction. Pruning can only remove decision from the tree and as all of the decisions on the clean data set should be correct then any improvement gains on the validation set will be balanced out in the test set by the loss of that decision.
However when applying pruning to the noisy data set you can see that it provides significant performance:

Results before pruning:

Confusion Matrix:

	$P1$	$P2$	$P3$	$P4$
$T1$	37.8889	3.6111	3.3778	4.1222
$T2$	2.9667	39.4333	4.4	2.9
$T3$	3.0444	3.8222	41.2333	3.4
$T4$	4.3111	2.3333	3.35568	39.8

Recall: 79.0976 %
Precision: 79.3213 %
F Score: 79.2083 %
Classification Rate: 79.1778 %

Results after pruning:

Confusion Matrix:

	$P1$	$P2$	$P3$	$P4$
$T1$	38.4889	3.2	3.3	4.0111
$T2$	2.9111	40.1111	4.0333	2.6444
$T3$	2.9667	3.5667	41.8667	3.1
$T4$	3.9667	2.1889	3.2	40.4444

Recall: 80.3527 %
Precision: 80.5946 %
F Score: 80.4725 %
Classification Rate: 80.4556 %

This is because there are outlying values in the noisy data set that do not represent the general trend of the data. Thus if the decisions that came from these values are pruned then there is no loss to the performance of the tree on the rest of the data.

3.3 Depth of Tree

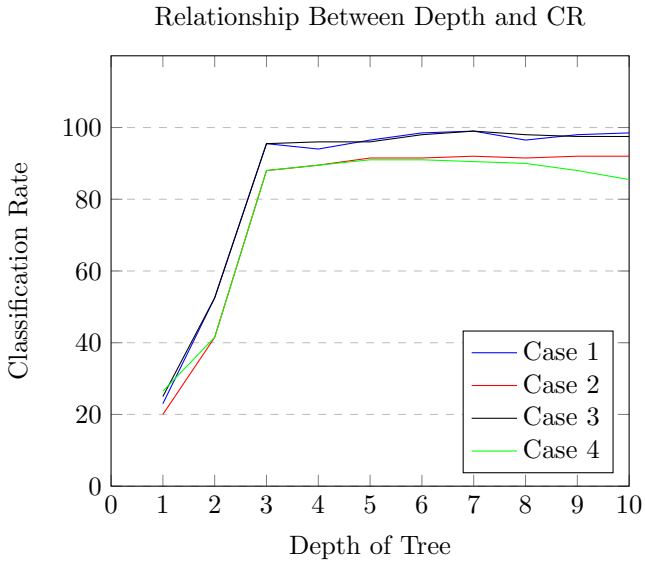
3.3.1 Before vs After pruning

The depth of our trees before pruning have an average of 12 layers for the clean data set and 16 layers for the tree built from the noisy data set. This difference is reasonable as the decisions that result from the noise will add a number of extra layers to the trees.

Initially we expected to see that the maximum depth would reduce significantly after pruning, however the results showed that there was minimal reduction in the depth of our trees after pruning, many times no change at all. After looking at the visualization of the trees (as seen in part 4) we could see that although the trees were smaller, but there was usually one or two longer branches that didn't get pruned that were keeping the depth of the tree high.

3.3.2 Relationship with Prediction Accuracy

Next, we investigated the relationship between tree depth and prediction accuracy. To effectively do that, we altered the code as to limit the depth of the tree to a certain value. Then we ran it for depths 1 to 10 and recorded the classification rate (CR) for all 4 cases defined in section 3.1. The reason for using CR rather than any other metric is simply because it gave a valid value even for the trees with the lowest depths (ex. depth 1 and 2). The graph below summarizes our findings:



As illustrated above, there is a clear relationship between depth and CR, hence prediction accuracy. The data suggests that initially, CR is significantly increased as depth is increased (ex. 20 % at depth 1 vs 89.5% at depth 4 for case 2). However after a certain depth (in this case 4) the improvement factor is reduced to only a minor increase in CR. This would also suggest that the underlying pattern can be reasonably expressed at a certain depth (in this case 4), and any increase in depth after that will show little to no improvement.

4 Diagram of Trees (Bonus)

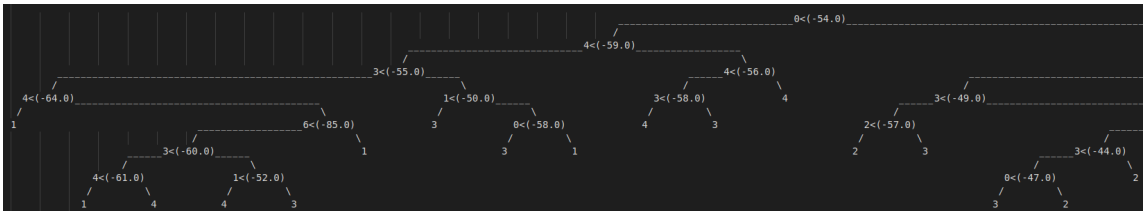
Given that the nodes of the trees that we have generated are represented as a dictionary, visualizing the tree by using the built-in Python function `print(rootNode)` can be a hard task, especially when the tree depth increases.

To solve this issue, we created the function `visualize(tree)` to generate a multiline string representing the tree that can be then printed to std:out or to a file.

The printable strings are generated by recursively getting the printable representations of the two subtrees and by joining them together with the representations of the root node.

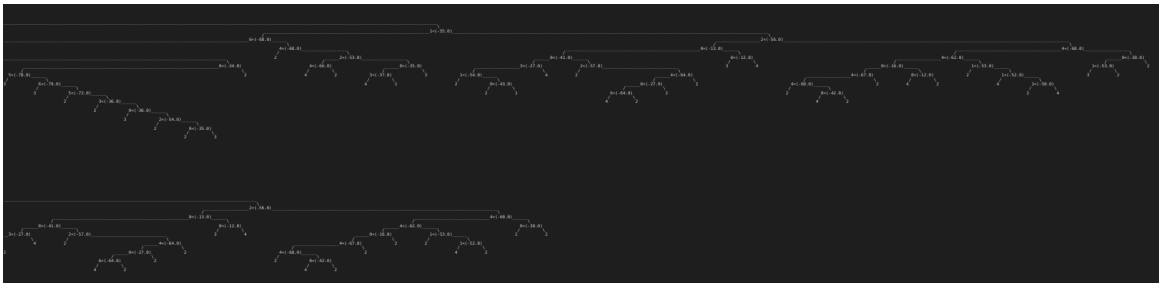
Recursively, the algorithm will get to a leaf node; this will only be represented by its value. Backtracking one step, a root node will be encountered and it will be represented by its decision value in a string format (i.e. `parameter < gain_value`). The join of the two leaves and the root node is performed by analyzing the width of the left and right elements (in this case they are only two single values, but more generally it is going to be a "box") and by connecting them to the root node by inserting dashes (`-`) and slashes (\ and `/`).
The box obtained above is then returned, together with its width and vertical dimensions, for the recursion to continue the backtracking.

Below is an example of part of the output of the visualizer when run on `cleanTree`.



The string generated is generally quite wide (in the example above the tree continues to the right) and most terminals would wrap the string across multiple lines, making the data printed pretty much useless. In order to have an understandable vision of the trees we suggest to print the output to a file and scroll left/right to see the full tree. [2]

Below is an example of how the visualizer helps with identifying at a glance whether pruning has occurred.



The image above represents a part of the output (in this case, the rightmost end) of the visualizer. The one

[2]For more information on how to run the visualizer and print trees to file, please refer to the `README.txt` file or run the command `python3 cw1.py -h`.

at the top is `noisyTree`, while the tree at the bottom is the one generated after pruning `noisyTree`. As it is possible to see, pruning is very obvious when using the visualizer.