DECISION TREE COURSEWORK

IMPERIAL COLLEGE LONDON

DEPARTMENT OF COMPUTING

CO395 Introduction to Machine Learning

Author:
Zhendong Fu, Huanyao Rong, Pinchu Ye, Yucheng Gong (CID: 01216584, 01221567, 01200801, 01098291)

Date: February 11, 2019

1 Introduction

For this assignment, we are given clean and noisy datasets containing tuples of 8 values: 7 emitted WI-FI signal strengths from a mobile phone and a room number representing its corresponding locations in one of the four rooms. Input dataset is shuffled and then used to build a decision tree that is able to classify locations when provided 7 new WI-FI signals. There are 2000 data examples in each set that are separated into different folds serving different purposes (training, validation and testing), in the same way as specified in the lecture notes. The implementation details are as follows.

2 Implementations

2.1 Loading data

Before building our decision tree, input data are first loaded onto the program using function `load_data`. This function takes in a relative path to the root directory of the main program, and check if there is such file at the specified path. If this file is found, data in the file would be loaded as a numpy array, using the `load_txt` function in numpy library; otherwise, the file does not exist, program would exit thereafter.

2.2 Building decision trees

Our decision tree is built in a recursive manner. The base case for this recursion is when all the data in the input dataset have the same labels, which means we have reached a leaf and hence a dictionary representing the leaf is returned.

Listing 1: Base case in decision tree building

```
if same_label(dataset):
        node = {'leaf': True,
        'room': dataset[0][LABEL_IDX]}
        return (node, depth)
```

In other cases, where the dataset does not have a consistent label, a split point needs to be found that will lead to the highest information gain for the two partitioned datasets. To find the most optimal splitting point, `find_split` is called. In our implementation, `find_split` is consist of two searching strategies for the best split point. At first, it will attempt to find the best split with function `find_column_split`. `find_column_split` will sort the data by its seven attributes one at a time, then iterate over that attribute and record as a split point when the labels and attribute values are both different between two consecutive tuples (split value is average of the two attribute values). Out of all split points find in one column, select the one that produces the highest information gain. Repeat these steps seven times to find all potential split points in the seven columns, and select the one that has the highest information gain across all columns, which will be considered as our final best split point. The input dataset will then be partitioned into two halves, one with attribute values smaller than and the other half with attribute values greater than or equal to the split value.

As mentioned before, `find_column_split` will find a split value of 63.5 on attribute 1 in the case 1 scenario below. However, as the size of dataset in the recursive calls becomes smaller, there might be times when no two adjacent tuples have different labels and different attribute values at the same time, as in case 2 listed below. `find_column_split`, therefore, cannot find a split point.

	attr. 1	...	label
case 1	63	...	1
	63	...	1
	64	...	2
	64	...	2
case 2	63	...	2
	64	...	2
	64	...	3

In this case, we adapt another strategy, `find_all_col_split`. It also sorts the data by one attribute at a time, but it record every split point when the attribute values or labels differ between two adjacent tuples. Likewise, the best split point for the input dataset is selected for having the highest information gain across all attributes. This two strategies can make sure a split point is always found for a dataset by working together.

The split point is represented as a node in the decision tree, in a dictionary structure as well:

Listing 2: Recursive case in decision tree building

```
(s_attr, s_val, l_dataset, r_dataset) = find_split(dataset)
(l_branch, l_depth) = decision_tree_learning(l_dataset, depth + 1)
(r_branch, r_depth) = decision_tree_learning(r_dataset, depth + 1)

node = {'attr': s_attr,
        'val': s_val,
        'left': l_branch,
        'right': r_branch,
        'leaf': False}

return (node, max(l_depth, r_depth))
```

The depths of the decision trees are also returned by the recursion, and the maximal depth of a decision tree is the depth of its branch that has largest depth amongst all branches.

2.3 Calculating information gain

After finding the possible split points, the performance of the partition is determined by calculating the information gain. Information gain measures by how much has the information entropy of the dataset been reduced. Therefore, split point that results in the highest information gain will be stated as the best split value for that dataset. The information gain of an split value is calculated by following equations:

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

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

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

$|S|$: number of samples in subset S

H : entropy of the dataset

2.4 Cross validation

For this task, we used 10-fold cross validation. The dataset is first shuffled by sampling them randomly using `shuffle` in the `random` library, after which the shuffled dataset is divided into 10 folds with equal size. One fold is randomly chosen as the test fold and would not be involved in any of the training or validation process. The rest 9 folds are used to train and validate in 9 iterations. In each iteration, a validation fold is chosen and it does not overlap with the validation folds in other iterations, while the rest of data are all used to build the decision tree. This also means every one of the 9 folds has a chance of being the validation fold. After all iterations have completed, we calculate the average confusion matrix and four average metrics (recall, precision, classification rate and f1 measure/score).

2.5 Evaluation and results

For this task, 10-fold cross validation is used in both clean and noisy datasets. In each iteration of the validation, input data set is separated into 10 folds with equal fold length, 9 of them are used for training and the remaining one is used for testing. Therefore, 9 decision trees are constructed using training data set and their performance is evaluated comparing the predicted labels trained by trees and actual labels in the test data fold. Confusion matrix is designed in each iteration and they are averaged after 9 iterations.

The average recall, average precision, average class rate and F1 measure are determined from the average confusion matrix. Following is the formula to calculate them:

TP : True Positives, TN : True Negatives, FP : False Positives, FN : $FalseNegatives$

$$Recall = \frac{TP}{TP+FN} \quad (1) \qquad Precision = \frac{TP}{TP+FP} \quad (2)$$

$$F_1 measure = 2 \times \frac{Precision * Recall}{Precision + Recall} \quad (3) \qquad ClassificationRate = \frac{TP+TN}{TP+TN+FP+FN} \quad (4)$$

2.5.1 Clean dataset before pruning

		Predicted Class			
		1	2	3	4
Actual Class	1	53.7	0.0	0.0	0.0
	2	0.0	35.6	1.4	0.0
	3	0.1	0.0	56.9	1.3
	4	1.0	0.0	0.0	48.0

Table 1: Average Confusion Matrix - Clean Dataset

	Class				
	1	2	3	4	Average
Recall Rate	99.4	96.2	94.8	98.0	97.1
Precision Rate	98.0	95.4	97.6	96.8	97.0
CR	99.3	99.3	99.3	99.3	99.3
F1 Measure	98.7	95.8	96.2	97.4	97.0

Table 2: Evaluation Results (%) - Clean Dataset

From table 2, the average of recall rate, precision rate, F1 measure and classification rate are very high and are all above 97%. Therefore the predicted performance of clean data are great for all classes. All classes have the same classification rate of 99.3% .

2.5.2 Noisy dataset before pruning

		Predicted Class			
		1	2	3	4
Actual Class	1	35.1	1.8	2.7	3.4
	2	3.3	49.1	2.9	3.7
	3	4.9	4.1	39.5	1.5
	4	3.3	5.0	4.6	35.1

Table 3: Average Confusion Matrix - noisy Dataset

	Class				
	1	2	3	4	Average
Recall Rate	81.6	83.2	79.0	73.1	79.2
Precision Rate	75.3	81.8	79.5	80.3	79.2
CR	89.1	89.1	89.1	89.1	89.1
F1 Measure	78.3	82.5	79.2	76.6	79.1

Table 4: Evaluation Results (%) - noisy Dataset

By observing table 4, all classes also have the same classification rate(88.1%). Class 4 has a high precision rate(80.3%) but a low recall rate(73.1%), which indicates that a lot of positive examples are missed but those predicted positives are positive. Class 1 has the lowest precision rate(75.3%) but a relatively high recall rate(81.6%) which means most of the positive examples are correctly recognized but there are a lot of wrongly predicted negatives.

3 Questions

3.1 Is there any difference in the performance when using the clean and noisy dataset?

By observing table 2 and table 4, we can see a noticeable performance difference between clean and noisy dataset before pruning. The classification rate is around 10% different, and the performance of recall rate, precision rate, F1 measure acts about 20% worse than clean dataset. This can be due to the fact that collected noisy data is inaccurate WIFI signal strength affected by signal receiver. However, he difference between distinct classification labels are not exactly the same. From the table, class 4 acts much more confused than the other 3 classes.

Table 6 and table 8 shows the performance of the dataset after pruning, the classification rate difference between clean and noisy dataset is reduced to 5% after pruning, which has 99.3% and 94.5% accuracy respectively. Class 2 have a relatively low recall rate(84.7%) and class 1 have a relatively low F1 measure(79.2%).

3.2 Pruning implementation and influences

3.2.1 Implementation

The main logic of pruning is implemented in function `prune_help`, which is a helper function for depth-first search using head-recursion. There are 3 arguments for this function: the current node that is being tested for pruning, the root node that remains same for all recursion, and the validation data for the validation error. The first arguments are used for recursion and the second and third arguments are used to test the tree. The main idea is to test all nodes whose 2 children are both leaves, and try to prune it using the results from both sides. This produce 3 cases: unmodified tree, pruned tree using label from left child node, and pruned tree using label from right child node. Then use the root node, which is copied by reference into the function, to evaluate the tree using the validation data for all 3 cases, compare the their performances, and take the decision with minimum mistakes.

The reason why we use head recursion is that we want to prune the subtree before testing the node, because the node that do not have 2 leaves as children may have 2 leaves after pruning its children.

3.2.2 Clean dataset after Pruning

		Predicted Class			
		1	2	3	4
Actual Class	1	53.7	0.0	0.0	0.0
	2	0.0	35.8	1.2	0.0
	3	0.1	2.3	56.2	1.4
	4	1.0	0.0	0.0	48.0

Table 5: Average Confusion Matrix - Clean Dataset

	Class				
	1	2	3	4	Average
Recall Rate	99.4	96.8	93.7	98.0	97.0
Precision Rate	98.0	94.0	97.9	96.6	96.6
CR	99.3	99.3	99.3	99.3	99.3
F1 Measure	98.7	95.4	95.8	97.3	96.8

Table 6: Evaluation Results (%) - Clean Dataset

3.2.3 Noisy dataset after pruning

		Predicted Class			
		1	2	3	4
Actual Class	1	40.3	0.7	0.2	1.8
	2	2.0	54.3	1.8	0.9
	3	3.2	4.8	40.1	1.9
	4	2.1	4.0	3.9	38.0

Table 7: Average Confusion Matrix - noisy Dataset

	Class				
	1	2	3	4	Average
Recall Rate	93.7	84.7	94.5	90.0	90.7
Precision Rate	84.7	85.1	87.2	89.2	86.6
CR	94.5	94.5	94.5	94.5	94.5
F1 measure	79.2	89.2	94.5	83.9	86.7

Table 8: Evaluation Results (%) - noisy Dataset

3.2.4 Influence of pruning

The influence to the clean dataset is not obvious: when testing using test sets, the overall accuracy remains almost unchanged, with some test sets better and some test sets worse only in small amount. This is because we prune the tree using the validation set and test it using the test set, so it is possible for the result to be a little bit worse for some test sets. However, when testing using noisy dataset, the accuracy increases almost for every test set. The primary reason is that the noisy data is more likely to produce over-fitting problem so pruning almost always makes the results better.

3.3 Relationship between maximal depth and prediction accuracy

	Fold									
	0	1	2	3	4	5	6	7	8	Average
original depth	12	10	13	11	12	13	13	13	12	12
validation error rate	4.5	4.0	4.5	5.0	6.0	6.0	5.0	5.0	3.5	4.35
pruned depth	9	5	11	8	8	4	10	5	11	7
validation error rate	3.5	3.5	5.0	3.0	4.0	3.5	3.5	2.0	4.5	3.25

Table 9: maximum depth and prediction accuracy - clean dataset

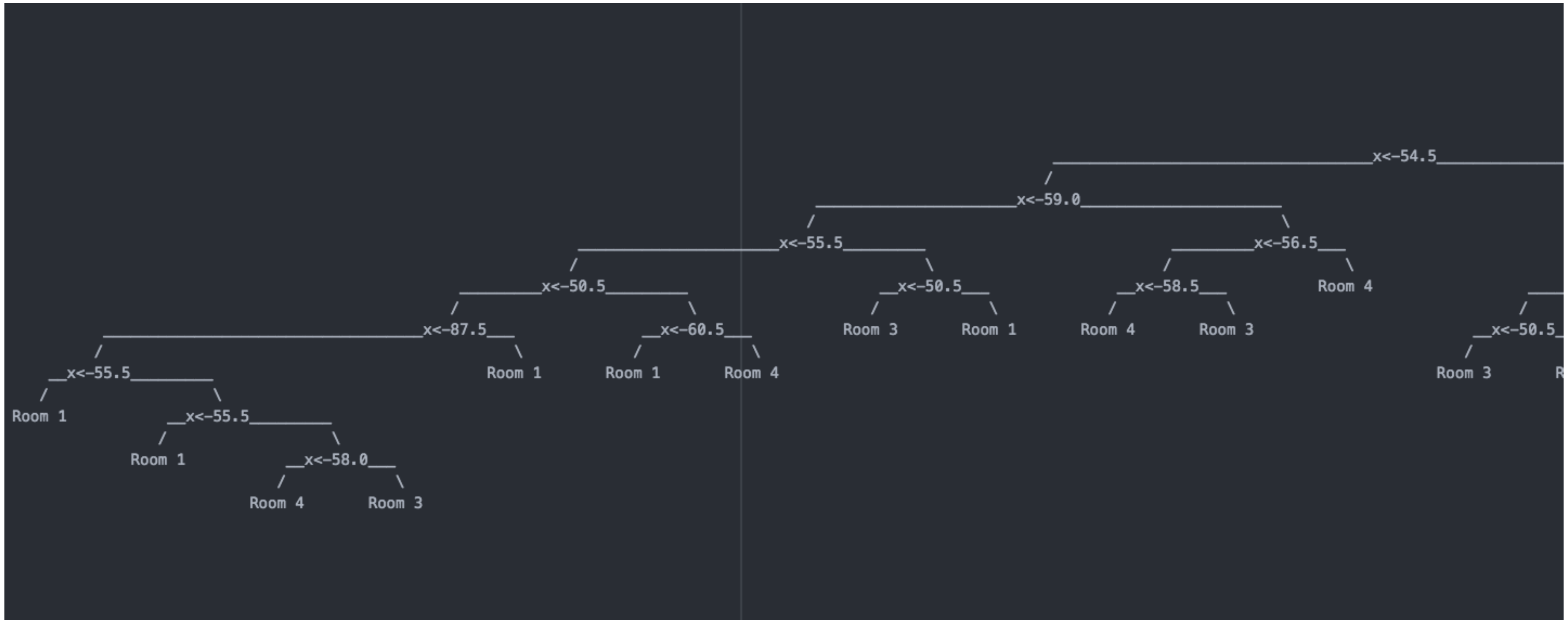
	Fold									
	0	1	2	3	4	5	6	7	8	Average
original depth	18	20	21	24	24	22	18	18	19	20
validation error rate	21.5	18.5	18.0	18.5	19.0	19.5	21.0	18.5	18.5	17.3
pruned depth	13	12	13	11	11	14	10	12	14	12
validation error rate	12.0	13.0	9.5	12.5	15.0	13.5	10.5	10.0	11.5	10.8

Table 10: maximum depth and prediction accuracy - noisy Dataset

Average maximal depth of the tree of clean dataset is 12 before pruning and 7 after pruning. And noisy dataset has average depth of 20 before pruning and 12 after pruning. Since the general performance of the decision tree is improved after pruning, we can conclude that maximal depth and prediction accuracy has a negative relationship. As the maximal depth becomes smaller, decision tree have a better prediction accuracy. Validation error rate for clean dataset is reduced from 4.35% to 3.25% and for noisy dataset is reduced from 17.3% to 10.8%. Although the classification rate is unchanged for clean dataset, it is improved from 89.1% to 94.5% as maximal depth decreases.

4 Visualization of decision trees

We visualize our decision trees by printing the edges, leaves and nodes to the output console. The output tree is quite large, hence we only attach a screenshot of a small part of the entire tree. The following is a visualization of a pruned tree produced from the clean dataset.



```
                                                                                                          ________________________________x<-54.5_____________
                                                                                                         /
                                                                                  ____________________x<-59.0____________________
                                                                                 /                                               \
                                                         _____________________x<-55.5________                         ________x<-56.5___
                                                        /                                    \                       /                  \
                                             _________x<-50.5________                   __x<-50.5___            __x<-58.5___         Room 4               ____
                                            /                        \                 /           \           /           \                            /
         _________________________________x<-87.5___            __x<-60.5___        Room 3      Room 1      Room 4      Room 3                      __x<-50.5_
        /                                          \           /           \                                                                        /
    __x<-55.5_________                          Room 1      Room 1      Room 4                                                                  Room 3      R
   /                  \
Room 1          __x<-55.5_________
               /                  \
            Room 1          __x<-58.0___
                           /            \
                        Room 4      Room 3
```