CO553 - Decision Tree Coursework

Implementation

Data representation: We implemented our decision tree data structure as a `Node` class, with four properties: `left`, `right`, `split` and `room`. If the node is not a leaf, the `left` and `right` properties hold respective child nodes and `split` holds a `Split` object which contains information on how a given datapoint should traverse that node on its way down the decision tree. The `Split` object contains two properties, `attrIndex` and `lessThan`, which store information about the attribute and threshold of the split respectively. If a datapoint's attribute at position `attrIndex` is less than `lessThan`, go left. Otherwise, go right. Finally, if a node is a leaf, then `left`, `right`, and `split` will all be `None` and `room` is set to the label corresponding to the predicted classification of data points that reach this node.

Building the decision tree: The decision tree is built to perfectly classify the training set. The algorithm starts by figuring out the best possible split. To do this, a `get_split` method is used to find the attribute that best divides the data, according to entropy gain. The `get_split` method iterates over each attribute, sorting the dataset by that attribute, and then calculating the entropy gain for every possible division value present in the dataset for that attribute. The split that provides greatest entropy gain is chosen.
The entropy gain generally refers to the entropy gain function, but in our implementation, we optimise by calculating only the weighted average of the entropies of the two partitions of the dataset, and not the entropy for the full dataset, since it does not play a role in choosing the best split.

Pruning: Pruning is recursive and in-place, and runs bottom-to-top. The pruning process takes a node, a partial validation set, and partial training set, each corresponding to the datapoints that would reach that node while traversing the tree downwards. It first checks if the node is a leaf, in which case there is no need to prune it; just return it. Secondly, if the validation set has no data points that reach this node, prune it and return. Otherwise, split the validation set into left and right datasets according to the split at this node. Next, recursively run pruning on the left and right nodes according to the left and right validation and training sets. Now, check if the left and right nodes are leaves; if they both are, consider pruning this node. [Because we're recursing from the bottom of the tree, we can guarantee that we won't need to prune the same tree multiple times, because if pruning were to cause new leaves to be created below this node, it would have done it already.] To consider this node for pruning, calculate the validation error on this node with the validation set. Then calculate which label (room) would be attached to this node if the two leaves were merged, according to the training set. Compare the validation error against this theoretical new leaf. If the validation error is lower for the theoretical new leaf, then remove the two child leaves and convert this node into a leaf with the above-calculated label (room).

K-Fold and Cross-Validation: To perform k-fold, we first split the dataset into k folds that evenly distribute the rows (to the extent possible). The algorithm then iterates through each fold, keeping one out for use as the test set. If k-fold is not being used with pruning, the remaining `k-1` folds are used as a training set to generate decision trees. The decision trees and corresponding test sets are returned. If pruning, the `k-1` remaining folds are iterated over to select a validation set, a decision

tree is generated with the remaining `k-2` folds, and then pruned using the selected validation set. After all `k-1` decision trees are generated and pruned (corresponding with the `k-1` possible validation sets), validation accuracy is calculated on each tree, and the one with the highest validation accuracy is chosen. A total of `k(k-1)` decision trees will be generated in total, and k decision trees will be returned, along with the corresponding left-out test set.

Computing average metrics: Metrics are calculated based on the outputs of k-fold cross validation, rather than just one model and one test set. We do this by averaging the confusion matrices for the test set and model pairs. This average confusion matrix is then used to calculate the additional metrics - precision, recall and accuracy. The F1-score is calculated simply using precision and recall values. The average height is calculated by averaging the heights calculated by recursion down each tree.

Parallelizing decision tree generation: One of the main advantages of using k-fold is how easy it is to compute in parallel. We found that by parallelizing each of the k-folds, each with its own process, we were able to achieve a nearly linear speed up in the number of parallel jobs. We also implemented a parallelized version of the `get_split` method described above to parallelize the process of selected the best split in a dataset for decision tree construction.

Plotting: A naive approach to plotting starts at the top and moves down, plotting left and right children a fixed distance down and left/right the parent. It is immediately apparent, however, that this leads to overlap between nodes at the same level. To avoid overlaps, information about the number of nodes at a given height must be known by all nodes at that height. This, however, is hard to implement in a recursive fashion. Another approach involves assuming that a tree is full. This approach leads to a visually unappealing tree in real cases, where the tree is far from full, with an exponential amount of white space where a branch ends early.

Our approach is to plot bottom-up from leaves. We start by counting the number of leaves with in-order traversal. We can thus assign the leaves x-coordinates by simply evenly distributing the width of the plot between each of them. Once that information is calculated, the algorithm traverses the tree recursively, along the way recording the y-coordinate of each node by simply moving a fixed amount with each downward step. Leaves are drawn first, with y-coordinate as above, and x-coordinate based on the left-to-right position of the leaf among all leaves, also as calculated above. The function then returns the x coordinate of the leaf to the previous call in the call stack. From there, nodes are drawn bottom-to-top, with y-coordinate depending on their depth, and x-coordinate calculated as the average of their children's x-coordinates. Once the algorithm draws the root node, it stops.

Getting the coordinate calculations right was the main challenge of the tree plotting function. After that, using matplotlib, the text was displayed according to the type of the node. For leaves, the room classification was written. For the remaining nodes, the attribute and threshold were written ("`A# < threshold`"). Additionally, leaves were given a different color for aesthetic reasons. The resulting images for the pruned trees of both the clean and noisy dataset can be found in the zip file as `pruned_clean_dataset.png` and `pruned_noisy_dataset.png` respectively. The images are not in the report as their dimensions are too large to fit in the page.

Results

Clean Dataset (without pruning)

Accuracy: 96.8%
Average Precision: 96.82%
Average Recall: 96.8%
Average F1: 96.80%
Average Height: 13.3

Analysis:
Accuracy is very high, despite the lack of pruning, which in theory should cause extreme overfitting. This tells us that the data doesn't contain much noise. Precision and recall are almost the same, which is apparent just from the relative symmetry of the confusion matrix. In other words, for example, the decision tree is equally likely to predict that a datapoint is in room 3 when it's actually in room 1 as vice versa. Rooms 1 and 2 are never confused with each other, and neither are 2 and 4. Rooms 2 and 3 are sometimes misclassified amongst themselves, so it's likely that they're near each other. The same can be observed, but to a lesser degree, with rooms 1 and 4. Accuracy is highest in rooms 1 and 4

Clean Dataset (with pruning)

Accuracy: 96.85%
Average Precision: 96.84%
Average Recall: 96.85%
Average F1: 96.84%
Average Height: 9.9

Analysis:
Results are marginally better than without pruning, with accuracy 0.05% higher, and F1 0.04% better. The average tree height goes down *significantly* from 13.3 to 9.9. This is indicative of a reduction in overfitting. The improvement in metrics can be explained by the reduction in overfitting. The small improvement can be explained by the fact that the dataset is very clean. Overfitting plays a larger role when overfitting noise, but in an evenly distributed dataset without much noise, pruning won't make much of a difference. As these results are based on the same dataset as the previous section, elements of the analysis of the last section are relevant here too. The symmetry in the confusion matrix observed above remain true, so precision and recall also remain very close. Other observations regarding the confusion matrix still stand with the pruned model.

Noisy Dataset (without pruning)

Accuracy: 80.55%
Average Precision: 80.57%
Average Recall: 80.53%
Average F1: 80.54%
Average Height: 19.1

Analysis: The height of the decision tree is significantly higher than for the clean dataset (19.1 compared to 13.3). This is due to the extra amount of nodes needed to overfit the training set. More nodes are needed since the dataset contains more noise and so every node generalizes more poorly when compared to a clean dataset. For instance, if one data point labeled as room 1 shares most of

its features with points from label 2 (due to noise), the decision tree might create several extra nodes to isolate that one case and label it as room 1.
All metrics are significantly lower when compared to the clean dataset. This makes sense, since the tree overfits to noise. Once again, the symmetry properties described previously are present, as well as the similar values for average precision and recall. Room 1 seems to be the most likely to be misclassified, while room 3 is the most likely to be correctly classified. Points in room 2 are twice as likely to be misclassified as room 1 than room 4.

Noisy Dataset (with pruning)

Accuracy: 88.25%
Average Precision: 88.27%
Average Recall: 88.27%
Average F1: 88.26%
Average Height: 15.4

Analysis:
Pruning dramatically improves all metrics, bringing the accuracy up from 80.55% to 88.25%. This is expected, given the amount of overfitting on noise in the unpruned tree. The average tree height decreases significantly from 19.1 to 15.4, which can be explained by the reduction in overfitting. As with the clean dataset, rooms 2 and 3 are the most likely to be misclassified between one another, as well as rooms 1 and 4.

Questions

Noisy-Clean Datasets: The clean dataset produced decision trees with extremely high accuracy, even without pruning (96.8%), and only slightly better accuracy with pruning (96.8%). The noisy dataset, on the other hand, initially produced decision trees with fair accuracy (80.55%) which improved only with pruning to a decently good accuracy (88.25%). With the clean dataset, pruning was helpful, but played a far smaller role because far less overfitting to noise took place, compared to with the noisy dataset.

Additionally, the prediction error in the clean dataset can be explained predominantly by noise in the dataset, rather than a failure of the decision tree. Rooms 1 and 4 are confused with each other, and similarly 2 and 3. The fact that these prediction errors are not uniform is indicative that the issues in the classified data points between these pairs is the result of rooms being near each other, rather than an issue with the decision tree construction. With the noisy dataset, this pattern is still present, but to a much smaller degree that's only visible after pruning. This means that, for the most part, data points from the clean dataset might be misclassified as being on the other side of a wall, while with the noisy dataset, data points might be misclassified as being further away.

It is interesting to note, that information about the rooms layout is leaked through the confusion matrix, even though the decision tree was never explicitly trained to capture this.

Pruning Question: Our explanation of the implementation of our pruning function can be found above, in our implementation section. As described above, pruning drastically slightly improves performance on the clean dataset and drastically improves performance on the noisy dataset. As explained above, overfitting plays a larger role in noisy datasets, where overfitting is common, compared to with clean data.

Depth Question: Pruning decreases the maximum height of trees significantly, largely because the deepest sections of the decision tree are likely to be the ones most responsible for overfitting. If one small section of the decision tree has a very large height, it's often present simply to overfit one or a few data points, and would likely be removed during the pruning process.

Across the board, a decrease in tree height (or depth) correlated with higher prediction accuracy, whether in the clean or noisy dataset. This makes sense, because deeper parts of the tree are most likely to be responsible for classifying outliers or boundary points. Comparing the average height of the pruned trees trained on the noisy dataset with those trained on the clean dataset, we see a significant difference, with heights of 15.4 and 9.9 respectively. Pruning is also more effective on noisy data, because generalizations of pruned nodes (by merging to branches into one class) can potentially correctly classify noisy data that has similar attributes to other classes. As per the example illustrated in the first bullet point from the noisy data without pruning analysis, pruning may be more effective on boundary points when more noise is present.

To reduce overfitting even further without losing valuable information, a different pruning algorithm could be used. However, it is important not to prune the tree too much, otherwise the model could prove to be too simple to explain the data.

Appendix

Clean Dataset

Average Confusion Matrix over 10 folds:

Actual/Predicted	Room 1	Room 2	Room 3	Room 4
Room 1 (Actual)	49.3	0	0.2	0.4
Room 2 (Actual)	0	47.1	1.7	0
Room 3 (Actual)	0.2	2.9	47.9	0.3
Room 4 (Actual)	0.5	0	0.2	49.3
F1	98.7	95.34	0.95	0.99
Precision	98.8	96.52	93.37	98.6
Recall	98.6	94.2	95.8	98.6

Clean Dataset (with pruning)

Average Confusion Matrix over 10 folds:

Actual/Predicted	Room 1	Room 2	Room 3	Room 4
Room 1 (Actual)	49.8	0	0.7	0.5
Room 2 (Actual)	0	47.8	2.1	0
Room 3 (Actual)	0.2	2.2	46.8	0.2
Room 4 (Actual)	0	0	0.4	49.3
F1	98.61%	95.7%	94.16	98.9%
Precision	97.65%	95.79%	94.74%	99.2%
Recall	99.%6	95.6%	93.6%	98.6%

Noisy Dataset (without pruning)

Average Confusion Matrix over 10 folds:

Actual/Predicted	Room 1	Room 2	Room 3	Room 4
Room 1 (Actual)	37.7	3.2	3.3	3.4
Room 2 (Actual)	3.9	39.9	2.8	2
Room 3 (Actual)	4.1	4.3	42.1	3
Room 4 (Actual)	3.3	2.3	3.3	41.4
F1	78.05%	81.18%	80.19%	82.72%
Precision	79.20%	82.1%	78.69%	82.31%
Recall	76.94%	80.28%	81.75%	83.13%

Noisy Dataset (with pruning)

Average Confusion Matrix over 10 folds:

Actual/Predicted	Room 1	Room 2	Room 3	Room 4
Room 1 (Actual)	44.5	1.8	2	2.4
Room 2 (Actual)	1.2	44.1	3.3	1.6
Room 3 (Actual)	1.2	2.6	44.6	2.5
Room 4 (Actual)	2.1	1.2	1.6	43.3
F1	89.27%	88.29%	87.11%	88.37%
Precision	87.77%	87.85%	87.62%	89.83%
Recall	90.82%	88.73%	86.6%	86.95%