**56. Optimizing Warehouse Logistics Using Decision Trees and Cost Complexity Pruning for Accurate Predictions**

In this project, I focus on applying tree-based methods to optimize warehouse logistics. At any point after building a decision tree, I predict a test observation by passing it down the tree while following each split. This observation will end up in a terminal node, and I use the mean of the training observations within that region to make the prediction. To illustrate this, I present a larger example, which, though simplified, demonstrates the concept clearly.

In the example, I show two panels. In the left panel, I present a set of regions that could not be achieved using sequentially split data. These regions aren't represented as boxes formed by sequential splitting, unlike those in the right panel. When I examine the right panel alongside the decision tree in the bottom left, I notice that the first split is made at T1, which represents a vertical split. This split divides the predictor space, consisting of two variables, X1 and X2, into a left and a right region. This is the initial partition.

Next, I perform a split on X2 at T2, dividing the left region into two separate regions. This takes care of the left partition, resulting in Region 1 and Region 2. On the right-hand partition, formed by the first split at T1, I split X1 again at T3, making another vertical split. This further partitions the space, resulting in Region 3 on the left. On the right, I execute one more split at T4 on variable X2, dividing the region into two additional parts. By sequentially splitting in this greedy fashion, I subdivide the two-variable space into five distinct regions.

At each of the terminal nodes, I approximate the regression function by the mean of the training observations that end up in those terminal nodes. This means that when I have a test observation and want to predict it, I start at the top of the tree and query its value for X1. If the value is less than T1, I go to the left; otherwise, I proceed to the right. I continue this process at each internal node, eventually landing in one of the five regions. The prediction is then the mean of the observations in that region.

Because I am working with a two-dimensional function in this example, I can represent the resulting piecewise constant function visually. The right panel illustrates how the function is constant within each of the regions defined by splits like T1 and T2. This provides a clear way to understand how my function is represented by the tree.

Having demonstrated the tree-growing process, I now face the question of how large the tree should be. When should I stop growing the tree? One option could be to grow the tree as large as possible. In the extreme case, I could end up with a tree where each observation is its own terminal node. However, this approach would likely lead to overfitting, resulting in poor test set performance. A tree this large would have zero training error because it perfectly fits the training data, but it would not generalize well to new test data, leading to high prediction errors.

Another option would be to stop growing the tree early. I could continue splitting until no further split significantly reduces the residual sum of squares (RSS). However, this strategy can be too shortsighted. Often, it stops too early because it appears that no beneficial split is available, when in reality, a better split might be achievable further down the tree.

Instead, I find that a more effective strategy is to grow a large tree initially, using a rule such as ensuring that each terminal node has no fewer than five observations. Then, with this large tree—probably larger than needed—I prune it from the bottom. This pruning process helps produce a mid-sized tree that balances bias and variance, achieving lower prediction error compared to either a very small tree or an overly large tree.

A "bushy" tree, with many splits, tends to have high variance because it overfits the data without any bias, meaning it’s too tailored to the training data and unlikely to predict well on new data. What I need is a method that reduces variance while maintaining a simple structure. In the CART software or the tree library in R, for example, I implement this by building a large tree and pruning it back using a technique called cost complexity pruning, or weakest link pruning.

Cost complexity pruning involves setting up an objective function similar to the Lasso method used in regression. The objective function measures the fit based on the sum of squares of observations around their assigned terminal nodes. Let’s denote these terminal nodes as R1 through Rt, where t represents the number of terminal nodes. The sum of squares of observations around the mean of each region (a terminal node) is calculated, and this is summed over all terminal nodes.

I want a tree with low variance, but I also need to control the size of the tree to prevent overfitting. To achieve this, I introduce a penalty for the total number of nodes in the tree, controlled by a penalty parameter, alpha, which I estimate using cross-validation. This is akin to the Lasso method where I penalize the sum of absolute values of coefficients to prevent overfitting. Here, I penalize the size of the tree.

Cost complexity pruning finds the best value of alpha by cross-validation. I then select the subtree from the larger tree that has the smallest value of this criterion. While software packages like CART or the tree library in R handle this cross-validation process automatically, it's beneficial to understand how this process works.

To summarize the tree-growing algorithm: I build a large tree with a simple stopping rule, such as not splitting a terminal node that has fewer than five observations. Then, I prune the tree to find the best subtree. For this, I need to determine the cost complexity parameter that provides the best trade-off between fit and tree size. I use cross-validation in the same way it is used in regression, dividing the data into k parts (usually five or ten). I set aside one part, fit trees of various sizes on the remaining k-1 parts, and evaluate the prediction error on the part left out.

The parameter alpha determines the size of the tree, and cross-validation is used to choose the best alpha. Once I choose alpha, I go back to the full tree and find the subtree that has the smallest prediction error when grown on all the training data.

To illustrate this process with a real-world example related to warehouse logistics, I took a dataset of warehouse operations and divided the observations into a training set and a test set. I wanted to compare the cross-validation error to the test error on a separate set. I built a regression tree on the training set using cross-validation to find the optimal cost complexity parameter, alpha, using six-fold cross-validation. I chose six folds to ensure a balanced split.

The result was a fully grown tree before pruning, with multiple branches. The only rule was not to split a terminal node with fewer than five observations, resulting in a tree with eleven nodes. However, not all these nodes are predictive. The length of the tree's branches varies because early splits cause significant reductions in the sum of squares, depicted by long branches. As I go deeper, the incremental improvements get smaller, which is why some branches are shorter.

By applying cross-validation, I identify the best-pruned tree. I see that as I vary alpha, it affects the tree size in direct correspondence. When alpha is zero, there is no penalty, resulting in the largest tree with about twelve terminal nodes. As alpha increases, the penalty on tree size also increases, eventually resulting in a minimal tree with just one node and no splits.

The cross-validation curve helps me identify that the optimal number of terminal nodes is around three, confirming that the three-node tree is indeed a good choice. Based on this graph, I finalize the pruned tree that best optimizes warehouse logistics operations by balancing prediction accuracy and tree complexity.