Contents

1 Regression Trees

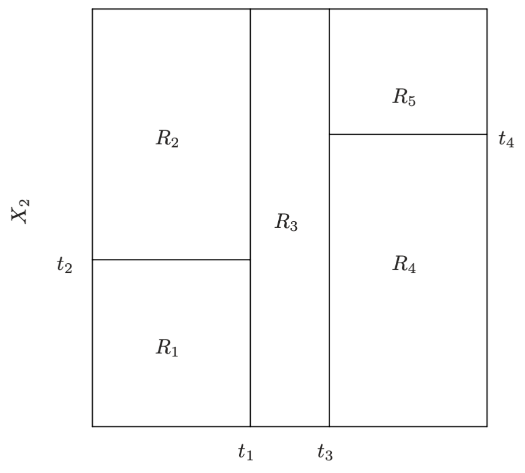
1.1 Motivating Example

Instead of relying on traditional ways to construct regressions, we may employ tree-based methods for building these models. The tree-based methods involve **stratifying** or **segmenting** the predictor space into a number of simple regions. In order to make a prediction for a given observation, we typically use the mean or the mode response value for the training observations in the region to which it belongs. Since the set of splitting rules used to segment the predictor space can be summarized in a tree, these types of approaches are known as **decision tree** methods.

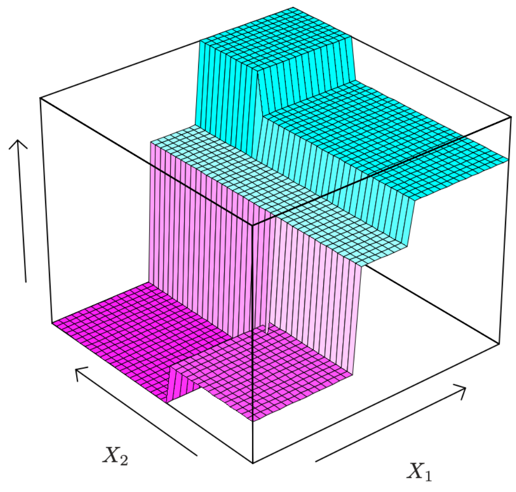
The decision tree approach makes predictions via stratification of the feature space. The process of building a regression tree, roughly speaking, involves two steps:

1. We divide the predictor space − that is, the set of possible values for $X_1, X_2, \ldots, X_p$ — into J distinct and non-overlapping regions, $R_1, R_2, \ldots, R_J$
2. For every observation that falls into the region R_j, we make the same prediction, which is simply the mean (or mode) of the response values for the training observations in R_j

For instance, suppose we have two predictor variables X_1 and X_2 and we partition the predictor space, which is a two-dimensional feature space, into five-regions $R_1, \ldots, R_5$. The partitioning can be visualized as follow:



(a) The output of recursive binary splitting on a two-dimensional feature space



(b) A perspective plot of the prediction surface corresponding to the partitioning; the heights or z-values represent the mean responses for all observations in each of the regions R_i

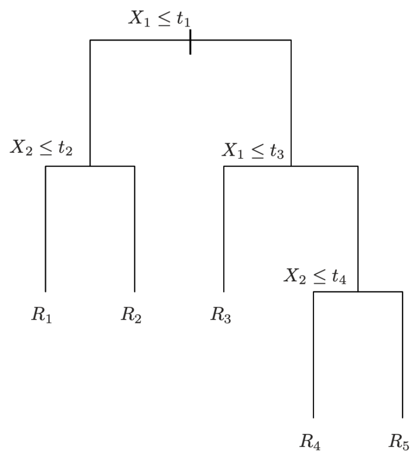


Figure 2: A tree corresponding to the partition in the top left panel

In keeping with the tree analogy, the regions $R_1, \ldots, R_5$ are known as **terminal nodes** or **leaves** of the tree. Decision trees are typically drawn upside down, in the sense that the leaves are at the bottom of the tree. The points along the tree where the predictor space is split are referred to as **internal nodes** or **children nodes**. In the example above, there are four internal nodes. Lastly, we refer to the segments of the trees that connect the nodes as branches.

1.2 Building Regression Trees Using Recursive Binary Splitting

In general, how do we construct the regions $R_1, \ldots, R_J$? In theory, the regions could have any shape. We choose to divide the predictor space into high-dimensional rectangles called **Hyperrectangle**, or **boxes**, for simplicity and for ease of interpretation of the resulting predictive model. The goal is to find boxes $R_1, \ldots, R_J$ that minimize the SSE (or any other cost function), given by

$$f(j,s) = \sum_{j=1}^{J} \sum_{i \in R_j} \left(Y_i - \hat{Y}_{R_j} \right)^2$$

where

- $\hat{Y}_{R_j}$ is the mean response for the training observations within the j^{th} box $\frac{\sum_{i \in R_j} Y_i}{n_j}$ (n_j is the number of observations in the j^{th} box)

The double summation operator is nothing more than a sum of a sum. For each j, the SSE is computed using observations where $i \in R_j$; then, we sum all j SSE's. However, it is computationally infeasible to consider every possible partition of the feature space into J boxes. For this reason, we take a top-down, greedy approach that is known as **recursive binary splitting**. The approach is top-down because it begins at the top of the tree, the root node at which point all observations belong to a single region, and successively splits the predictor space; each split is indicated via two new branches (True or False) further down on the tree. It is greedy because at each step of the tree-building process, the best split is made *at that particular* *step*, rather than looking ahead and picking a split that will lead to a better tree in some future step. Thus, a greedy algorithm often produces a solution that's reasonably good but not guaranteed to be optimal. Unfortunately, finding the optimal tree is known to be an NP-Complete problem: it requires $O(\exp(m))$ time, making the problem intractable even for small training sets.

In order to perform recursive binary splitting, we first select the predictor X_j and the cutpoint s such that splitting the predictor space into the regions $\{X \mid X_j < s\}$ and $\{X \mid X_j \geq s\}$ leads to the greatest possible reduction in SSE. (The notation $\{X \mid X_j < s\}$ means the region of predictor space in which predictor variable X_j takes on a value less than s.) That is, we consider all predictors $X_1, \ldots, X_p$, and all possible values of the cutpoint s for each of the predictors $X_1, \ldots, X_p$, and then choose the predictor j and cutpoint s pairing such that the resulting tree has the lowest SSE. In greater detail, for any j and s at each splitting, we define the pair of half-planes

$$R_1(j,s) = \{X \mid X_j < s\} \text{ and } R_2(j,s) = \{X \mid X_j \geq s\}$$

and we seek the value of j and s that minimize the equation

$$\sum_{i:\ x_i \in R_1(j,s)} \left(Y_i - \hat{Y}_{R_1}\right)^2 + \sum_{i:\ x_i \in R_2(j,s)} \left(Y_i - \hat{Y}_{R_2}\right)^2$$

where $\hat{Y}_{R_1}$ is the mean response for the training observations in $R_1(j,s)$, and $\hat{Y}_{R_2}$ is the mean response for the training observations in $R_2(j,s)$. The variable i of the summation operators reads as follows: i such that x_i is either in the half-plane $R_1(j,s)$ or $R_2(j,s)$. **In other words, the trials or instances are split based on whether the predictor variable X_j for a particular trial or instance takes on a value either $<$ or $\geq$ the cutpoint s.** Next, we repeat the process, looking for the best predictor and best cutpoint in order to split the data further so as to minimize the SSE within each of the resulting regions. However, this time, instead of splitting the entire predictor space (which means that we apply the splitting to both the resulting regions R_1 and R_2), we split only one of the two previously identified regions, which leads to three regions. Again, we look to split one of these three regions further, so as to minimize the SSE. In the five region example above, we see that the it is indeed the case that only one of the two branches is further split at each internal node.

Finally, the process continues until a stopping criterion is reached. Or, if left unconstrained, the algorithm will adapt itself to the training data, fitting it very closely to the point of over-fitting. Once the regions $R_1, ..., R_J$ have been created,we predict the response for a given test observation using the mean of the training observations in the region to which that test observation belongs.

1.3 Scikit-Learn Recursive Binary Splitting

1.3.1 Regression

In machine learning applications, the recursive binary splitting algorithm is sometimes implemented slightly differently as a slightly different cost function is employed. For instance, the Sklearn package in Python uses the Classification and Regression Tree (CART) algorithm (also called "growing" trees). The algorithm works by first splitting the training set into two subsets (we can call these R_{left} and R_{right} to be consistent with the notations used earlier) using a single feature j and a threshold s_j. The algorithm searches for the pair (j, s_j) that minimizes SSE for each of the two subsets of instances, weighted by the ratio of the number of instances in that subset to the total number of instances in the training set. The CART cost function for regression can be expressed as follows:

$$f\left(j, s_j\right) = \frac{n_{R_{\text{left}}}}{n} SSE_{R_{\text{left}}} + \frac{n_{R_{\text{right}}}}{n} SSE_{R_{\text{right}}}$$

where

- $n_{R_{\text{left}}}$ and $n_{R_{\text{right}}}$ are the numbers of instances in each of the subsets or regions
- n is the total number of instances or trials in the training set
- The SSE, or sum of squares residuals, are defined as follows (some practitioners call these MSE):

$$SSE_{R_{\text{left}}} = \sum_{i:\, x_i \in R_{\text{left}}(j,s)} \left(Y_i - \hat{Y}_{R_{\text{left}}}\right)^2$$

$$SSE_{R_{\text{right}}} = \sum_{i:\, x_i \in R_{\text{right}}(j,s)} \left(Y_i - \hat{Y}_{R_{\text{right}}}\right)^2$$

Once the CART algorithm has successfully split the training set in two, it splits the subsets using the same logic, then the sub-subsets, and so on, recursively. This implementation differs from the earlier one in two important ways:

- The SSE's are weighted by their sizes.
- We are splitting the entire predictor space; that is, we apply the splitting to both the resulting regions R_{right} and R_{left} at each internal node and not just one of the two previously identified regions.

1.3.2 Classification

For classification, the cost function is similar as that for regression tasks except that an 'impurity' measure is used:

$$f\left(j, s_j\right) = \frac{n_{R_{\text{left}}}}{n} G_{R_{\text{left}}} + \frac{n_{R_{\text{right}}}}{n} G_{R_{\text{right}}}$$

where

- $n_{R_{\text{left}}}$ and $n_{R_{\text{right}}}$ are the numbers of instances in each of the subsets or regions
- n is the total number of instances or trials in the training set
- The G's represent the Gini impurity index.

Gini Impurity Index

To compute Gini impurity index for a node or subset or region, R, with k classes $i \in 1, 2, \ldots, k\}$, first let $p_{R,k}$ be the proportion of class k instances or trials in the node or subset or region R:

$$p_{R,k} = \frac{1}{n_R} \sum_{x_i \in R} I(x_i = k)$$

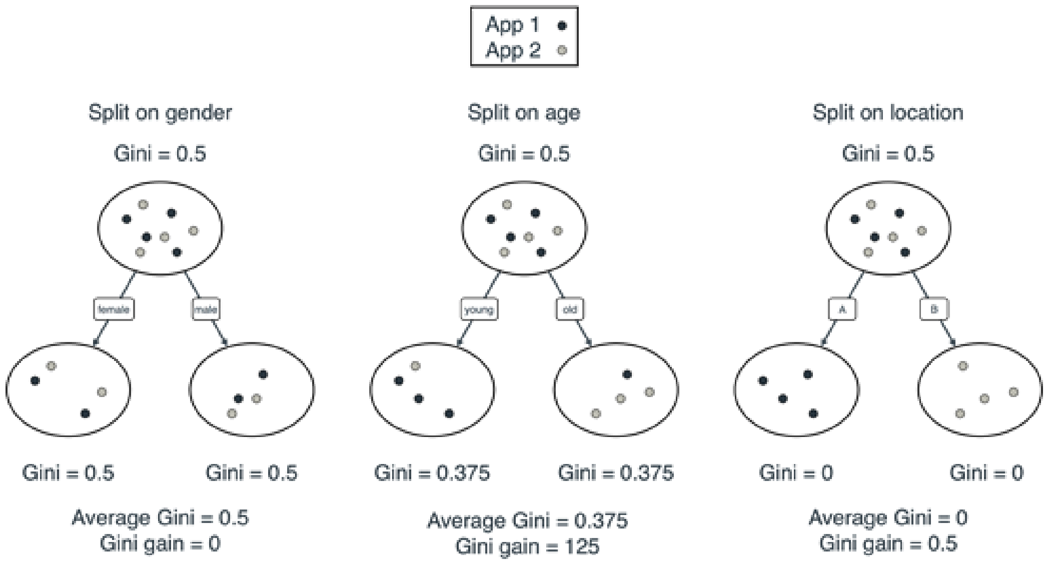
where

- n_R is the number of instances in the node or subset or region
- $\sum_{x_i \in R} I(x_i = k)$ is the number (count) of class k instances or trials in node R

Then, the Gini impurity index of region R with k classes $i \in 1, 2, \ldots, k\}$ is defined as follows:

$$\begin{aligned}
G_R &= \sum_k p_{R,k}\,(1 - p_{R,k}) \\
&= \sum_k (p_{R,k} - p_{R,k}^2) \\
&= \sum_k p_{R,k} - \sum_k p_{R,k}^2 \\
&= 1 - \sum_k p_{R,k}^2 \\
&= 1 - (p_{R,1}^2 + p_{R,2}^2 + \ldots + p_{R,k}^2) \\
&= 1 - p_{R,1}^2 - p_{R,2}^2 - \ldots - p_{R,k}^2 \\
&= 1 - (\text{prop of class 1 trials in } R)^2 - (\text{prop of class 2 trials in } R)^2 - \ldots - (\text{prop of class } k \text{ trials in } R)^2
\end{aligned}$$

Generally, the lower the gini index the lower probability of misclassifying an observation. This is because Gini impurity is a measurement of the likelihood of an misclassification of an instance, if that instance were randomly classified according to the proportions of class labels in a node or region or subset. Gini impurity is lower bounded by 0, which means that the region contains only one class.



Entropy

Another measure of impurity is **entropy**, which is a mathematical form that measures of heterogeneity or impurity in a node or leaf or region or subset. The formula is as follows:

$$E_R = -\sum_k p_{R,k} \log\left(p_{R,k}\right)$$

where $p_{R,k}$ is again the proportion of class k instances in region R. Entropy is derived from the expected value of surprises, where surprise for an event, A, is defined as:

$$S = \log(\frac{1}{p_A})$$

where p_A is the probability that event A happens. So the surprise of an event has an inverse relationship with the probability of the event happening. That is, the lower the probability, the larger the surprise; the higher the probability of an event happening, the lower the surprise. The expected value of surprise is then the weighted sum of the form:

$$E(S) = \sum_i x_i \cdot p(X = x_i)$$

where x is the value of surprise and $p(X = x)$ is the probability of observing that value of surprise. Plugging in S for x in the equation above:

$$\begin{aligned} E(S) &= \sum_i \log(\frac{1}{p_i})p(i) \\ &= \sum_i p(i)[\log(1) - \log(p_i)] \\ &= \sum_i p(i)[0 - \log(p_i)] \\ &= \sum_i -p(i) \cdot \log(p_i)] \\ &= -\sum_i p(i) \cdot \log(p_i)] \end{aligned}$$

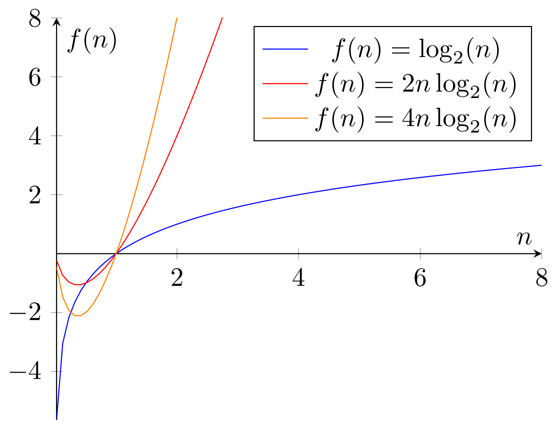
This is the form seen above.

1.4 Computational Complexity

Making predictions for an instance requires traversing the decision tree from the root to a particular leaf or terminal node. Decision Trees generally are approximately balanced, so traversing the decision tree requires going through roughly $O(\log_2(n))$ (log_2 is the binary log) nodes where n is the number of trials or instances in the training set.

$$x = \log_2 n \iff 2^x = n$$

Since each internal node only requires checking the value of one feature, the overall **prediction complexity** is $O(\log_2(n))$, independent of the number of features p. So predictions are very fast, even when dealing with large training sets. The training algorithm, on the other hand, compares all features on all n samples at each internal node. Comparing all features on all samples at each node results in a **training complexity** of $O(p \times n\log_2(n))$. This added complexity $p \times n$ is due to the fact that, at each internal node, we need to check all n trials (or instances or samples or rows) for each of the p features. The run times will increase rapidly when there are many features. The computational complexity can be visualized as follows:



The depth (i.e., the number of children nodes) of a well-balanced binary tree containing n leaves or terminal nodes is equal to $\log_2(n)$, rounded up. A binary Decision Tree (one that makes only binary decisions, as is the case with all trees in Scikit-Learn) will end up more or less well balanced at the end of training, with one leaf or terminal per training instance if it is trained without restrictions. As an example, if the training set contains one million instances, the Decision Tree will have a depth of $\log_2(10^6) = 20$ (in reality, this number may be a bit higher since the tree may not be perfectly well balanced).