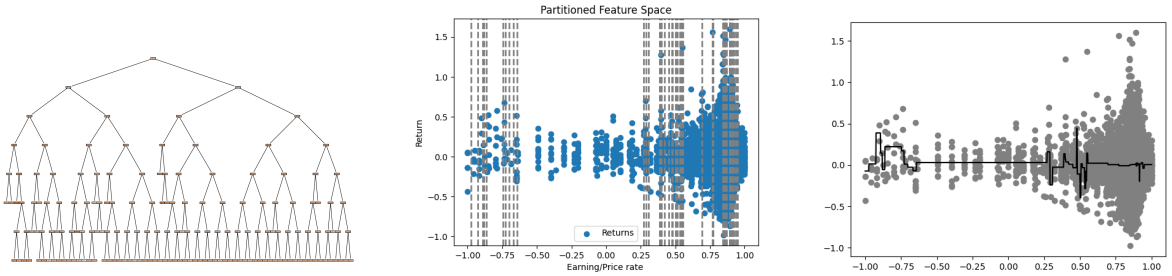
# 9 Regression Trees

## 9.1 Regression Trees

Regression trees are essentially recursive binary partitions on some feature space $X$ resulting in a piece-wise prediction. That is, Trees are equivalent to recursively partitioning the feature space into two parts each time until obtaining different partitions, each corresponding to a constant prediction [81] .In a 2 dimensional feature space this equivalence between building a tree and partitioning a feature space can be clearly illustrated.




((a)) Regression Tree. ((b)) Partitioned Feature Space. ((c)) Constant Piecewise Regression.

Figure 12: Regression Trees can be interpreted in three ways: as trees, as a feature space partitioning, or as a constant piecewise regression function. I use my dataset to provide a visual representation using the Earnings to Price Ratio as the factor.

In fact, one can either, perform a recursive binary partitioning on some rectangular ( Two dimensional ) feature space: That is, I start with the first split , let's say in $X_1 = s_1$ , then in each of the two defined regions, I split again. Thus in $X_1 > s_1$ I split at $X_1 = s_2$ and in $X_1 < s_1$ I split at $X_1 = s_3$ and I repeat the process recursively, until obtaining the desired partition. The resulting feature space is split in $K$ regions, each corresponding to a constant prediction output. [82] Or equivalently, one can represent the same result in a tree: the leafs (i.e. the terminal nodes) represent the last partitioned regions in the feature space, the binary nodes represent the different binary splitting decisions, and the output of each leaf corresponds to the constant prediction output of each partitioned region.

Formally, we represent regression trees as such: For some design matrix $X$ and a dependent variable $Y$; if we have $K$ ;$R_1, ..., R_K$; partitioned regions with each a corresponding piece wise constant prediction $c_K$, one can define a regression tree model as such : $f(x) = \sum_{k=1}^{k} c_k I\left(x \in R_k\right)$ . The model is thus characterized by two defining parameters: the partitioned regions and the associated constant prediction output.

Ideally, one would like to find the partitioning that minimizes the squared loss between the observed

[81] A recursive program is one in which a function, such as Binary Partition, relies on prior, simplified instances of itself.
[82] Notice, however that I still have not discussed the choice of the splitting parameters and the splitting point (I explain this below).

and the predicted outputs. Two facts emerge, first , knowing the partitioned regions $R_k$, our model prediction is the average of the dependent variables in the partition. That is, $Agmin_{c_k}(y_i - f(x_i))^2 \Rightarrow \hat{c}_k = \text{ave}(y_i \mid x_i \in R_k)$ [83]. Secondly, it is computationally unrealistic to find such a partition, as it involves evaluating the loss for all possible partitions,i.e. comparing resulting trees from every conceivable split and ordering.

Accordingly, I perform a greedy algorithm (Introduced by Breiman,1984 [2]) to solve this problem. Instead of considering all possible splits, the greedy algorithm, focuses only on a single partition: Explicitly, our problem reduces to finding the best partitioning feature $X_j$ and point $s$ that splits the region into two sub regions $R_1(j,s) = \{X \mid X_j \leq s\}$ and $R_2(j,s) = \{X \mid X_j > s\}$ - Using a squared loss function, we determine the parameters as such :

$$Argmin_{X_j,s}\left[\min_{c_1}\sum_{x_i \in R_1(j,s)}(y_i - c_1)^2 + \min_{c_2}\sum_{x_i \in R_2(j,s)}(y_i - c_2)^2\right]$$

Accordingly, $\hat{c}_1 = \text{ave}(y_i \mid x_i \in R_1(j,s))$ and $\hat{c}_2 = \text{ave}(y_i \mid x_i \in R_2(j,s))$.

Constraining the problem to a single split each time makes partition now feasible by computing the squared loss - also called the impurity function [84] - over all features for all split points [85], within a single level of the tree only. This algorithm facilitates the construction of a regression tree[86]; nevertheless, it exhibits certain limitations: By constraining parameter fitting to a single level only, we risk overlooking scenarios where a sub optimal splitting feature (or splitting point) at a particular location on the tree could, in fact, contribute to the creation of a more efficient tree overall. The use of a greedy algorithm neglects the possibility that a partition considered weak at a certain level might be advantageous for the entire tree. This issue is mitigated by the introduction of ensemble methods explained in the next section .

In the greedy algorithm, the tree size serves as a hyper parameter. Setting it too high may result in overfitting, while setting it too low can lead to underfitting. [87]. Consequently, it becomes essential

[83] In fact, assuming that the partition $R_k$ is known; if $x \in R_k \rightarrow I(x \in R_k) = 1 \rightarrow f(x) = c_K \cdot 1$ Accordingly, in order to find the constant parameter in $R_k$ one need to minimize the mean squared error of the observation and the constant prediction in the assumed partition region. $\underset{c_K}{\text{Argmin}}\mathbb{E}\left[(Y_K - c_K)^2\right] = \underset{c_k}{\text{Argmin}}\left[E(Y_K - c_k)\right]^2 + V(Y_K - c_k)$ $= \underset{c_K}{\text{Argmin}}\left[\mathbb{E}(Y_K) - c_K\right]^2 + V(c_K)$, we solve this optimization problem by equating the derivative with respect to $c_k$ to zero (Since the objective function is a quadratic derivable convex function); thus, $\frac{\partial MSE(c_K)}{\partial c_K} = -2(\mathbb{E}(Y_K) - c_K) = 0$ solving this equation, we get $c_K = \mathbb{E}(Y_K)$ . Hence, the optimal output for a squared loss objective function given the partition space is the average of the dependent variables in that partition.

[84] The impurity function is a generic function that quantifies the "purity" of the split. In regression trees, loss functions are used for this purpose, while in classification trees it takes different forms, notably Misclassification error , cross entropy or Gini index etc...

[85] There are different ways to choose which splitting points we should use when evaluating different splitting scenarios. I choose $s$ values to be the different quantile values of the feature

[86] For N observations, Greedy Algorithm is $O(N^2)$ while the "Naive" infeasible partitioning is $O(2^N)$. Hence, for 100 observations , we need 10,000 operations in a Greedy Algorithm against 1.267×1030 for the naive splitting to decide on $Xj$ and $s$

[87] This is due to the fact that the tree size controls complexity, impacting generalization, approximation, and estimation

to regularize the tree size. Various methods exist to achieve this goal. One approach involves defining a maximum number of observations in the leaves, or establishing a threshold beyond which the mean squared error at each iteration should not increase. However, I choose not to employ these methods due to their myopic behavior regarding the subsequent levels of the tree. Instead, I utilize a Tree pruning approach.

That is, I construct a large tree using the greedy algorithm, prune the tree at some nodes and obtain a more performing sub tree. Pruning is the act of cutting down the regression tree at a certain node. There are different ways of approaching this method; one can, for example, perform a "Reduced Error Pruning", which consists of assessing all nodes, and calculate the cost associated with collapsing each node then proceed to collapse the nodes that result in the smallest increase in mean squared error. Note: Pruning will always increase the overall mean squared error of the tree as the tree is a non parametric method and will surely overfit the data if grown large enough (The dynamics of non parametric models are discussed in the Generalized Additive Models chapter ), this is why, we choose to prune for the smallest increase in the overall mean. The Reduced Error Method is a fast but naive method as there is no explicit penalty criterion for pruning other than overall mean squared error.

I thus adopt a more interpretable method: The "Cost-complexity pruning" method introduced by Breiman et al. (1984)[2], is a tree regularization method that first consists of building a large tree; by fixing very loose stopping criteria ( For instance, choosing a a maximum number of observations per leaf to be 1, or choosing a large depth condition... The list of methods for building a large tree is exhaustive), then finding the subtree that has the minimum mean squared error given a penalty on its number of leafs ( its terminal nodes).

Formally, I pick a subtree $T \subset T_0$; that minimizes the following Cost function

$$C_\alpha(T) = \sum_{k=1}^{|T|} L_k(T) + \alpha|T|$$

with $L_k(T) = \frac{1}{N_k}\left(y_i - \hat{c}_k\right)^2$ the mean squared error associated to the subtree; $\hat{C}_k = \frac{1}{N_k}\sum_{x_i \in R_k} y_i$, the constant prediction in Partition $k$; $|T|$ the total number of leafs of the tree and $\alpha$ the penalization tuning parameter. Cost complexity pruning involves penalizing the number of terminal nodes, and finding accordingly the best subtree. The rationale is as follows: As explained, since growing a tree invariably results in a decrease in the overall mean squared error ( as trees ultimately overfit ), we aim to select a subtree that mitigates overfitting. To achieve this, a comparison of various subtrees is necessary. Considering that a smaller tree inherently yields a higher mean squared error, the $|T|$ penalty introduces a size-related penalization to the mean squared error of each subtree. Introducing

errors. These dynamics have been discussed in the Statistical Leaning Theory chapter

this penalty term provides a dynamic method for comparing subtrees, where the cost of a smaller tree may be lower than that of a larger one, depending on the tree's size (or equivalently, its number of terminal nodes) and the penalty tuning. This method relies on cross validation: The penalization hyperparameter $\alpha$ is found by picking the most performing one on the validation set.