# Notes for ISL Chapter 8: Tree-based Methods

Justin Burruss

2025-12-27

## Background

The Python code and notes below are for the ISL study group. This is to try out some of the concepts from Chapter 8 on the `Hitters` data set. The document was created in RMarkdown with the Python code running via the `reticulate` library plus a little LaTeX.

Our ground rule: when implementing concepts from the chapter, just use basic Python + Pandas + NumPy. It's OK to use more when visualizing or evaluating results.

## Regression Trees

Let's first see if we can recreate figure 8.1 from the text—the nodes and numbers if not the polished appearance.

### `RegressionTree` class

Here's our plan for `RegressionTree` with the minimal features we need for chapter 8.

- Create the tree using the Pandas dataframe, preserving the column names
- Parameters
  - minimum leaf size
  - maximum tree depth
  - weights (so we can reuse this code later for boosting)
- Structure
  - binary
  - use `<=` and actual values for thresholds
  - save global `X` and `Y` and record global bounds
- Method to grow the tree from the X and Y
  - grow recursively
  - minimizing MSE, optionally with weights
  - use running sums with `np.cumsum()` to cut down on MSE burden at splits
- Method to predict based on new X
- String representation
  - replicate Linux `tree`
  - traverse recursively

- show splits for non-leaf nodes
- show values, counts, and MSE for leaf nodes

The book uses `<` and half-values like `4.5` where it's labeling the midpoint between adjacent values. We'll go our own way here and use `<=` and actual values—arguably this is cleaner for integer data such as `Years` and `Hits`.

We will be looking at many possible splits, each time checking weighted MSEs, so it would be best reformulate weighted MSE in such a way to make it cheap.

**Reformulating weighted MSE**

If we have our dependent variable made up of $n$ number of individual $y_i$, and we have $n$ number of individual estimates $\hat{y}_i$, then our residual sum of squares (RSS) is the sum of the squared errors:

$$\text{RSS} = \sum_{i=1}^{n}[(y_i - \hat{y}_i)^2]$$

Our mean squared error (MSE) is the arithmetic mean of RSS:

$$\text{MSE} = \frac{\sum_{i=1}^{n}[(y_i - \hat{y}_i)^2]}{n}$$

If we have $n$ number of individual weights $w_i$, then the weighted MSE with our $n$ estimates $\hat{y}_i$ becomes:

$$\text{Weighted MSE} = \frac{\sum_{i=1}^{n}[w_i(y_i - \hat{y}_i)^2]}{\sum_{i=1}^{n} w_i}$$

Let's think this through: if we're looking at a candidate split in the process of training a regression tree, needing to calculate MSE for a candidate, we're in a situation where all the candidate $\hat{y}_i$ values would be set to a single estimate $\hat{y}$ for node. The one value for $\hat{y}$ for the candidate is the weighted mean of the $y_i$ values in the candidate split, which we can call $\bar{y}_w$. Thus we'll have just the one $\hat{y}$ for all $y_i$ in the leaf, and that one $\hat{y}$ will be set to the weighted mean $\bar{y}_w$.

Our weighted mean $\bar{y}_w$ is given by:

$$\text{Weighted Mean} = \bar{y}_w = \frac{\sum_{i=1}^{n} w_i y_i}{\sum_{i=1}^{n} w_i}$$

We can use this insight to rewrite weighted MSE for a candidate split (WMSE) in terms of $\bar{y}_w$. First a little algebra to crack open the formula and decompose it into weighted variance and weighted bias.

$$\begin{aligned}
\text{WMSE} &= \frac{\sum_{i=1}^{n} w_i(y_i - \hat{y})^2}{\sum_{i=1}^{n} w_i} && \text{orignial formula above} \\
&= \frac{\sum_{i=1}^{n} w_i(y_i - \bar{y}_w + \bar{y}_w - \hat{y})^2}{\sum_{i=1}^{n} w_i} \\
&= \frac{\sum_{i=1}^{n} w_i[(y_i - \bar{y}_w)^2 + 2(y_i - \bar{y}_w)(\bar{y}_w - \hat{y}) + (\bar{y}_w - \hat{y})^2]}{\sum_{i=1}^{n} w_i} && \text{expanding} \\
&= \frac{\sum_{i=1}^{n} w_i(y_i - \bar{y}_w)^2}{\sum_{i=1}^{n} w_i} + \frac{\sum_{i=1}^{n} w_i 2(y_i - \bar{y}_w)(\bar{y}_w - \hat{y})}{\sum_{i=1}^{n} w_i} + \frac{\sum_{i=1}^{n} w_i(\bar{y}_w - \hat{y})^2}{\sum_{i=1}^{n} w_i} && \text{summation is linear}
\end{aligned}$$

Observe the numerator $\sum_{i=1}^{n} w_i 2(y_i - \bar{y}_w)(\bar{y}_w - \hat{y})$ of the middle term. We can rearrange to uncover some hidden zeroes.

$$\begin{aligned}
\sum_{i=1}^{n} w_i 2(y_i - \bar{y}_w)(\bar{y}_w - \hat{y}) &= 2\sum_{i=1}^{n} w_i (y_i - \bar{y}_w)(\bar{y}_w - \hat{y}) && \text{pull out constant} \\
&= 2\sum_{i=1}^{n} [\, w_i \bar{y}_w (y_i - \bar{y}_w) - w_i \hat{y} (y_i - \bar{y}_w) \,] \\
&= 2[\sum_{i=1}^{n} w_i \bar{y}_w (y_i - \bar{y}_w) - \sum_{i=1}^{n} w_i \hat{y} (y_i - \bar{y}_w) \,] && \text{summation is linear} \\
&= 2[\, \bar{y}_w \sum_{i=1}^{n} w_i (y_i - \bar{y}_w) - \hat{y} \sum_{i=1}^{n} w_i (y_i - \bar{y}_w) \,] && \text{pull out constants}
\end{aligned}$$

The $\sum_{i=1}^{n} w_i (y_i - \bar{y}_w)$ we recognize as the weighted deviations from the weighted mean, but that's just zero:

$$\begin{aligned}
\sum_{i=1}^{n} w_i (y_i - \bar{y}_w) &= \sum_{i=1}^{n} w_i y_i - \sum_{i=1}^{n} w_i \bar{y}_w \\
&= \sum_{i=1}^{n} w_i y_i - \bar{y}_w \sum_{i=1}^{n} w_i && \text{pull out constant} \\
&= \sum_{i=1}^{n} w_i y_i - \frac{\sum_{i=1}^{n} w_i y_i}{\sum_{i=1}^{n} w_i} \sum_{i=1}^{n} w_i && \text{definition of } \bar{y}_w \\
&= \sum_{i=1}^{n} w_i y_i - \sum_{i=1}^{n} w_i y_i && \text{cancellation} \\
&= 0 && \text{cancellation}
\end{aligned}$$

Plug that back in to $2[\, \bar{y}_w \sum_{i=1}^{n} w_i (y_i - \bar{y}_w) - \hat{y} \sum_{i=1}^{n} w_i (y_i - \bar{y}_w) \,]$ and we find that it is zeroed out:

$$\begin{aligned}
2[\, \bar{y}_w \sum_{i=1}^{n} w_i (y_i - \bar{y}_w) - \hat{y} \sum_{i=1}^{n} w_i (y_i - \bar{y}_w) \,] &= 2[\, \bar{y}_w \cdot 0 - \hat{y} \cdot 0 \,] && \text{plugging in result from above} \\
&= 0
\end{aligned}$$

That means our whole middle term is zeroed out. Continuing from where we left off with WMSE:

$$\begin{aligned}
\text{WMSE} &= \frac{\sum_{i=1}^{n} w_i (y_i - \bar{y}_w)^2}{\sum_{i=1}^{n} w_i} + \frac{\sum_{i=1}^{n} w_i 2(y_i - \bar{y}_w)(\bar{y}_w - \hat{y})}{\sum_{i=1}^{n} w_i} + \frac{\sum_{i=1}^{n} w_i (\bar{y}_w - \hat{y})^2}{\sum_{i=1}^{n} w_i} && \text{continuing} \\
&= \frac{\sum_{i=1}^{n} w_i (y_i - \bar{y}_w)^2}{\sum_{i=1}^{n} w_i} + \frac{0}{\sum_{i=1}^{n} w_i} + \frac{\sum_{i=1}^{n} w_i (\bar{y}_w - \hat{y})^2}{\sum_{i=1}^{n} w_i} && \text{plug in result from above} \\
&= \frac{\sum_{i=1}^{n} w_i (y_i - \bar{y}_w)^2}{\sum_{i=1}^{n} w_i} + \frac{\sum_{i=1}^{n} w_i (\bar{y}_w - \hat{y})^2}{\sum_{i=1}^{n} w_i} && \text{middle term is zeroed out}
\end{aligned}$$

Now we have just a left term and a right term. Our left term we can recognize as being weighted variance, the right as weighted bias.

First, take a look at the numerator of the right term. We have a difference between $\bar{y}_w$, which is a constant defined as the weighted mean of the $y_i$ values we're considering for our candidate split, and $\hat{y}$, which is a constant that we've selected as our estimator for the $y_i$ values in our leaf. But we want to choose an unbiased

estimator, i.e., to set $\hat{y} = \bar{y}_w$. Look what happens when we choose an unbiased estimator by setting $\hat{y} = \bar{y}_w$:

$$\begin{aligned}
\text{Weighted Bias} &= \frac{\sum_{i=1}^{n} w_i(\bar{y}_w - \hat{y})^2}{\sum_{i=1}^{n} w_i} && \text{continuing} \\
&= \frac{\sum_{i=1}^{n} w_i(\bar{y}_w - \bar{y}_w)^2}{\sum_{i=1}^{n} w_i} && \text{setting } \hat{y} = \bar{y}_w \\
&= \frac{\sum_{i=1}^{n} w_i(0)^2}{\sum_{i=1}^{n} w_i} && \bar{y}_w - \bar{y}_w = 0 \\
&= \frac{0}{\sum_{i=1}^{n} w_i} \\
&= 0
\end{aligned}$$

So in choosing an unbiased estimator $\hat{y} = \bar{y}_w$ our right term gets zeroed out. This leaves just the left term: the weighted variance of our candidate. There's no $\hat{y}$ to be seen: it's independent of our choice of $\hat{y}$. Instead, it follows from our choice of $y_i$ values to include in the leaf. It would be zero if we built a leaf having just one distinct value for all $y_i$ in the leaf.

The upshot is by setting $\hat{y} = \bar{y}_w$ and end up with the following:

$$\text{WMSE} = \frac{\sum_{i=1}^{n}[w_i(y_i - \hat{y}_i)^2]}{\sum_{i=1}^{n} w_i} = \frac{\sum_{i=1}^{n}[w_i(y_i - \bar{y}_w)^2]}{\sum_{i=1}^{n} w_i}$$

Now let's shape this into our final form for the "grow the tree" for loop. We will introduce some variables for readability.

Let:

$$S_w = \sum_{i=1}^{n} w_i$$

$$S_{wy} = \sum_{i=1}^{n} w_i y_i$$

$$S_{wy^2} = \sum_{i=1}^{n} w_i y_i^2$$

Now we can express $\bar{y}$ as:

$$\begin{aligned}
\bar{y}_w &= \frac{\sum_{i=1}^{n} w_i y_i}{\sum_{i=1}^{n} w_i} \\
&= \frac{S_{wy}}{S_w} && \text{sub in the new variables for readability}
\end{aligned}$$

We further develop WMSE:

$$
\begin{aligned}
\text{WMSE} &= \frac{\sum_{i=1}^n w_i(y_i - \bar{y}_w)^2}{\sum_{i=1}^n w_i} && \text{continuing from where we left off}\\
&= \frac{\sum_{i=1}^n w_i(y_i^2 - 2y_i\bar{y}_w + \bar{y}_w^2)}{\sum_{i=1}^n w_i} && (a-b)^2 = a^2 - 2ab + b^2\\
&= \frac{\sum_{i=1}^n w_i y_i^2}{\sum_{i=1}^n w_i} - \frac{\sum_{i=1}^n w_i 2y_i\bar{y}_w}{\sum_{i=1}^n w_i} + \frac{\sum_{i=1}^n w_i\bar{y}_w^2}{\sum_{i=1}^n w_i} && \text{summation is linear}\\
&= \frac{\sum_{i=1}^n w_i y_i^2}{\sum_{i=1}^n w_i} - 2\bar{y}_w\frac{\sum_{i=1}^n w_i y_i}{\sum_{i=1}^n w_i} + \frac{\sum_{i=1}^n w_i\bar{y}_w^2}{\sum_{i=1}^n w_i} && \text{pull out constant } 2\bar{y}_w\\
&= \frac{\sum_{i=1}^n w_i y_i^2}{\sum_{i=1}^n w_i} - 2\bar{y}_w\frac{\sum_{i=1}^n w_i y_i}{\sum_{i=1}^n w_i} + \bar{y}_w^2\frac{\sum_{i=1}^n w_i}{\sum_{i=1}^n w_i} && \text{pull out constant } \bar{y}_w^2\\
&= \frac{S_{wy^2}}{S_w} - 2\bar{y}_w\frac{S_{wy}}{S_w} + \bar{y}_w^2\frac{S_w}{S_w} && \text{sub in for the summations}\\
&= \frac{S_{wy^2}}{S_w} - 2\bar{y}_w\frac{S_{wy}}{S_w} + \bar{y}_w^2 && \text{cancellation}\\
&= \frac{S_{wy^2}}{S_w} - 2\frac{S_{wy}}{S_w}\frac{S_{wy}}{S_w} + (\frac{S_{wy}}{S_w})^2 && \text{sub in for } \bar{y}_w\\
&= \frac{S_{wy^2}}{S_w} - 2(\frac{S_{wy}}{S_w})^2 + (\frac{S_{wy}}{S_w})^2\\
&= \frac{S_{wy^2}}{S_w} - (\frac{S_{wy}}{S_w})^2 && \text{final form}
\end{aligned}
$$

This will make for a better implementation. For each candidate feature, if we first sort, we can then use `np.cumsum()` to calculate all our candidate $S_w$, $S_{wy}$, and $S_{wy^2}$ just once before looping over and checking each possible split location: as we slide along left to right we simply look up our already-calculated weights & sums.

### Coding

As always, we'll stick with basic Python + Pandas + NumPy for the implementation.

NumPy already has a function for doing weighted means: `np.average()`.

In `__split` as as we slide along left to right our `[i-1]` is for the left, `[i]` for the right. We scan elements from leftmost candidate index `self.min_cnt_leaf` up through rightmost candidate index `len(y) - self.min_cnt_leaf + 1`, i.e., last element plus one minus our minimum leaf count. To illustrate, if `self.min_cnt_leaf=5`, then our first candidate is checking elements $0, 1, 2, 3, 4$ as left and remaining as the right.

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.lines as lines
import matplotlib.patches as patches
import matplotlib.colors as mcolors

class RegressionTree:
    """Class to build regression trees from Pandas dataframes"""
```

```
    class Node:
        """Data structure for nodes of the regression tree"""
        def __init__(self,
                     feature: str | None = None,
                     threshold: float | None = None,
                     left = None,
                     right = None,
                     value: float | None = None,
                     cnt: int | None = None,
                     mse: float | None = None):
          self.feature = feature
          self.threshold = threshold
          self.left = left
          self.right = right
          self.value = value
          self.cnt = cnt
          self.mse = mse

    def __init__(self,
                 X: pd.DataFrame = None,
                 Y: pd.Series = None,
                 max_depth: int = 2,
                 min_cnt_leaf: int = 50,
                 weights: np.ndarray | None = None):
        self.X = X
        self.Y = Y
        self.root = None
        self.max_depth = max_depth
        self.min_cnt_leaf = min_cnt_leaf
        self.weights = np.ones(len(Y)) if weights is None else weights
        self.bounds = {c: (X[c].min(), X[c].max()) for c in X.columns}

    def fit(self,
            X: pd.DataFrame = None,
            Y: pd.Series = None,
            weights: np.ndarray | None = None) -> None:
        """Grow the tree to fit the X and Y"""
        new_data = False
        if X is not None:
            self.X = X
            new_data = True
        if Y is not None:
            self.Y = Y
            new_data = True
        if weights is not None:
            self.weights = weights
        elif new_data or self.weights is None:
            self.weights = np.ones(len(self.Y))
        self.root = self.__grow(self.X, self.Y, w=self.weights)

    def predict(self, X_new: pd.DataFrame) -> pd.Series:
        """Predict new Y given new X rows"""
        Y_new = X_new \
```

```
            .apply(lambda row: self.__predict_row(row, self.root), axis=1) \
            .rename(self.Y.name)
        return Y_new

    def __str__(self):
        if self.root is None:
            return "RegressionTree (empty)"
        return self.__as_text(self.root, prefix="", is_last=True, depth=0)

    def __as_text(self, node, prefix="", is_last=True, depth=0):

        # line prefixed with edge symbol when applicable
        if depth == 0:
            # root node: no prefix
            line = ""
        elif depth == 1:
            # children of root: prefix with edge symbol
            line = ("└──" if is_last else "├──")
        else:
            # children of children: use prefix from recursive call plus edge
            line = prefix + ("└──" if is_last else "├──")

        # leaf nodes: display value, count, and MSE; no branches to traverse
        if node.value is not None:
            value = f"{node.value:.2f}"
            cnt = f"{node.cnt:.0f}" if node.cnt is not None else "NA"
            mse = f"{node.mse:.2f}" if node.mse is not None else "NA"
            return line + f"[Leaf: {value} Cnt: {cnt} MSE: {mse}]\n"

        # non-leaf nodes: display split and traverse branches
        s = line + f"[{node.feature} <= {node.threshold}]\n"

        # prefix for children
        if depth == 0:
            child_prefix = ""
        elif depth == 1:
            child_prefix = "│   " if not is_last else "    "
        else:
            child_prefix = prefix + ("│   " if not is_last else "    ")

        # recursively grow the string
        s += self.__as_text(node.left,  child_prefix, False, depth+1)
        s += self.__as_text(node.right, child_prefix, True,  depth+1)
        return s

    def __mse(self, y: np.ndarray | pd.Series, w: np.ndarray | None = None):
        if w is None:
            return np.mean((y - y.mean())**2)
        ybar = np.average(y, weights=w)
        return np.average((y - ybar)**2, weights=w)

    def __split(self,
                X: pd.DataFrame,
```

```
        Y: pd.Series,
        w: np.ndarray) -> tuple[str | None, float | None, float | None]:
    """Attempt to split into left and right using MSE"""

    # faster if we use NumPy when splitting
    y = Y.to_numpy()

    # start with parent MSE, i.e., MSE before any split into left and right
    sum_w = w.sum()
    sum_wy = np.sum(w * y)
    sum_wy2 = np.sum(w * y**2)
    parent_mse = sum_wy2 / sum_w - (sum_wy / sum_w)**2

    best_feature = None
    best_threshold = None
    best_mse = parent_mse

    # check each possible feature
    for feature in X.columns:

        # faster if we use NumPy when splitting
        x = X[feature].to_numpy()

        # sort once for quicker scans of candidate splits later
        order = np.argsort(x)
        x_s = x[order]
        y_s = y[order]
        w_s = w[order]

        # cumulative sums to cut down on MSE recalcs at candidate splits
        w_c = np.cumsum(w_s)
        wy_c = np.cumsum(w_s * y_s)
        wy2_c = np.cumsum(w_s * y_s**2)

        # check each possible split point
        for i in range(self.min_cnt_leaf, len(y) - self.min_cnt_leaf + 1):

            # skip duplicate feature values
            if x_s[i] == x_s[i - 1]:
                continue

            # look up our already-calculated weights & sums
            w_left = w_c[i-1]
            w_right = sum_w - w_left
            wy_left = wy_c[i-1]
            wy2_left = wy2_c[i-1]

            # find right weighted y and weighted y squared
            wy_right  = sum_wy  - wy_left
            wy2_right = sum_wy2 - wy2_left

            # weighted MSEs for candidate left and right
            mse_left  = wy2_left  / w_left  - (wy_left  / w_left )**2
```

```
            mse_right = wy2_right / w_right - (wy_right / w_right)**2

            # candidate MSE
            mse = (w_left * mse_left + w_right * mse_right) / sum_w

            if mse < best_mse:
                best_mse = mse
                best_feature = feature
                best_threshold = x_s[i-1]

    if best_feature is None:
        return None, None, None

    return best_feature, best_threshold, best_mse

def __grow(self,
           X: pd.DataFrame,
           Y: pd.Series,
           depth: int = 0,
           w: np.ndarray | None = None):
    """Recursively grow the tree"""

    if w is None:
        w = np.ones(len(Y))

    # two stopping criteria: depth or sample count
    if (depth >= self.max_depth or
        len(Y) < 2*self.min_cnt_leaf):
        return self.Node(value=np.average(Y, weights=w),
                         cnt=len(Y),
                         mse=self.__mse(Y, w))

    # attempt a split
    feature, threshold, mse = self.__split(X, Y, w)

    # no split: build and return a leaf with value, count, and MSE
    if feature is None:
        return self.Node(value=np.average(Y, weights=w),
                         cnt=len(Y),
                         mse=mse)

    # split success
    left_mask = X[feature] <= threshold
    right_mask = ~left_mask

    # recurse only if we have sufficient samples, otherwise return leaf
    if left_mask.sum() < self.min_cnt_leaf or right_mask.sum() < self.min_cnt_leaf:
        return self.Node(value=np.average(Y, weights=w),
                         cnt=len(Y),
                         mse=mse)

    # build left and recurse
    left_child = self.__grow(X[left_mask], Y[left_mask], depth+1, w=w[left_mask])
```

```
        # build right and recurse
        right_child = self.__grow(X[right_mask], Y[right_mask], depth+1, w=w[right_mask])

        # return non-leaf node
        return self.Node(feature=feature,
                         threshold=threshold,
                         left=left_child,
                         right=right_child)

    def __predict_row(self,
                      row: pd.Series,
                      node: "RegressionTree.Node") -> float:
        if node.value is not None:
            return node.value
        if row[node.feature] <= node.threshold:
            return self.__predict_row(row, node.left)
        else:
            return self.__predict_row(row, node.right)
```

To replicate the splitting approach used in the text having `<` and half-values like `4.5`, just swap out `best_threshold = x_s[i-1]` for `best_threshold = 0.5 * (x_s[i-1] + x_s[i])` to return the midpoint between the rightmost left value and leftmost right value.

### Testing

Now let's validate against figure 8.1. First we load the data using Pandas, taking care to follow the book approach of using log salary.

```
# read CSV, dropping a few incomplete records
c = pd.read_csv(r"E:\docs\Classes\ISL\Hitters.csv").dropna()
Y = np.log(c['Salary'])
X = c[['Hits', 'Years']]
```

Then we fit and print. We're expecting a top level split on `Years < 4.5`, i.e., `Years <= 4`, a right branch on `Hits < 117.5`, i.e., `Hits <= 117`, and leaf values of 5.11, 6.00, and 6.74.

```
tree = RegressionTree(X, Y)
tree.fit()
print(tree)
```

```
## [Years <= 4]
## ├──[Leaf: 5.11 Cnt: 90 MSE: 0.47]
## └──[Hits <= 117]
##    ├──[Leaf: 6.00 Cnt: 90 MSE: 0.31]
##    └──[Leaf: 6.74 Cnt: 83 MSE: 0.25]
```

We get the same thresholds and values as the book.

### Visualizing Regions

Figure 8.2 has a clean 2-D visualization of the regions defined by the tree from figure 8.1. Let's define a function to extract the bounds for a tree with two `X` columns, then use matplotlib to visualize.

First the bounds function. We defined our bounds with a dictionary in the tree, which here means we cut down on variable names. Let's go with another recursive tree traversal here.

```
def get_regions(node: "RegressionTree.Node",
                bounds: dict[str, tuple[float, float]],
                x_axis: str = 'Years',
                y_axis: str = 'Hits'):
    """Traverse tree to build list of regions"""
    if node.value is not None:
        return [(bounds.copy(), node.value)]

    left_bounds = bounds.copy()
    right_bounds = bounds.copy()

    if node.feature == x_axis:
        low, high = bounds[x_axis]
        left_bounds[x_axis] = (low, node.threshold)
        right_bounds[x_axis] = (node.threshold, high)
    elif node.feature == y_axis:
        low, high = bounds[y_axis]
        left_bounds[y_axis] = (low, node.threshold)
        right_bounds[y_axis] = (node.threshold, high)

    R = []
    R += get_regions(node.left, left_bounds, x_axis, y_axis)
    R += get_regions(node.right, right_bounds, x_axis, y_axis)
    return R
```

Time to validate. We should have three regions with bounds that tie with the `Years=4` and `Hits=117` thresholds we printed earlier.

```
R = get_regions(tree.root, tree.bounds, 'Years', 'Hits')
for i, r in enumerate(R, start=1):
    print(f"R{i}: {r}")
```

```
## R1: ({'Hits': (1, 238), 'Years': (1, 4)}, 5.1067896059973705)
## R2: ({'Hits': (1, 117), 'Years': (4, 24)}, 5.9983798474087635)
## R3: ({'Hits': (117, 238), 'Years': (4, 24)}, 6.739686922104511)
```

Everything ties.

Now let's visualize the regions. We can use `matplotlib.lines` to draw inner boundaries. To remove ambiguity around data points on the boundary, we'll shift the inner boundary lines a half step. We'll label each region with a region number and the expected log `Wages` value. The `f"$R_{{{i}}}$\n${value:.1f}$"` f-string in the code below uses `{{` and `}}` to indicate a literal `{` and `}` for math mode, and the `{i}` part is to substitute in the `i` value, hence the triple curly braces.

```
fig, ax = plt.subplots(figsize=(5, 5))

# de-emphasize individual points
ax.scatter(tree.X['Years'], tree.X['Hits'], facecolor='gray', alpha=0.6, s=12)

values = np.array([value for _, value in R])
```

```
for i, (bounds, value) in enumerate(R, start=1):
    x0, x1 = bounds['Years']
    y0, y1 = bounds['Hits']

    # inner boundaries for horizontal, drawn just above boundary
    if y1 < tree.X['Hits'].max():
        ax.add_line(lines.Line2D([x0+0.5 if x0 > 1 else 0, x1+0.5],
                                 [y1+0.5, y1+0.5],
                                 color='black', linewidth=1))

    # inner boundaries for vertical, drawn just right of boundary
    if x1 < tree.X['Years'].max():
        ax.add_line(lines.Line2D([x1+0.5, x1+0.5],
                                 [y1+0.5, y0+0.5],
                                 color='black', linewidth=1))

    # label each region in the center
    ax.text(np.mean((x0, x1)), np.mean((y0, y1)),
            f"$R_{{{i}}}$\n${value:.1f}$",  ha='center', va='center',
            fontsize=12, color='black',
            bbox=dict(facecolor='white', edgecolor='lightgrey', alpha=0.8))

# emphasize boundaries: only put ticks where we have boundaries
ax.set_xticks(sorted({b for bounds, _ in R for b in bounds['Years']}))
ax.set_yticks(sorted({b for bounds, _ in R for b in bounds['Hits']}))

# suppress borders of plot area
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['left'].set_visible(False)
ax.spines['bottom'].set_visible(False)

plt.xlabel('Years')
plt.ylabel('Hits')
plt.title('Partition of Hitters Data')
plt.show()
```


This matches what we see in the text.

### More Regions

Let's test these display methods on a more complex tree.

```
tree = RegressionTree(X, Y, max_depth=5, min_cnt_leaf=15)
tree.fit()
print(tree)
```

```
## [Years <= 4]
## ├──[Years <= 3]
## │  ├──[Hits <= 113]
## │  │  ├──[Hits <= 82]
## │  │  │  ├──[Leaf: 4.69 Cnt: 28 MSE: 0.57]
## │  │  │  └──[Leaf: 4.80 Cnt: 15 MSE: 0.07]
## │  │  └──[Leaf: 5.26 Cnt: 19 MSE: 0.11]
## │  └──[Leaf: 5.58 Cnt: 28 MSE: 0.36]
## └──[Hits <= 117]
##    ├──[Years <= 6]
##    │  ├──[Leaf: 5.69 Cnt: 26 MSE: 0.28]
##    │  └──[Hits <= 70]
```

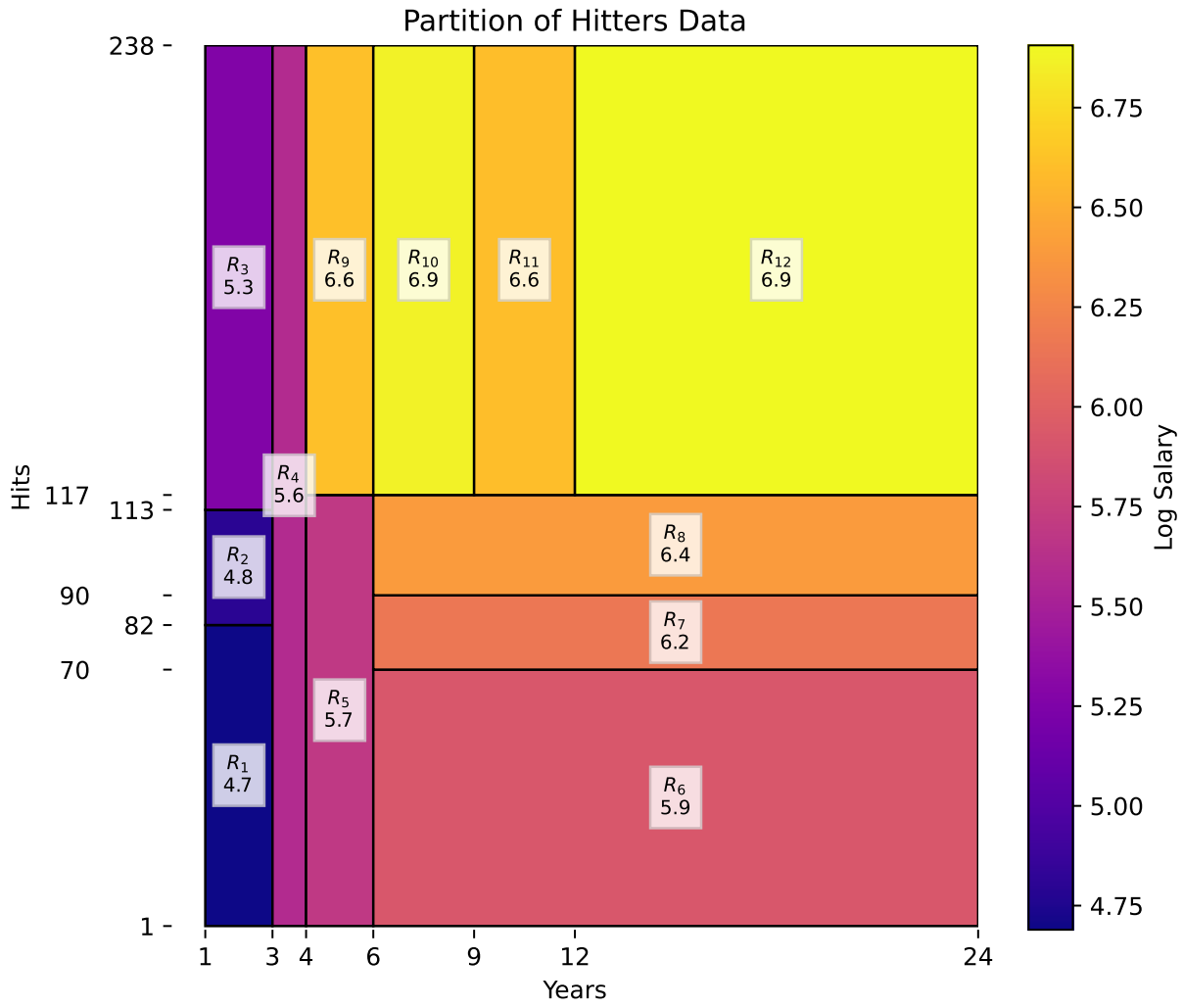
```
##     |     ├──[Leaf: 5.92 Cnt: 28 MSE: 0.27]
##     |     └──[Hits <= 90]
##     |        ├──[Leaf: 6.15 Cnt: 17 MSE: 0.14]
##     |        └──[Leaf: 6.39 Cnt: 19 MSE: 0.26]
##     └──[Years <= 12]
##        ├──[Years <= 6]
##        |  ├──[Leaf: 6.61 Cnt: 28 MSE: 0.32]
##        |  └──[Years <= 9]
##        |     ├──[Leaf: 6.87 Cnt: 19 MSE: 0.11]
##        |     └──[Leaf: 6.60 Cnt: 15 MSE: 0.28]
##        └──[Leaf: 6.91 Cnt: 21 MSE: 0.18]
```

Still readable.

Now the viz: we'll shift focus to the regions by doing away with the scatter plot, coloring the regions, and adding a colorbar for the region values. Jittering the y-axis tick labels will prevent overlap for close boundaries. Functionality in `matplotlib.cm` makes building the colormap fairly straightforward.

```
R = get_regions(tree.root, tree.bounds, 'Years', 'Hits')

fig, ax = plt.subplots(figsize=(7, 6))

# normalize our region values and construct a color scale
values = np.array([value for _, value in R])
norm = mcolors.Normalize(vmin=values.min(), vmax=values.max())
cmap = plt.cm.plasma
sm = plt.cm.ScalarMappable(cmap=cmap, norm=norm)

for i, (bounds, value) in enumerate(R, start=1):
    x0, x1 = bounds['Years']
    y0, y1 = bounds['Hits']

    # shaded area with a narrow black boundary line
    rect = patches.Rectangle((x0, y0), x1-x0, y1-y0, edgecolor='black',
                             lw=1, facecolor=cmap(norm(value)))
    ax.add_patch(rect)

    # label each region in the center
    ax.text(np.mean((x0, x1)), np.mean((y0, y1)),
            f"$R_{{{i}}}$\n${value:.1f}$",  ha='center', va='center',
            fontsize=8, color='black',
            bbox=dict(facecolor='white', edgecolor='lightgrey', alpha=0.8))

# only place tick marks on boundary values
ax.set_xticks(sorted({b for bounds, _ in R for b in bounds['Years']}))
ax.set_yticks(sorted({b for bounds, _ in R for b in bounds['Hits']}))

# jitter y-tick labels a little to accommodate close boundaries
for i, label in enumerate(ax.get_yticklabels()):
    label.set_x(0 if i % 2 == 0 else -0.08)

# suppress borders of plot area
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
```

```
ax.spines['left'].set_visible(False)
ax.spines['bottom'].set_visible(False)

# color bar
cbar = plt.colorbar(sm, ax=ax)
cbar.set_label('Log Salary')

plt.xlabel('Years')
plt.ylabel('Hits')
plt.title('Partition of Hitters Data')
plt.tight_layout()
plt.show()
```




Even with just 12 regions it can be hard to squeeze things in.

# Boosting with Algorithm 8.2

We can experiment with boosting using algorithm 8.2 as a starting place. We already have a `RegressionTree` class, so let's compose a `BoostedTrees` class that builds ensembles out of those trees.

## `BoostedTrees` class

Here's our plan for `BoostedTrees` with the minimal features we need for chapter 8.

- Uses `RegressionTree` for individual trees
- Parameters
  - number of boosts
  - learning rate (lambda is a reserved word in Python, use another name)
  - `RegressionTree` parameters for minimum leaf size and maximum tree depth
- Method to fit an ensemble
  - initial value defaults to algorithm 8.2 value but permit a quick start
- Method to predict based on new X
  - stick with `np.ndarray` this time for less overhead
- Method to get the MSE by boost for plotting later

### Coding

We'll store our individual trees in a list. The book would have us initialize $\hat{f}(x) = 0$ and $r_i = y_i$, and we will code for that exact algorithm, but let's also permit a quicker start using the mean $Y$ value.

```
class BoostedTrees:
    """Class to build ensemble of RegressionTrees using Boosting"""

    def __init__(self,
                 boost_cnt: int = 5,
                 learning_rate: float = 0.1,
                 max_depth: int = 1,
                 min_cnt_leaf: int = 50):
        self.boost_cnt = boost_cnt
        self.learning_rate = learning_rate
        self.max_depth = max_depth
        self.min_cnt_leaf = min_cnt_leaf
        self.trees = []
        self.init_value = 0.0

    def fit(self,
            X: pd.DataFrame,
            Y: pd.Series,
            quick_start: bool = False) -> None:
        """Grow an ensemble of boosted trees"""

        # initial fit using zero (per text) or simply taking mean Y
        self.init_value = Y.mean() if quick_start else 0.0
        y_hat = np.full(len(Y), self.init_value)
```

```
        self.trees = []

        for _ in range(self.boost_cnt):

            # residuals from last boost
            r = Y.values - y_hat

            # fit next tree on prior residuals
            tree = RegressionTree(
                X,
                pd.Series(r, index=Y.index),
                max_depth=self.max_depth,
                min_cnt_leaf=self.min_cnt_leaf
            )
            tree.fit()

            # update predictions
            update = tree.predict(X).values
            y_hat += self.learning_rate * update

            self.trees.append(tree)

    def predict(self, X_new: pd.DataFrame) -> np.ndarray:
        """Predict new Y given new X rows"""
        y_hat = np.full(len(X_new), self.init_value)

        for tree in self.trees:
            y_hat += self.learning_rate * tree.predict(X_new).values

        return y_hat

    def mse_by_boost(self, X_new: pd.DataFrame, Y: pd.Series) -> np.ndarray:
        """Return MSE by boost iteration for an already-fitted ensemble"""
        y_hat = np.full(len(X_new), self.init_value)
        MSE = []
        for tree in self.trees:
            y_hat += self.learning_rate * tree.predict(X_new).values
            MSE.append(np.mean((Y - y_hat)**2))
        return MSE
```

Now to try it out.

### Training

We can use a single regression tree as a baseline. We will add in some more independent variables and permit the regression tree to grow into smaller leaves and greater depth.

```
X = c[['Hits', 'Years', 'RBI', 'Walks', 'Runs', 'PutOuts']]
tree = RegressionTree(X, Y, max_depth=4, min_cnt_leaf=20)
tree.fit()
print(tree)
```

```
## [Years <= 4]
```

```
## ├──[Years <= 3]
## |   ├──[Hits <= 112]
## |   |   ├──[PutOuts <= 152]
## |   |   |   ├──[Leaf: 4.66 Cnt: 20 MSE: 0.54]
## |   |   |   └──[Leaf: 4.80 Cnt: 22 MSE: 0.28]
## |   |   └──[Leaf: 5.23 Cnt: 20 MSE: 0.13]
## |   └──[Leaf: 5.58 Cnt: 28 MSE: 0.36]
## └──[Hits <= 117]
##     ├──[Walks <= 21]
##     |   ├──[Leaf: 5.69 Cnt: 26 MSE: 0.19]
##     |   └──[Walks <= 43]
##     |       ├──[Leaf: 6.02 Cnt: 43 MSE: 0.31]
##     |       └──[Leaf: 6.34 Cnt: 21 MSE: 0.22]
##     └──[Walks <= 59]
##         ├──[Walks <= 46]
##         |   ├──[Leaf: 6.65 Cnt: 30 MSE: 0.08]
##         |   └──[Leaf: 6.47 Cnt: 20 MSE: 0.38]
##         └──[Leaf: 6.99 Cnt: 33 MSE: 0.22]
```

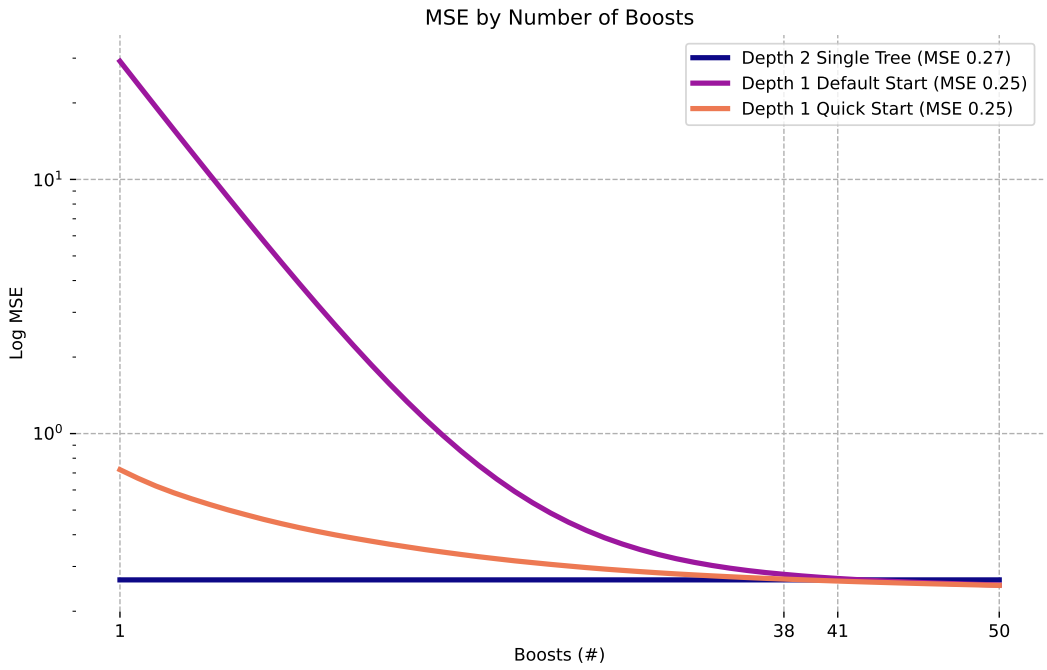
Now let's try two ensembles of 50 "decision stumps" (trees of depth 1): one with the algorithm from the text, another than uses the mean $Y$ as a starting place. Using a tree stump ensures we're pitting weak classifiers against a stronger single regression tree.

```
# use our current depth 2 single tree as a baseline
y_hat_tree = tree.predict(X)
mse_tree = np.mean((Y - y_hat_tree)**2)

# try 50 boosts using a depth 1 "decision stump"
bt50 = BoostedTrees(boost_cnt=50, max_depth=1, min_cnt_leaf=50)
bt50.fit(X, Y)
y_hat_boost50a = bt50.predict(X)
mses_50a = bt50.mse_by_boost(X, Y)

# once more but using mean Y as our starting place
bt50.fit(X, Y, quick_start=True)
y_hat_boost50b = bt50.predict(X)
mses_50b = bt50.mse_by_boost(X, Y)
```

We can plot the ensemble training MSEs by number of boosts. We'll need to use a log of training MSE to accommodate the scales.

```
# viz
colors = plt.cm.plasma(np.linspace(0, 1, 4))
plotting_x = np.arange(len(mses_50a))+1

interesting_x = [plotting_x[0]]
interesting_x.extend(np.nonzero(np.diff(np.sign(mses_50a - mse_tree)))[0])
interesting_x.extend(np.nonzero(np.diff(np.sign(mses_50b - mse_tree)))[0])
interesting_x.append(plotting_x[-1])

fig, ax = plt.subplots(figsize=(10,6))

ax.set_xticks(sorted(interesting_x))
```

```
ax.plot([min(plotting_x), max(plotting_x)], [mse_tree, mse_tree],
        lw=3, color=colors[0], label=f"Depth 2 Single Tree (MSE {mse_tree:.2f})")

plt.plot(plotting_x, mses_50a, color=colors[1], lw=3,
         label=f"Depth 1 Default Start (MSE {np.mean((Y - y_hat_boost50a)**2):.2f})")
plt.plot(plotting_x, mses_50b, color=colors[2], lw=3,
         label=f"Depth 1 Quick Start (MSE {np.mean((Y - y_hat_boost50b)**2):.2f})")

# Change y-axis to log scale
ax.set_yscale('log')

# suppress borders of plot area
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
ax.spines['left'].set_visible(False)
ax.spines['bottom'].set_visible(False)

plt.title('MSE by Number of Boosts')
plt.xlabel('Boosts (#)')
plt.ylabel('Log MSE')
plt.legend()
plt.grid(linestyle='--')
plt.show()
```




The quick start method does indeed give us a much better initial guess, and both quickly surpass the single depth 2 tree (no need to go past 50 trees).

### Cross Validation

How does actual test MSE compare? Let's define a function to do k-fold cross validation (as we did back in chapter 5) and see. This time we'll write it to use `pd.DataFrame` and `pd.Series` as input.

```
rng = np.random.default_rng(2025)

def cvk(model,
        X: pd.DataFrame,
        Y: pd.Series,
        k: int = 5) -> pd.DataFrame:
    """Return a DataFrame of train & test scores from k-fold"""
    train_scores = []
    test_scores = []
    indices = Y.index.to_numpy().copy()
    rng.shuffle(indices)
    folds = np.array_split(indices, k)

    for i in range(k):

        # for each test fold, the other folds serve as training data
        test_fold = folds[i]
        train_fold = np.concatenate([folds[j] for j in range(k) if j != i])

        # train the model
        tree.fit(X.loc[train_fold], Y.loc[train_fold])

        # append training scores to the list
        train_scores.append(
            np.mean((Y.loc[train_fold] - tree.predict(X.loc[train_fold]))**2))

        # append test scores to the list
        test_scores.append(
            np.mean((Y.loc[test_fold] - tree.predict(X.loc[test_fold]))**2))

    return(pd.DataFrame({
        'TrainScore': train_scores,
        'TestScore': test_scores
        }, index=["Fold " + str(i) for i in range(1, k + 1)]))
```

First, what do we see with our regression tree?

```
CV = cvk(tree, X, Y, k=5)
print(pd.concat([CV,
                 pd.DataFrame({
                           'TrainScore': [np.mean(CV['TrainScore'])],
                           'TestScore': [np.mean(CV['TestScore'])]
                           }, index=['Mean'])]))
```

```
##         TrainScore  TestScore
## Fold 1    0.218660   0.534567
## Fold 2    0.287432   0.266855
## Fold 3    0.251518   0.331794
```

```
## Fold 4    0.262562   0.274527
## Fold 5    0.270829   0.290627
## Mean      0.258200   0.339674
```

We don't seem to have overfit too much. How about the boosted trees? Let's dial it back to the point where it had the same training MSE before we test.

```
# save the tree result for a comparison later
comparison =  pd.DataFrame({
                'TrainScore': [np.mean(CV['TrainScore'])],
                'TestScore': [np.mean(CV['TestScore'])]},
    index=[f"Max Depth {tree.max_depth} Min Cnt {tree.min_cnt_leaf} Single Tree"])

# boost up to the point where it had the same training MSE as the tree
n_boosts = np.nonzero(np.diff(np.sign(mses_50a - mse_tree)))[0][0]
bt = BoostedTrees(boost_cnt=n_boosts,
                  learning_rate=bt50.learning_rate,
                  max_depth=bt50.max_depth,
                  min_cnt_leaf=bt50.min_cnt_leaf)
CV = cvk(bt, X, Y, k=5)
print(pd.concat([CV,
                 pd.DataFrame({
                            'TrainScore': [np.mean(CV['TrainScore'])],
                            'TestScore': [np.mean(CV['TestScore'])]
                            }, index=['Mean'])]))
```

```
##         TrainScore  TestScore
## Fold 1    0.289728   0.347714
## Fold 2    0.250558   0.415732
## Fold 3    0.245720   0.439181
## Fold 4    0.247930   0.349310
## Fold 5    0.279217   0.220101
## Mean      0.262631   0.354408
```

Once again, not a horrible spread between training MSE and test MSE. Let's save this last result for later and continue.

```
comparison = pd.concat([comparison,
                        pd.DataFrame({
                            'TrainScore': [np.mean(CV['TrainScore'])],
                            'TestScore': [np.mean(CV['TestScore'])]
                            }, index=[
                                f"Max Depth {bt.max_depth} "
                                f"Min Cnt {bt.min_cnt_leaf} "
                                f"LR {bt.learning_rate} "
                                f"Boosted {bt.boost_cnt}"])])
```

Next we can try varying the learning rate.

### Slower Learners

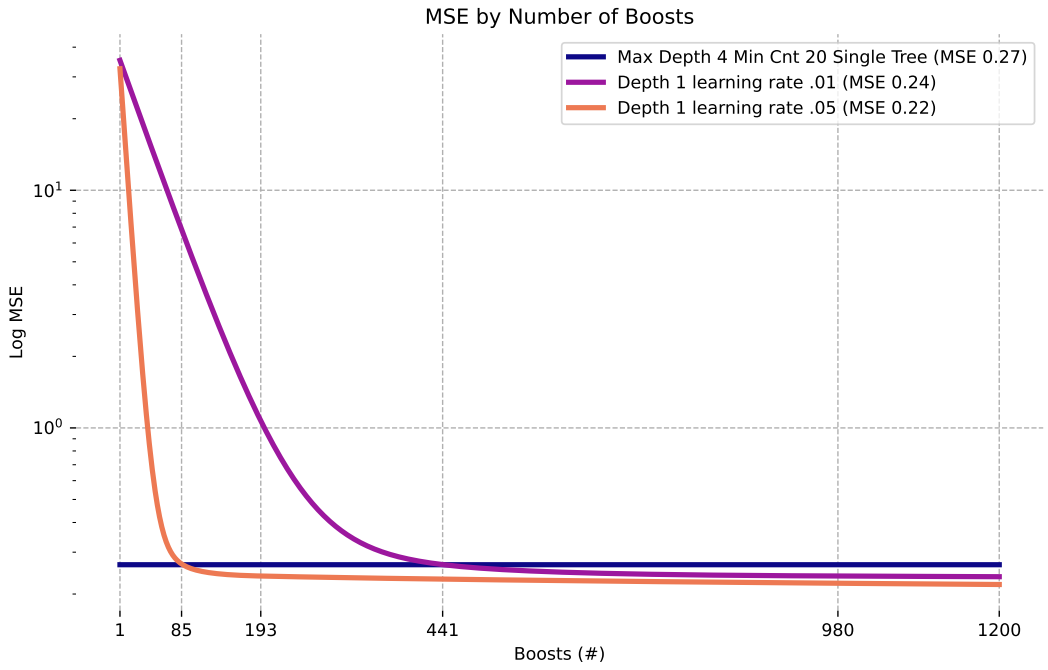
Let's slow this down but greatly increase the number of boosts.

```
# slow down the learning rate but include many boosts
bt1200lr01 = BoostedTrees(boost_cnt=1200, learning_rate=0.01, max_depth=1, min_cnt_leaf=50)
bt1200lr01.fit(X, Y)
y_hat_bt1200lr01 = bt1200lr01.predict(X)
mses_bt1200lr01 = bt1200lr01.mse_by_boost(X, Y)

# faster learning rate, but still slower than originally
bt1200lr05 = BoostedTrees(boost_cnt=1200, learning_rate=0.05, max_depth=1, min_cnt_leaf=50)
bt1200lr05.fit(X, Y)
y_hat_bt1200lr05 = bt1200lr05.predict(X)
mses_bt1200lr05 = bt1200lr05.mse_by_boost(X, Y)
```

We can visualize as before. Again, we'll put x-axis tick marks on just the interesting points. Let's mark where the boosted trees pass the regression tree as well as where the training MSE reductions flatten out.

```
# viz
colors = plt.cm.plasma(np.linspace(0, 1, 4))
plotting_x = np.arange(len(mses_bt1200lr01))+1

# second derivative of the curves
d2_mses_bt1200lr01 = np.gradient(np.gradient(mses_bt1200lr01))
d2_mses_bt1200lr05 = np.gradient(np.gradient(mses_bt1200lr05))

# interest x are endpoints, crossings, and where the curves flatten out
interesting_x = [plotting_x[0]]
interesting_x.extend(np.nonzero(np.diff(np.sign(mses_bt1200lr01 - mse_tree)))[0])
interesting_x.append(np.where(d2_mses_bt1200lr01[2:-2] <= 0)[0][0]+2)
interesting_x.extend(np.nonzero(np.diff(np.sign(mses_bt1200lr05 - mse_tree)))[0])
interesting_x.append(np.where(d2_mses_bt1200lr05[2:-2] <= 0)[0][0]+2)
interesting_x.append(plotting_x[-1])

fig, ax = plt.subplots(figsize=(10,6))

ax.set_xticks(sorted(interesting_x))

ax.plot([min(plotting_x), max(plotting_x)], [mse_tree, mse_tree],
        lw=3, color=colors[0],
        label=(f"Max Depth {tree.max_depth} "
               f"Min Cnt {tree.min_cnt_leaf} "
               f"Single Tree (MSE {mse_tree:.2f})"))

ax.plot(plotting_x, mses_bt1200lr01, color=colors[1], lw=3,
         label=f"Depth 1 learning rate .01 (MSE {np.mean((Y - y_hat_bt1200lr01)**2):.2f})")

ax.plot(plotting_x, mses_bt1200lr05, color=colors[2], lw=3,
         label=f"Depth 1 learning rate .05 (MSE {np.mean((Y - y_hat_bt1200lr05)**2):.2f})")

# Change y-axis to log scale
ax.set_yscale('log')

# suppress borders of plot area
ax.spines['top'].set_visible(False)
ax.spines['right'].set_visible(False)
```

```
ax.spines['left'].set_visible(False)
ax.spines['bottom'].set_visible(False)

plt.title('MSE by Number of Boosts')
plt.xlabel('Boosts (#)')
plt.ylabel('Log MSE')
plt.legend()
plt.grid(linestyle='--')
plt.show()
```




These slower learners took longer to surpass the training MSE of the single tree.

Let's cross validate at the point the curve flattened out for the slower of two and see if we managed to resist seriously overfitting. We can print the results along with the previous trials.

```
# boost up to the point where it flattened out
n_boosts = np.where(d2_mses_bt1200lr01[2:-2] <= 0)[0][0]+2
bt = BoostedTrees(boost_cnt=n_boosts,
                  learning_rate=bt1200lr01.learning_rate,
                  max_depth=bt1200lr01.max_depth,
                  min_cnt_leaf=bt1200lr01.min_cnt_leaf)
CV = cvk(bt, X, Y, k=5)
comparison = pd.concat([comparison,
                        pd.DataFrame({
                            'TrainScore': [np.mean(CV['TrainScore'])],
                            'TestScore': [np.mean(CV['TestScore'])]
                            }, index=[
                                f"Max Depth {bt.max_depth} "
                                f"Min Cnt {bt.min_cnt_leaf} "
                                f"LR {bt.learning_rate} "
```

```
                                f"Boosted {bt.boost_cnt}"])])
print(comparison)
```

```
##                                           TrainScore  TestScore
## Max Depth 4 Min Cnt 20 Single Tree          0.258200   0.339674
## Max Depth 1 Min Cnt 50 LR 0.1 Boosted 41    0.262631   0.354408
## Max Depth 1 Min Cnt 50 LR 0.01 Boosted 980  0.259605   0.343403
```

Again, not too bad.

# Boosting with AdaBoost.R2

Let's try another type of boosting for our regression trees. We'll use an algorithm from the 1997 paper *Improving Regressors using Boosting Techniques* by Harris Drucker where he modified an existing algorithm into what these days is called *AdaBoost.R2*.

## `AdaBoostR2Trees` class

The paper gives us three choices of loss functions. We'll choose the linear one. We'll make just a couple of additions: 1) it's a good idea to add in an early stop when we're very close to zero error, and 2) we'll renormalize the weights each time.

As always we'll limit the code to just use basic Python + Pandas + NumPy. None of these come with a weighted median function, so we will write our own using `np.argsort()`, `np.cumsum()`, and `np.searchsorted()`.

Let's sketch out a plan for the `AdaBoostR2Trees` class.

- Uses `RegressionTree` for individual trees
- Parameters
  - number of boosts
  - `RegressionTree` parameters for minimum leaf size and maximum tree depth
- Method to fit an ensemble (use algorithm from 1997 paper)
  - allow for early stop when very close to zero error
- Method to predict based on new X
- Method to calculate weighted medians

### Coding

We'll use lists to save trees and $log(\frac{1}{\beta})$ values. To set an epsilon for our "close enough to zero" stop, we'll use `1e-12` times whatever the scale is of our `Y`.

```
class AdaBoostR2Trees:
    """Class to build ensemble of RegressionTrees using AdaBoost.R2"""

    def __init__(self,
                 boost_cnt: int = 5,
                 max_depth: int = 1,
```

```
                min_cnt_leaf: int = 50):
        self.boost_cnt = boost_cnt
        self.max_depth = max_depth
        self.min_cnt_leaf = min_cnt_leaf
        self.trees = []
        self.log_beta_m1s = []

    def fit(self,
            X: pd.DataFrame,
            Y: pd.Series) -> None:
        """Grow an ensemble of boosted trees"""

        # clear any existing ensemble
        self.trees = []
        self.log_beta_m1s = []

        # define an epsilon for early stopping
        scale = np.max(np.abs(Y.values))
        epsilon = 1e-12 * scale if scale > 0 else 1e-12

        # initially equal weights
        w = np.ones(len(Y))/len(Y)

        # repeat while average loss L is less than 0.5
        for _ in range(self.boost_cnt):

            # fit next tree on updated weights
            tree = RegressionTree(
                X,
                Y,
                max_depth=self.max_depth,
                min_cnt_leaf=self.min_cnt_leaf,
                weights=w
            )
            tree.fit()

            # evaluate our new tree
            y_hat = tree.predict(X).values
            abs_err = np.abs(Y.values - y_hat)
            max_err = abs_err.max()

            # stop early if close to zero error
            if max_err < epsilon:
                break

            # loss must be in [0, 1], choosing to use linear loss, other
            # choices are square law and exponential
            L_i = w * abs_err / max_err
            L = np.sum(L_i)

            # early stop criteria from H. Drucker's paper
            if L >= 0.5:
                print(f"Stop on Loss {L} >= 0.5")
```

```
                break

            # tree is suitable: save it
            self.trees.append(tree)

            # form beta = L / (1-L), low beta means high confidence
            beta = L / (1-L)

            # save the log(1/beta) from this iteration
            self.log_beta_m1s.append(np.log(1/beta))

            # update the weights w_i -> w_i beta^(1-L_i) for next iteration
            w *= beta**(1-L_i)
            w /= w.sum() # renormalizing each time

    def predict(self, X_new: pd.DataFrame) -> np.ndarray:
        """Predict new Y given new X rows"""
        # calculate and return the cumulative weighted median
        y_hat_w = np.zeros(len(X_new))
        y_hats = np.column_stack(tree.predict(X_new).values for tree in self.trees)
        for i in range(len(y_hat_w)):
            y_hat_w[i] = self.__weighted_median(y_hats[i], self.log_beta_m1s)
        return y_hat_w

    def __weighted_median(values, weights):
        indices = np.argsort(values)
        w_c = np.cumsum(weights[indices])
        return values[indices[np.searchsorted(w_c, 0.5 * w_c[-1])]]
```

Now to exercise the code.

### Training and Cross Validation

Let's try 50 boosts and compare with our prior outcomes.

```
abr = AdaBoostR2Trees(boost_cnt=50, max_depth=1, min_cnt_leaf=50)
CV = cvk(abr, X, Y, k=5)
comparison = pd.concat([comparison,
                        pd.DataFrame({
                            'TrainScore': [np.mean(CV['TrainScore'])],
                            'TestScore': [np.mean(CV['TestScore'])]
                            }, index=[
                                f"Max Depth {abr.max_depth} "
                                f"Min Cnt {abr.min_cnt_leaf} "
                                f"AdaBoosted.R2 {abr.boost_cnt}"])])
print(comparison)
```

```
##                                           TrainScore  TestScore
## Max Depth 4 Min Cnt 20 Single Tree          0.258200   0.339674
## Max Depth 1 Min Cnt 50 LR 0.1 Boosted 41    0.262631   0.354408
## Max Depth 1 Min Cnt 50 LR 0.01 Boosted 980  0.259605   0.343403
## Max Depth 1 Min Cnt 50 AdaBoosted.R2 50     0.260134   0.334751
```

Seems to work well.