Statistical Machine Learning Notes 12

Trees

Instructor: Justin Domke

Contents

1 Regression Trees

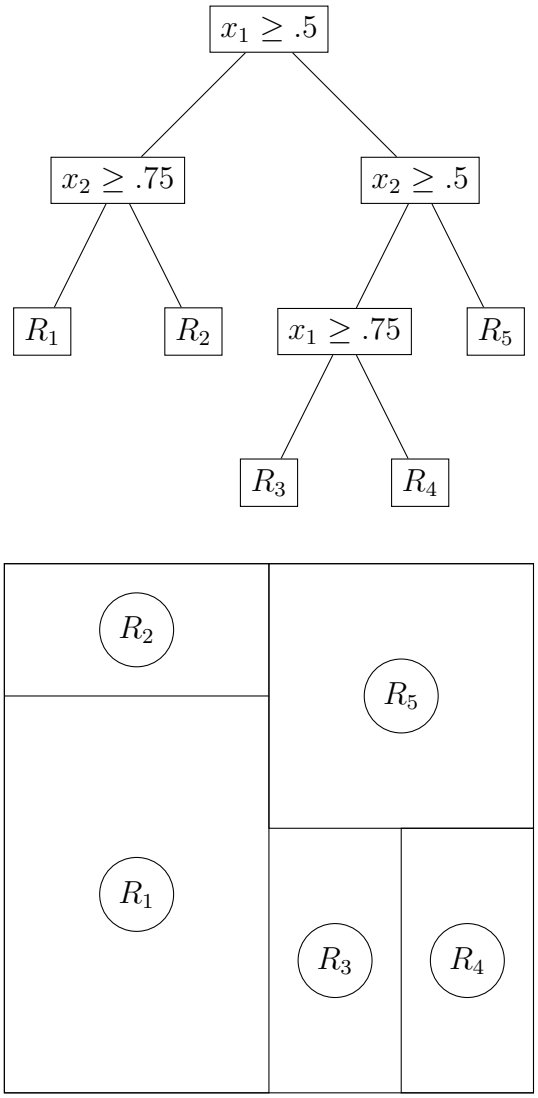
Tree-based methods can be seen as fitting a model that is piecewise constant over some disjoint number of regions R_m.

$$f(\mathbf{x}) = \sum_m c_m I[\mathbf{x} \in R_m]$$

This can also be written as

$$f(\mathbf{x}) = c_m, \, \mathbf{x} \in R_m.$$

In trees, the regions R_m are usually defined by means of binary splits. For example, the tree below corresponds to the following group of regions.



2 Fitting Regression Trees

Suppose we have some dataset $D = \{(\hat{\mathbf{x}}, \hat{y})\}$. We would like to pick the regions R_m and the constants c_m to minimize the squared error

$$\sum_{\hat{\mathbf{x}}, \hat{y} \in D} \big(f(\hat{\mathbf{x}}) - \hat{y}\big)^2.$$

Suppose, first, that the regions R_m are fixed, and we want to fit the constants c_m. We can pretty easily obtain the intuitive result that c_m should be the average of the values $\hat{y}$ corresponding to inputs $\hat{\mathbf{x}}$ in region R_m.

$$\begin{aligned}\arg\min_{c_m} \sum_{\hat{\mathbf{x}},\hat{y}\in D} \big(f(\hat{\mathbf{x}}) - \hat{y}\big)^2 &= \arg\min_{c_m} \sum_{(\hat{\mathbf{x}},\hat{y})} \big(\sum_n c_n I[\mathbf{x} \in R_n] - \hat{y}\big)^2 \\ &= \arg\min_{c_m} \sum_{(\hat{\mathbf{x}},\hat{y}):\hat{\mathbf{x}}\in R_m} \big(c_m - \hat{y}\big)^2 \\ &= \operatorname*{mean}_{(\hat{\mathbf{x}},\hat{y}):\hat{\mathbf{x}}\in R_m} \hat{y}\end{aligned}$$

Now, how to fit the regions R_m? Or, equivalently, how to select all the binary splits? Unfortunately, this turns out to be intractable to do exactly. Instead, we are forced to rely on greedy heuristics. We start with a single node tree, and find the single split such that the training error is minimized. We then recursively split subregions.

Now, if we have D dimensions, and there are N training elements, how can we find the best split? Define

$$R_1(d,s) = \{\mathbf{x} : \mathbf{x}_d \le s\}, \quad R_2(d,s) = \{\mathbf{x} : \mathbf{x}_d > s\}$$

The goal is to find a dimension d and a threshold s to do the optimization

$$\min_{d,s}\big(\min_{c_1} \sum_{(\hat{\mathbf{x}},\hat{y}):\hat{\mathbf{x}}\in R_1(d,s)} (\hat{y} - c_1) + \min_{c_d} \sum_{(\hat{\mathbf{x}},\hat{y}):\hat{\mathbf{x}}\in R_2(d,s)} (\hat{y} - c_2)\big).$$

As we saw above, the minimizations over c_1 and c_2 are accomplished by just setting each to be the mean of all the values $\hat{y}$ corresponding to the region. The question remains, how to optimize over d and s?

Since there are a finite number of dimensions, we can just search over all of them. If we allow s to be a real number, there are an infinite number of possible thresholds. However, there are only a finite number of *effective* thresholds. Given dimension d, take the values $\hat{x}_d$ in sorted order. Then, we can consider

$$s = \frac{1}{2}\hat{x}_d^1 + \frac{1}{2}\hat{x}_d^2$$

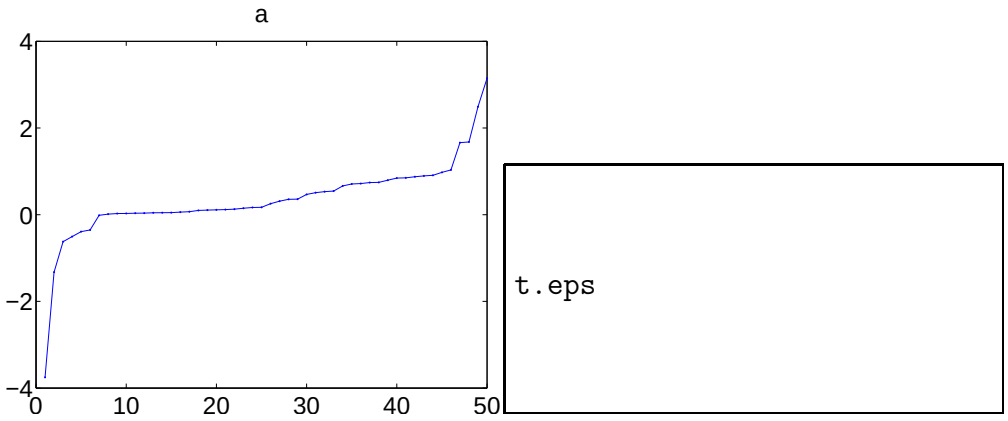
$$s = \frac{1}{2}\hat{x}_d^2 + \frac{1}{2}\hat{x}_d^3$$

$$\vdots$$

$$s = \frac{1}{2}\hat{x}_d^{N-1} + \frac{1}{2}\hat{x}_d^N.$$

Thus, we have a total of ND pairs (d, s) to consider. Done by brute-force, we could compute the value of the least-squares error for each in time $O(N)$, for a total time complexity of $O(N^2D)$. However, as we will see below, for the least-squares error, a clever algorithm can compute the function values for all the thresholds in a particular dimension in time $O(N \log N)$ (the complexity of sorting), and so reduces the overall complexity to $O(ND \log N)$.

Notice that we cannot use a simple strategy like a binary search to find the best s, since the objective is nonconvex in s.



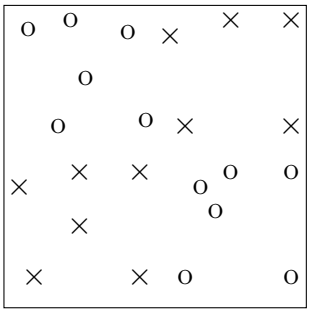
3 Controlling complexity

In linear methods, we generally controlled complexity by regularizing parameters. How should we control complexity with trees? There are a number of possibilities that seem to be used in practice:

- Grow trees to a fixed depth. For example, only allow a total of 5 splits from the origin to a leaf, meaning a total of 2^5 leaf nodes.
- Grow trees to a fixed number of leaf nodes. When growing a tree, one can use various criteria to choose what node to next "split". If these are chosen intelligently, choosing a fixed number (e.g. 25 leaf nodes). generally yields a non-balanced tree. (Note that in many computational contexts, having unbalanced trees is a very bad thing. Here, however, it is not a problem.)
- Only split nodes with a given number of points in them. For example, we might refuse to split nodes with less than 50 data points falling into them.

Because trees are grown greedily, it is generally thought better to grow the tree out to some significant depth, and then "prune" it to avoid overfitting. The reason is that a split that

initially appears to only slightly decrease training error may emerge to be very useful further down the tree. For example, consider the following dataset:



Any single split will produce only a small decrease in training error. However, by splitting twice, we can achieve a classification error of zero.

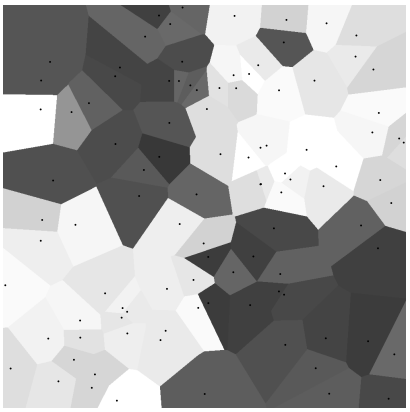
Surprisingly enough, the most common way in practice to control complexity may be to limit the tree to a single split! Such "stumps" are widely used in the context of boosting, as we will see later on.

4 Examples

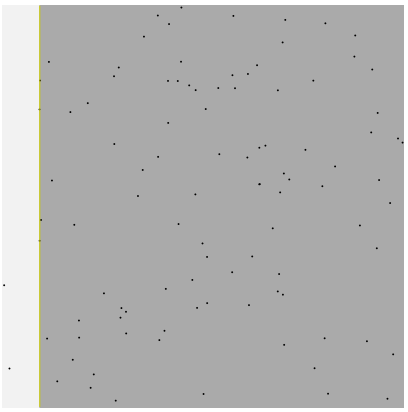
These examples show some two-dimensional datasets. To simultaneously visualize the locations $\hat{\mathbf{x}}$ and values $\hat{y}$, we use a Voronoi diagram. Each pixel in the plane is colored with an intensity proportional to the value $\hat{y}$ corresponding to the nearest point $\hat{\mathbf{x}}$.

In these examples nodes that contain 10 or less data points are not further split.

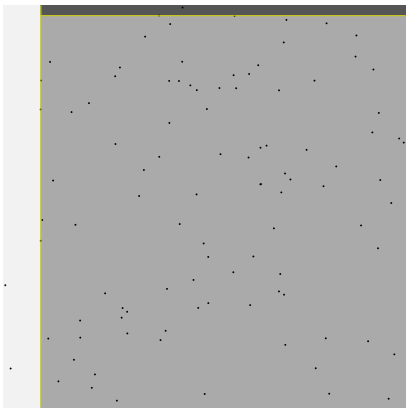
data



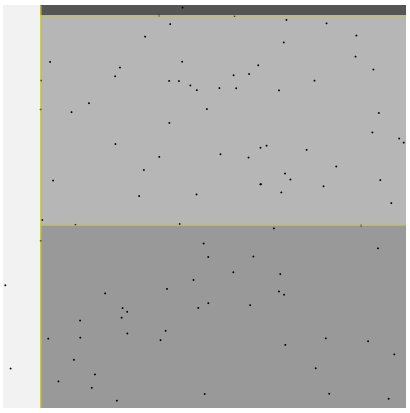
1 split



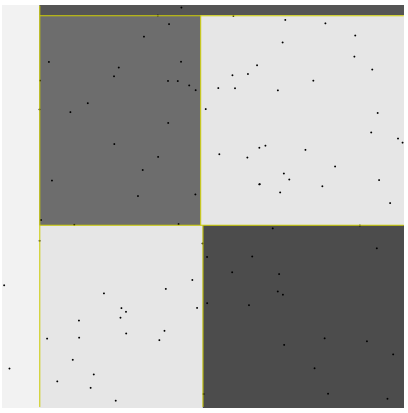
2 splits



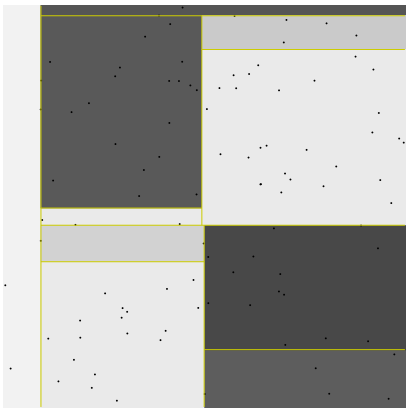
3 splits



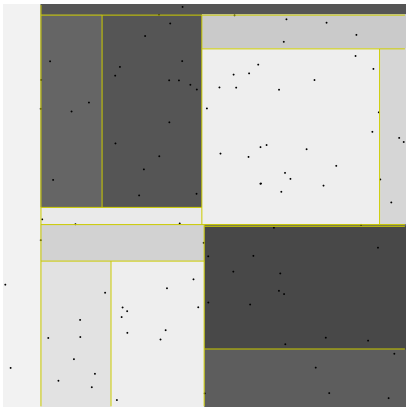
4 splits



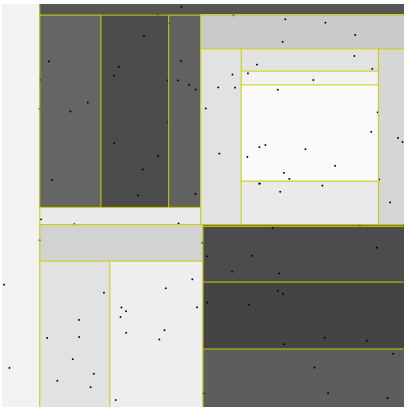
5 splits



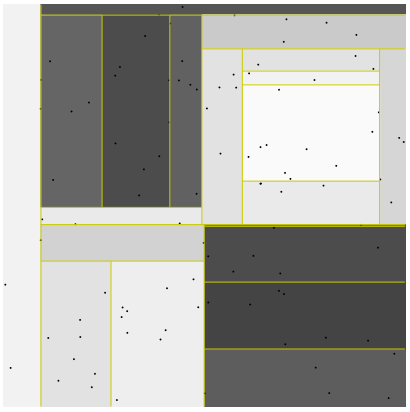
6 splits



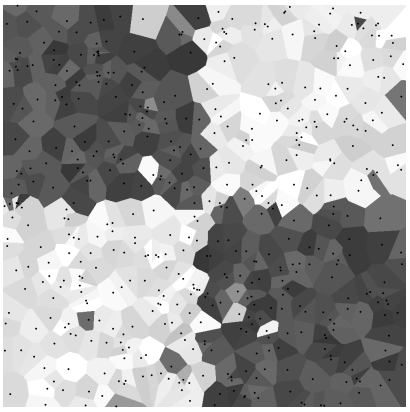
10 splits



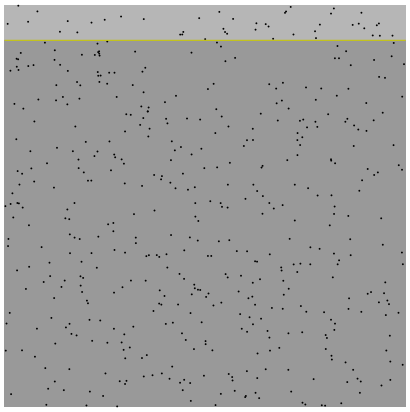
15 splits



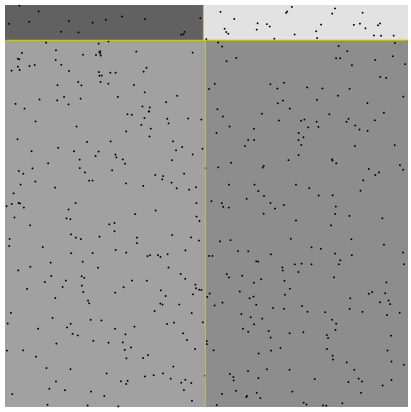
data



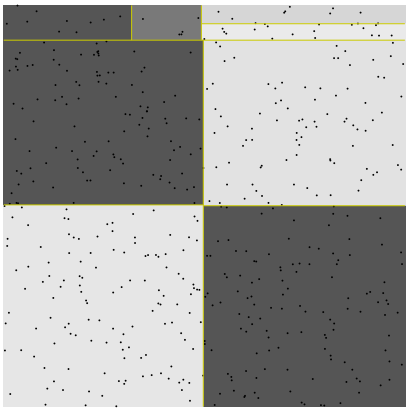
1 split



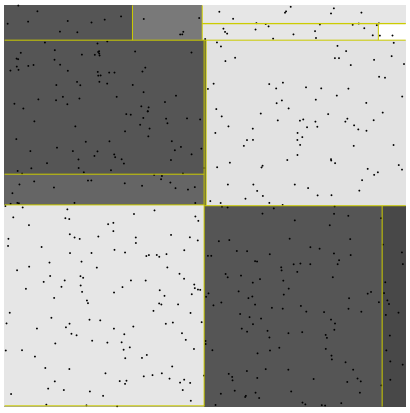
2 splits



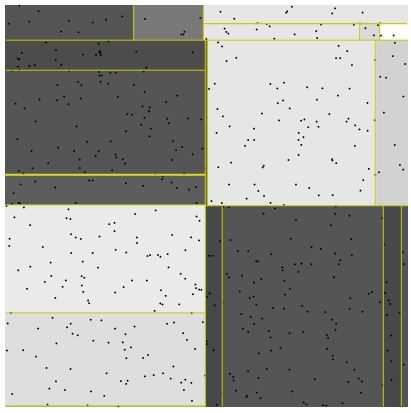
3 splits



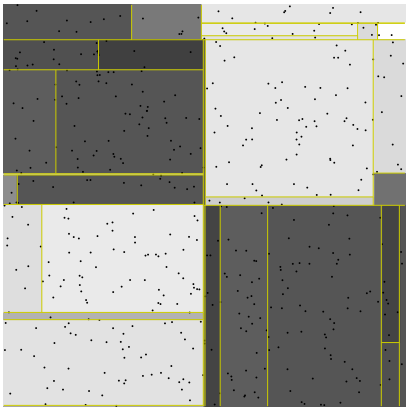
4 splits



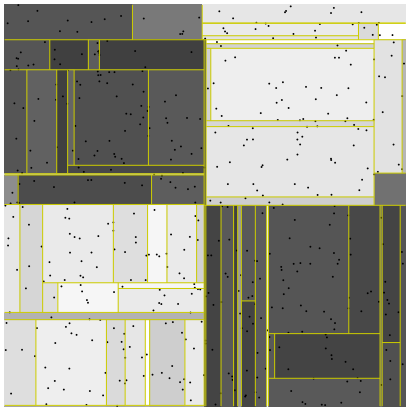
5 splits



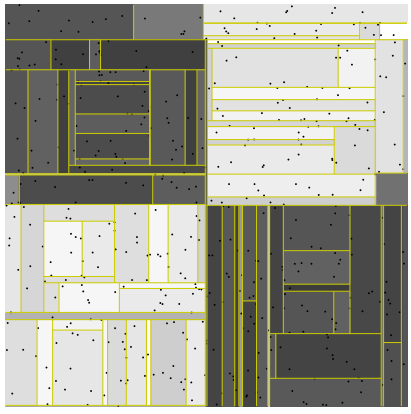
6 splits



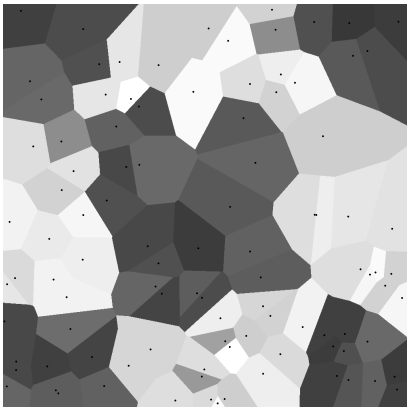
10 splits



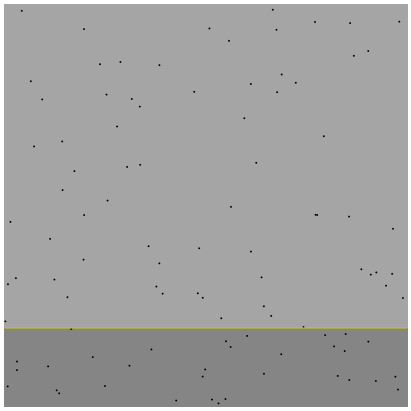
15 splits



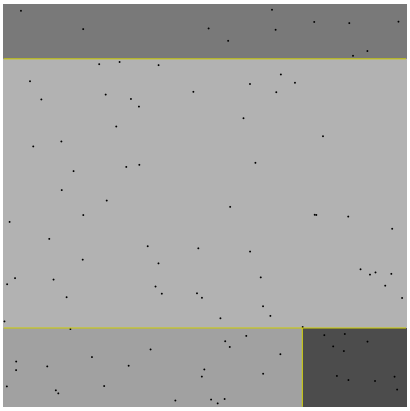
data



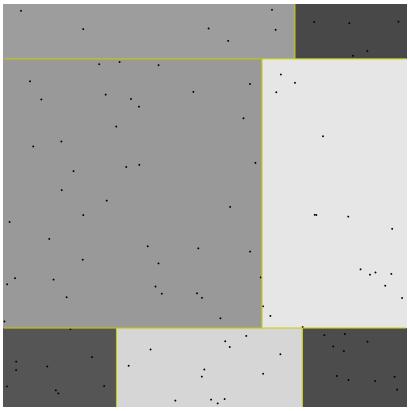
1 split



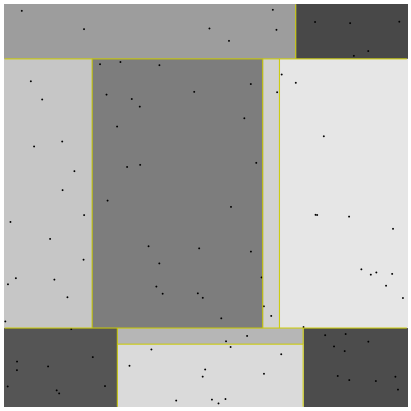
2 splits



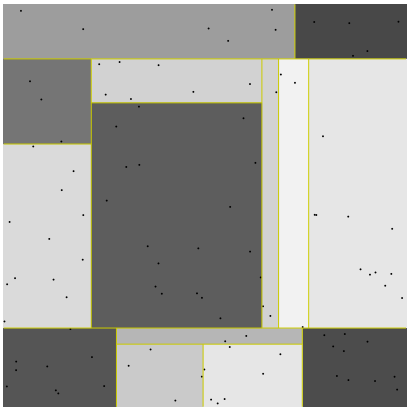
3 splits



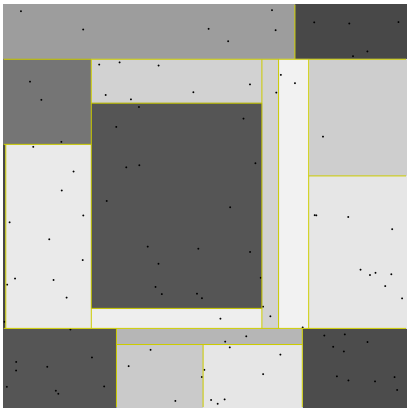
4 splits



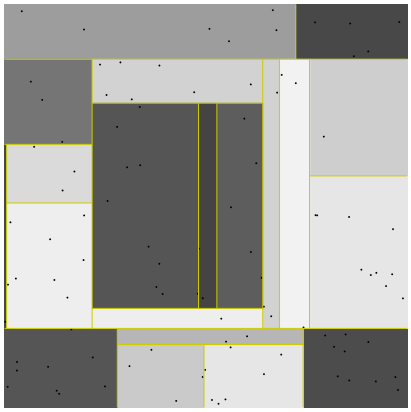
5 splits



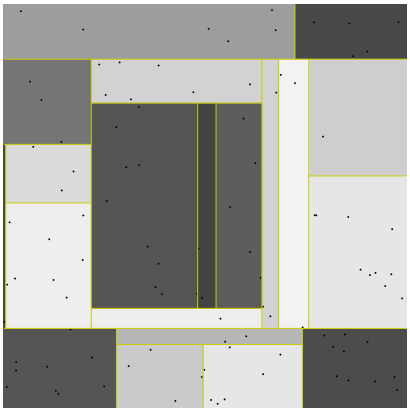
6 splits



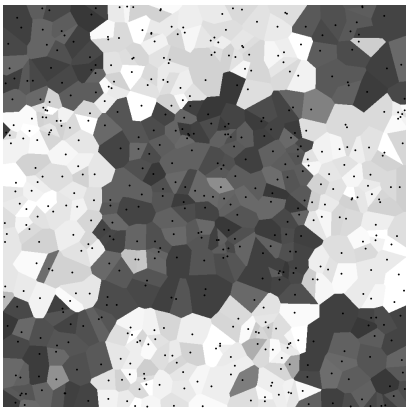
10 splits



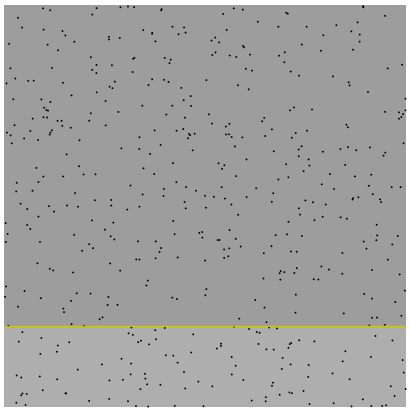
15 splits



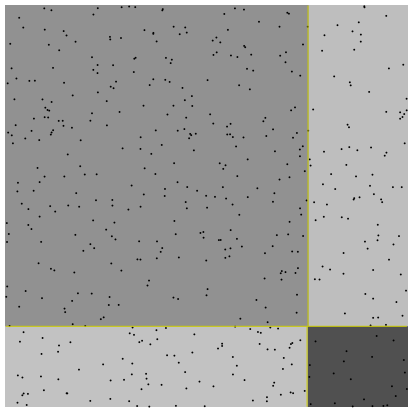
data



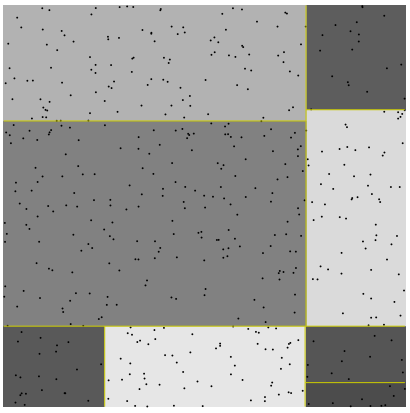
1 split



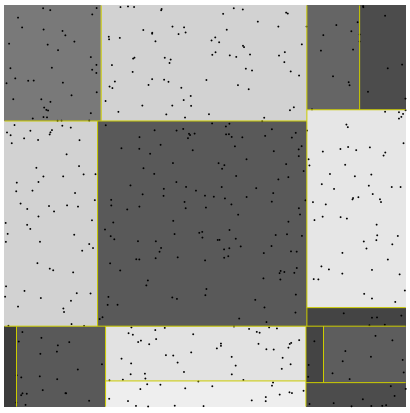
2 splits



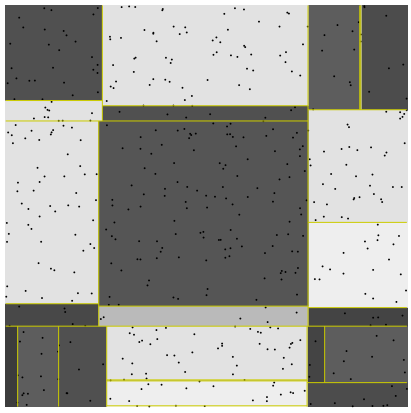
3 splits



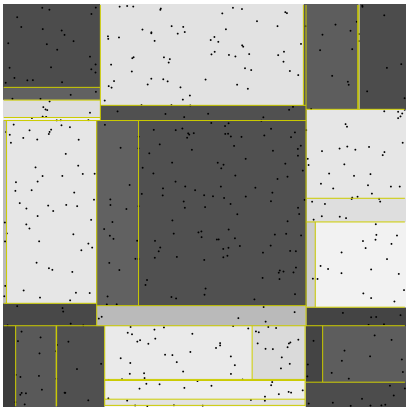
4 splits



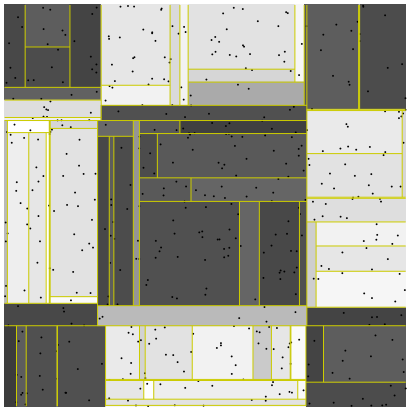
5 splits



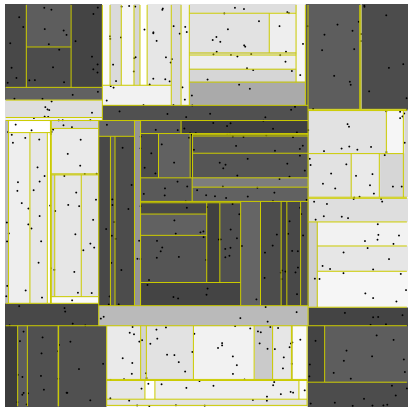
6 splits



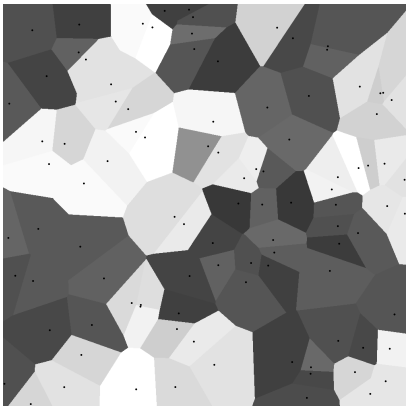
10 splits



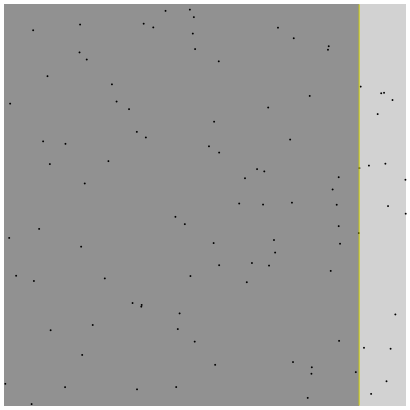
15 splits



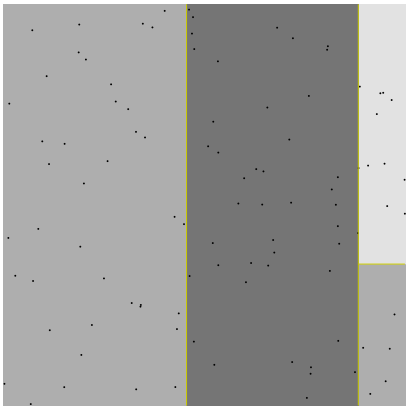
data



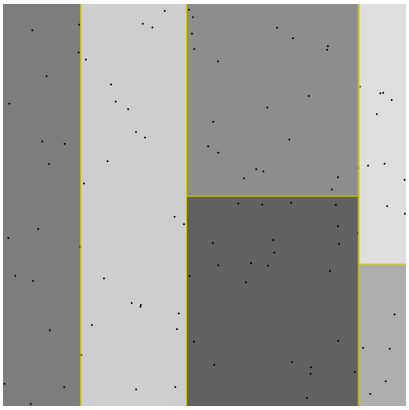
1 split



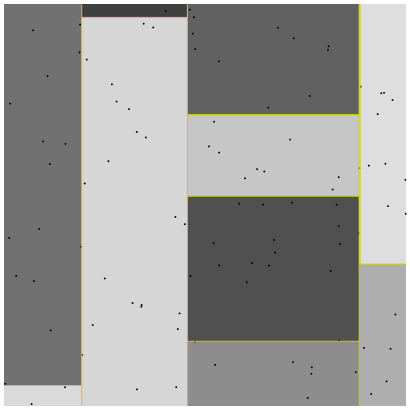
2 splits



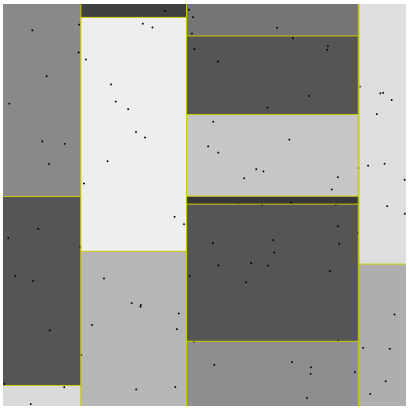
3 splits



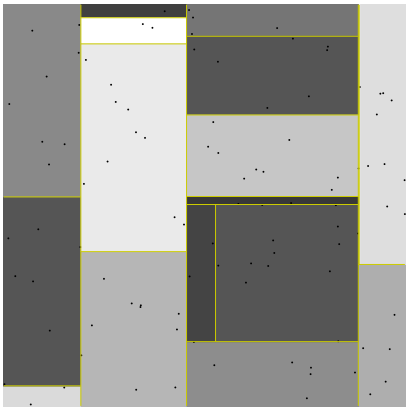
4 splits



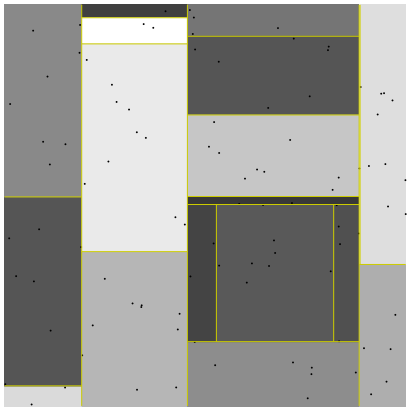
5 splits



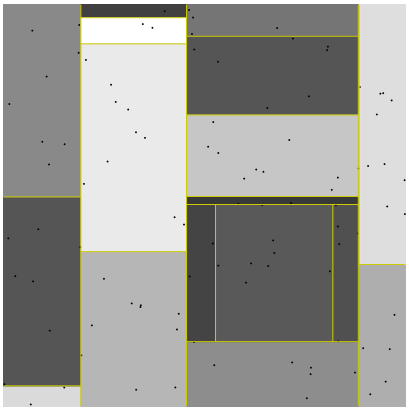
6 splits



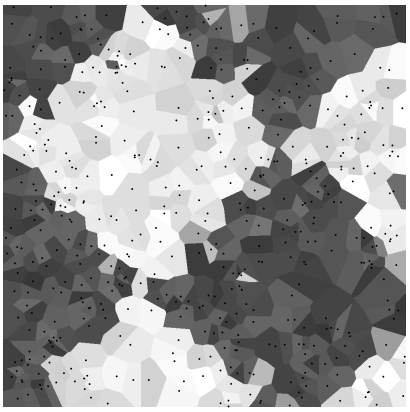
10 splits



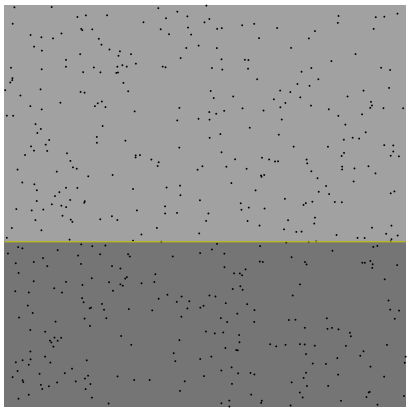
15 splits



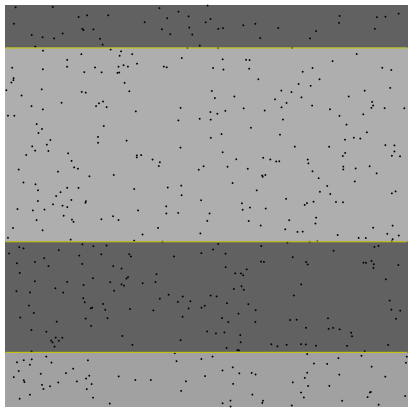
data



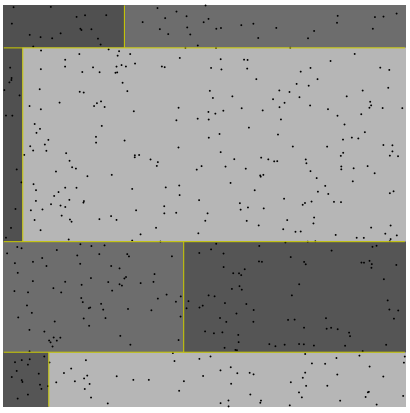
1 split



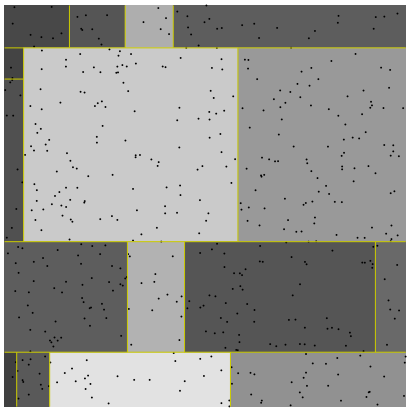
2 splits



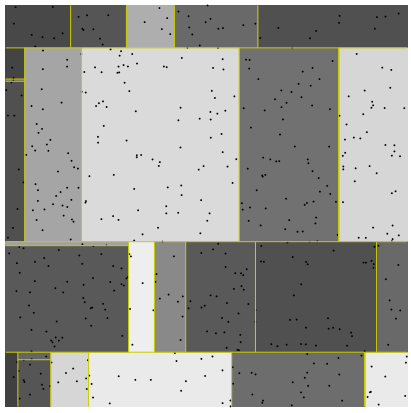
3 splits



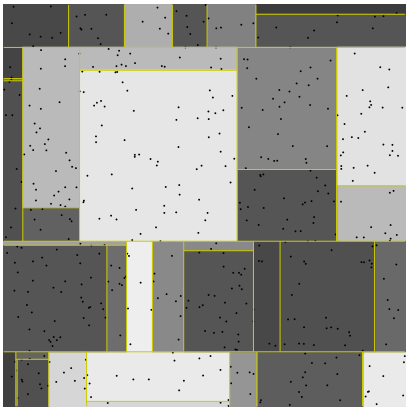
4 splits



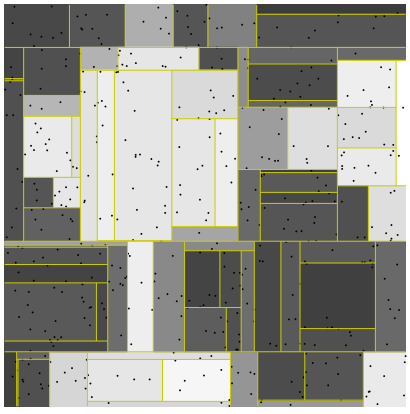
5 splits



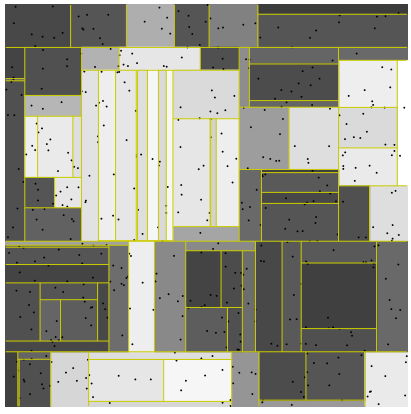
6 splits



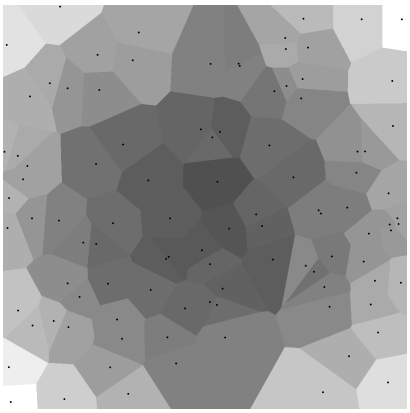
10 splits



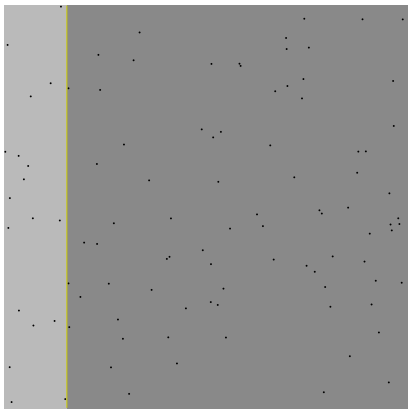
15 splits



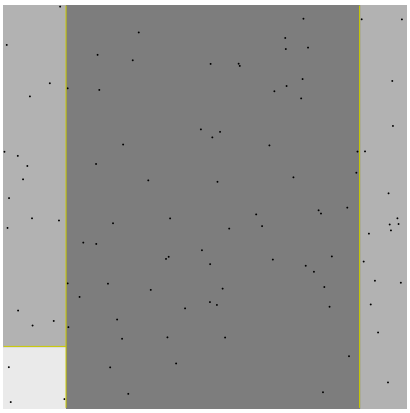
data



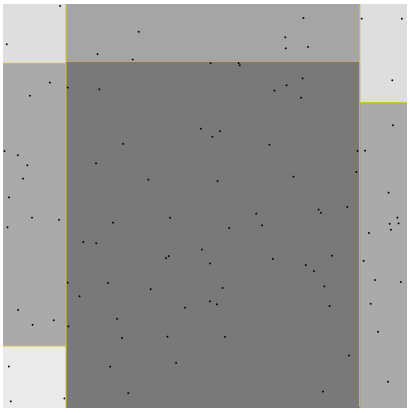
1 split



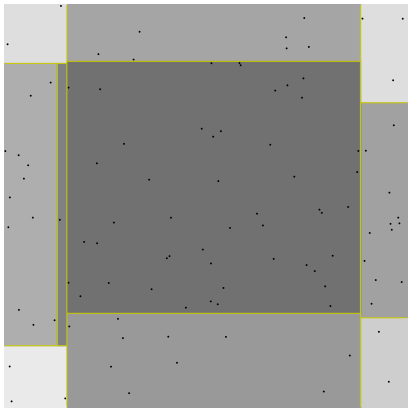
2 splits



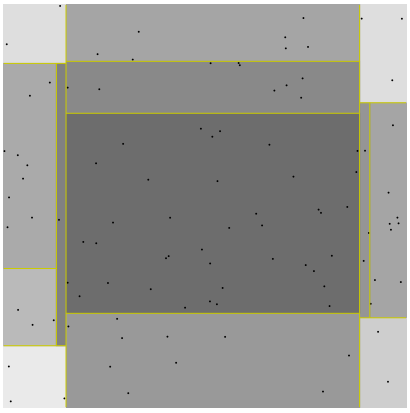
3 splits



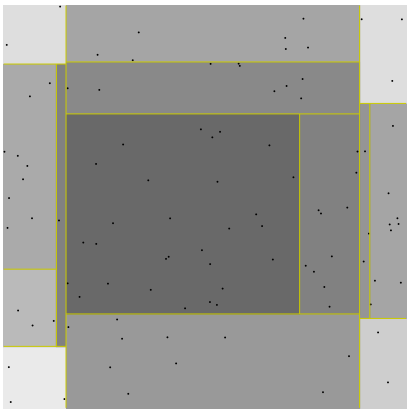
4 splits



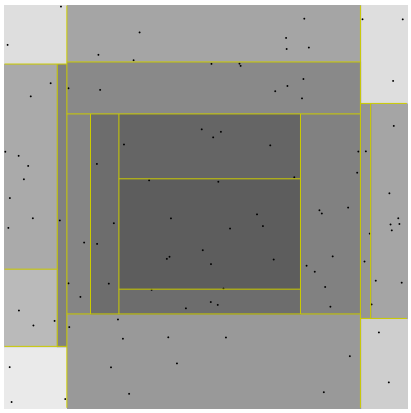
5 splits



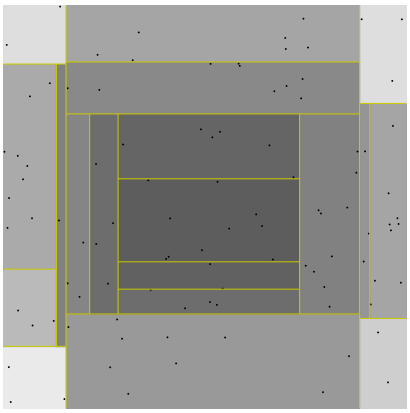
6 splits

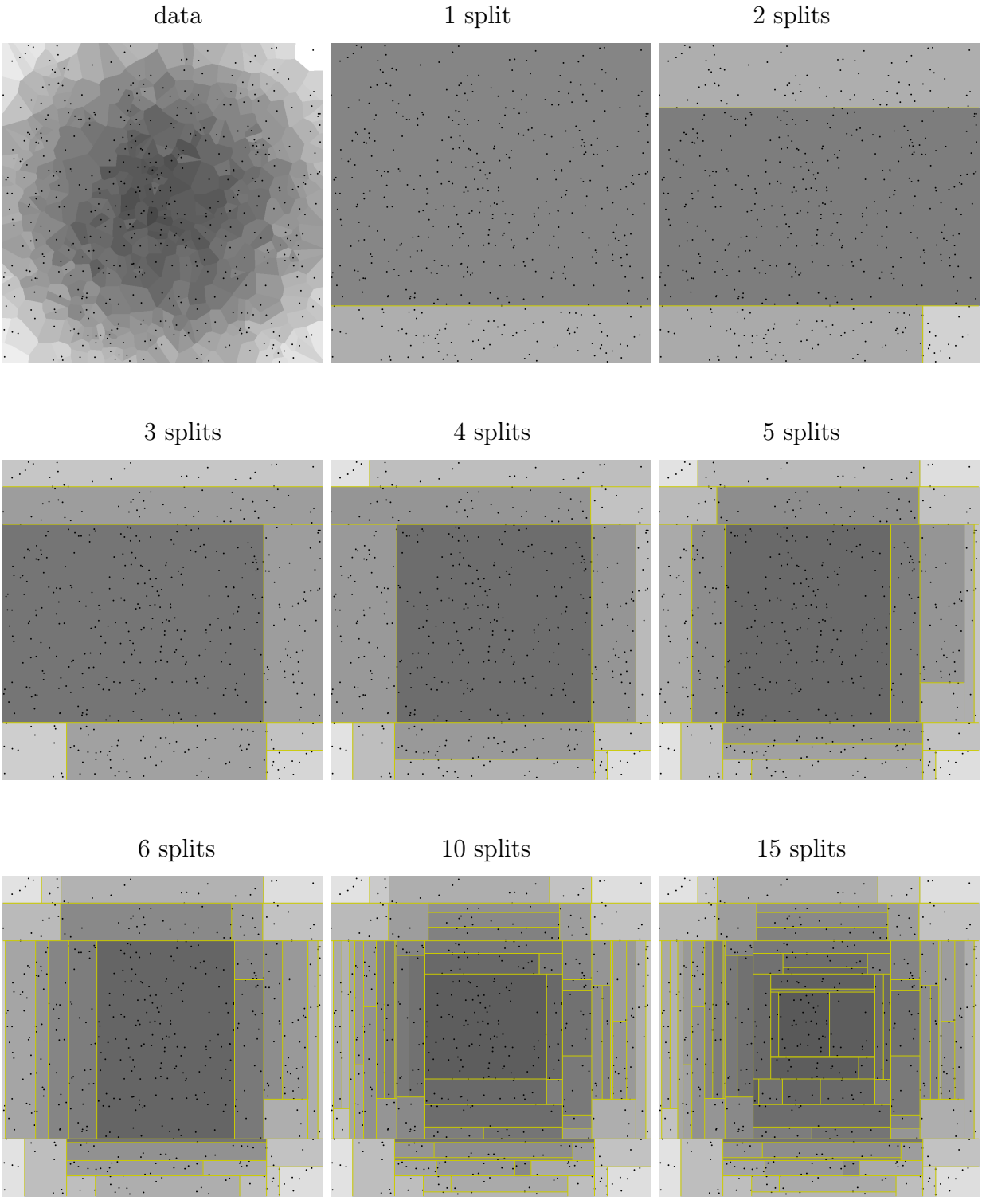


10 splits



15 splits




data
1 split
2 splits
3 splits
4 splits
5 splits
6 splits
10 splits
15 splits

5 Conclusion

The advantages of tree-methods are that they are:

- **Fast to learn**.
- **Fast to evaluate**.
- **Interpretable**. Even in high dimensions, we can draw a tree, and understand it quite well.
- **Invariant to affine scalings**. That is, if we take all the inputs $\hat{\mathbf{x}}$, and set $\hat{\mathbf{x}}'$ through some affine scaling $\hat{x}'_i = a_i \hat{x}_i + b_i$, exactly the same tree will be learned, with the split points correspondingly transformed. This means it is totally unnecessary to center or rescale the data, as is important with linear methods.
- **Somewhat tolerant of missing data**. If some variables $\hat{x}_d$ are unknown, decision trees can still sometimes make use of the dth dimension. The way this is done is merely to refuse to split on a dimension with hidden data. Further down the tree, there will be less data at a given node. If the variable is available in all those, it can still be made use of.

Some of the disadvantages of tree-methods are:

- They are **aligned to the coordinate axes**. As we saw above with the "grid" data, regression trees struggled to deal with diagonal structure in the regression function. One can design methods that try to fit more general splitting criteria like $[\mathbf{v} \cdot \mathbf{x} \leq t]$. However, recall that above we used a brute-force search over d to deal with only splits of the form $[\hat{\mathbf{e}}_d \cdot \mathbf{x} \leq t]$. Thus, dealing with the large computational problem to find the best $\mathbf{v}$ is difficult. There have been several heuristics proposed for doing this. Note also, of course, that we will pay a variance penalty for using these more powerful functions.
- They are not very **smooth**. That is, the function value changes discontinuously between the different regions.
- We have to learn them **greedily**. Trees are strongly reliant on the greedy heuristics used to grow them. As we saw above on the "grid" data where the optimal tree is of 4 nodes, with splits at $x_1 \leq .5$ and $x_2 \leq .5$, the algorithm failed to find this tree. The reason is that each of these splits, alone, does not cause much decrease in the training error. It is possible to design more expensive tree-growing procedures that "look ahead" more than a single split down the tree, which can combat these problems to some degree, though at a significant cost.

6 Fast threshold selection

Consider the following problem. We have a sorted list of numbers, $a_1, a_2, ..., a_N$. We want to find a threshold t such that we can minimize

$$\sum_{i \le t} (a_i - \operatorname*{mean}_{i \le t} a_i)^2 + \sum_{i > t} (a_i - \operatorname*{mean}_{i > t} a_i)^2.$$

Now, we can obviously find this threshold with complexity $O(N^2)$. Just try $t = 1$, $t = 2$, ..., $t = N - 1$. For each threshold, compute the two means, and then compute the total function value. However, can we do better $O(N^2)$?

It turns out that we can reduce the complexity to $O(N)$. The key idea is the following recursion[1]:

$$\operatorname*{mean}_{i \le n+1} a_i = \frac{1}{n+1}\big(a_{n+1} + n \operatorname*{mean}_{i \le n} a_i\big).$$

(This can also be done in reverse to produce the means over all $i > n$).

Which we can use to compute all the means in time $O(N)$. Next, note that

$$\begin{aligned} \sum_{i \le t} (a_i - \operatorname*{mean}_{j \le t} a_j)^2 &= \sum_{i \le t} \big(a_i^2 - 2a_i \operatorname*{mean}_{j \le t} a_j + (\operatorname*{mean}_{j \le t} a_j)^2\big) \\ &= \sum_{i \le t} a_i^2 - 2\big(\sum_{i \le t} a_i\big) \operatorname*{mean}_{j \le t} a_j + t(\operatorname*{mean}_{j \le t} a_j)^2. \end{aligned}$$

There is a similar equation for the sums over $i > t$. Putting it all together, we have the following fast threshold selection algorithm.

- Input a_1,a_2, ..., a_N.
- For all t, compute $\sum_{i \le t} a_i$ and $\sum_{i > t} a_i$.

[1] Want proof? Here is the algebra.

$$\begin{aligned} \sum_{i \le n+1} a_i &= a_{n+1} + \sum_{i \le n} a_i \\ (n+1) \operatorname*{mean}_{i \le n+1} a_i &= a_{n+1} + n \operatorname*{mean}_{i \le n} a_i \\ \operatorname*{mean}_{i \le n+1} a_i &= \frac{1}{n+1}\big(a_{n+1} + n \operatorname*{mean}_{i \le n} a_i\big) \end{aligned}$$

- For all t, compute $\sum_{i \le t} a_i^2$ and $\sum_{i > t} a_i^2$.
- For all t, compute $\operatorname*{mean}_{i \le t} a_i$ and $\operatorname*{mean}_{i > t} a_i$
- For all t, compute $f_t = \sum_{i \le t} (a_i - \operatorname*{mean}_{i \le t} a_i)^2 + \sum_{i > t} (a_i - \operatorname*{mean}_{i > t} a_i)^2$.
- Select $t^* = \arg\min_t f_t$.