Decision Trees

Last modified: 2026-11-19

```
# reticulate provisions the packages this page's Python chunks import.
python_packages <- c("numpy", "pandas", "matplotlib", "scikit-learn")
reticulate::py_require(python_packages)
```

Source

Adapted from Kameron Decker Harris' 2022 weekly notes, and drawing on Logan Sizemore's Spring 2025 lecture on decision trees. The Gini index and the reference on classification and regression trees are from the cited sources rather than from that lecture, which uses entropy[a] alone for impurity.

[a] https://morrison-lab.github.io/pds/expectation.html#def-entropy

1 Decision trees

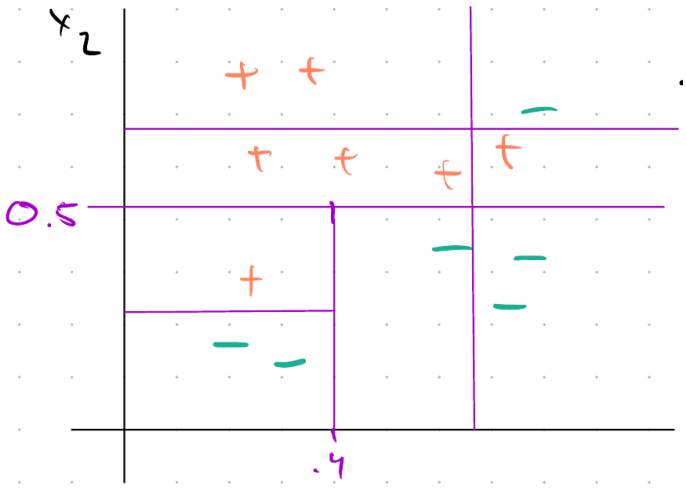


Figure 1: Points labeled + (orange) and − (green) in the plane of two features, cut into rectangles by purple segments parallel to the axes. For example, one horizontal segment is at $x_2 = 0.5$, and one vertical segment below it is at $x_1 = 0.4$.

A decision tree partitions the input space into rectangles aligned with the axes. The number of nodes, that is of comparisons, measures the complexity of the tree, and a complex tree overfits.

The parts have names. The topmost node is the **root**; an internal node, which holds a comparison between one feature and one threshold, is a **decision node**; a terminal node, which holds a prediction, is a **leaf**. Applying a decision node's comparison to send each point down one branch or the other is a **split**.

Goal:

- Keep the tree as compact as possible.

- Make the leaves pure.

Finding the optimal tree is NP-hard, so instead we greedily minimize the "impurity" of each split.

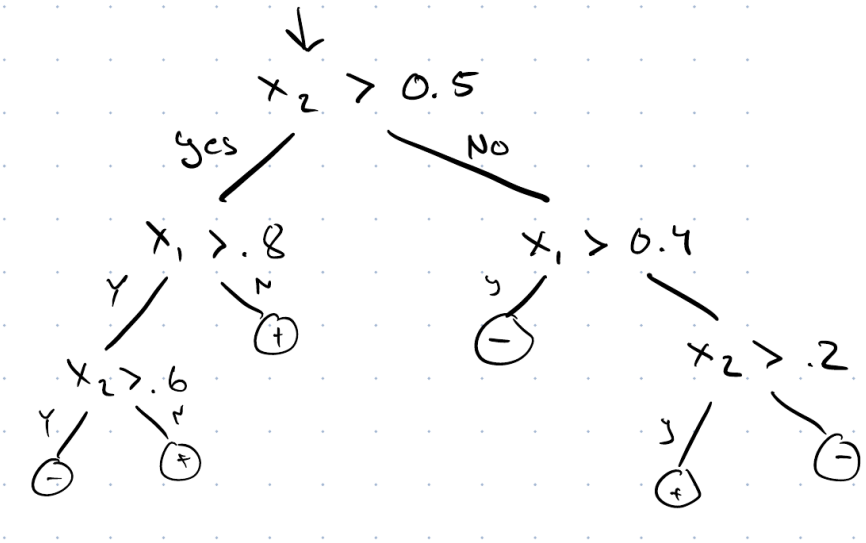


Figure 2: A tree whose root tests $x_2 > 0.5$. Each branch is labeled yes or no, and each circled leaf is marked $+$ or $-$.

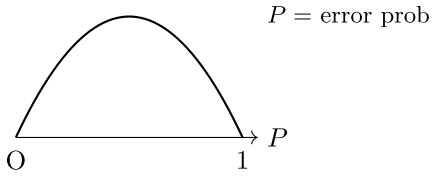


Figure 3: An arch above a horizontal axis running from 0 to 1, labeled P. The arch is at 0 at both ends and highest in the middle.

Watch

- Hastie and Tibshirani cover tree-based methods[a] [1], video 8.1.
- Weinberger builds decision trees, from 3:35[b] [2], Lecture 29.
- Starmer builds classification trees[c] [3].

[a] https://www.youtube.com/watch?v=QNnayf--_yk
[b] https://www.youtube.com/watch?v=a3ioGSwfVpE&t=215s
[c] https://www.youtube.com/watch?v=_L39rN6gz7Y

Read

GeeksforGeeks explains decision trees[a] [4].

[a] https://www.geeksforgeeks.org/machine-learning/decision-tree-introduction-example/

Read

The University of Washington's CSE 446 slides cover trees[a] [5], Lecture 15.

[a] https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/15-trees.pdf

1.1 Measuring impurity

The word "impurity" above is central to the algorithm, so we define it here. A node holding points of one class only is pure; a node holding an even mix of every class is as impure as a node can be. Two standard measures capture exactly this.

Entropy comes from information theory. An outcome of probability p carries $-\log_2 p$ bits of surprisal[1]: large when the outcome was unlikely, and zero when it was certain. Predicting heads on a single toss of a fair coin and being right is worth one bit; predicting ten heads in a row and being right is worth 10 bits. The population average of the surprisals, weighted by the probabilities themselves, is the **entropy** [6]:

$$H(S) = -\sum_{k=1}^{C} p_k \log_2 p_k$$

where p_k is the fraction of the points in the set S that carry class k, and $0 \log_2 0$ is read as 0. It is zero when one class holds all the mass — there is nothing left to transmit — and largest, at $\log_2 C$, when the classes are evenly split. So it measures impurity directly: the entropy of a node is the number of bits per point you would still have to send to resolve the labels once the reader knows which leaf a point fell into.

Here are three example nodes. A node holding 10 points of each of two classes has $H = 1$ bit, the worst possible. A node holding 10 and 4 has $H = 0.863$. A node holding 10 and 0 has $H = 0$.

Gini impurity is the other standard choice:

$$G(S) = 1 - \sum_{k=1}^{C} p_k^2$$

It is the probability that two points drawn independently from S carry different labels — equivalently, the error rate you would get by labeling a point at random from S's own class proportions. It agrees with entropy at both ends, zero for a pure node and largest for an even mix, and differs only mildly in between, so the choice between the two rarely changes the tree that comes out. Entropy is the criterion of Quinlan's ID3 [7]; Gini is the default in the classification and regression tree procedure of Breiman, Friedman, Olshen and Stone [8].

Watch

- Weinberger builds decision trees from purity functions, from 14:43[a] [2], Lecture 29.
- Starmer explains entropy[b] [9].

[a] https://www.youtube.com/watch?v=a3ioGSwfVpE&t=883s
[b] https://www.youtube.com/watch?v=YtebGVx-Fxw

Read

GeeksforGeeks explains Gini impurity and entropy[a] [10].

[a] https://www.geeksforgeeks.org/machine-learning/gini-impurity-and-entropy-in-decision-tree-ml/

[1] https://morrison-lab.github.io/ids/entropy.html#def-surprisal

1.2 Choosing a split

A split sends the points of S into B branches $S_1, \dots, S_B$ — $B = 2$ for a yes-or-no comparison on one feature. Score it by how much impurity it removes. The impurity left afterwards is each branch's impurity weighted by the share of the points that landed in it, and the **information gain** is the difference, which is the mutual information[2] between a point's branch and its class:

$$\mathrm{IG}(S, \mathrm{split}) = H(S) - \sum_{b=1}^{B} \frac{|S_b|}{|S|} H(S_b)$$

The weighted sum stops the greedy rule from being misled. Separating one point into its own branch makes that branch perfectly pure, but it arrives with weight $1/|S|$ and leaves the other branch almost exactly as impure as the parent was, so the gain is negligible. The rule rewards a split for purifying a large share of the data, not for purifying something.

Growing the tree is then a loop: at each leaf, try every feature and every valid threshold, compute the gain for each candidate, keep the largest, and recurse on the branches it creates.

Example 1.1 (Comparing two candidate splits)**.**
Take 16 patients, 8 with a positive test and 8 with a negative one, described by two measurements x_1 and x_2. The root holds an even split, so $H = 1$ bit. Consider two candidate splits:

Table 1: The counts and $\mathbb{H}$ in each branch of two candidate splits.

split	branch	positive	negative	H
$x_1 \le 3$	yes	3	0	0
	no	5	8	0.961
$x_2 \le 1$	yes	6	2	0.811
	no	2	6	0.811

The first leaves $\frac{3}{16}(0) + \frac{13}{16}(0.961) = 0.781$ bits, for a gain of 0.219. The second leaves $\frac{8}{16}(0.811) + \frac{8}{16}(0.811) = 0.811$ bits, for a gain of 0.189. The first wins and becomes the root.
The winner is not the more balanced split. It carves off a perfectly pure branch, but only a small one, and still beats a split that makes both branches fairly pure: a pure branch removes all of its own impurity, and its small weight is the price. A split that purified only a single point would lose, because its weight would be too small to matter.

i Watch

The CS229 lecture on trees formulates the splits, from 0:54[a], and compares split criteria, from 8:07[b] [11], Lecture 9.

[a] https://www.youtube.com/watch?v=wr9gUr-eWdA&t=54s
[b] https://www.youtube.com/watch?v=wr9gUr-eWdA&t=487s

i Read

GeeksforGeeks explains information gain[a] [12].

[a] https://www.geeksforgeeks.org/machine-learning/information-gain-and-mutual-information-for-machine-learning/

[2] https://morrison-lab.github.io/ids/entropy.html#rem-information-gain

Exercise 1.1 (Compute the entropy of a skewed distribution)**.** A node holds points of four classes, in the proportions $p = (\frac{1}{2}, \frac{1}{4}, \frac{1}{8}, \frac{1}{8})$. Compute its entropy H in bits, by hand.

Solution

Solution 1.1. Each class costs $-\log_2 p_k$ bits, and the entropy averages those costs with the proportions as weights:

$$\begin{aligned} H &= -\left[\tfrac{1}{2}\log_2\tfrac{1}{2} + \tfrac{1}{4}\log_2\tfrac{1}{4} + 2\cdot\tfrac{1}{8}\log_2\tfrac{1}{8}\right] \\ &= \tfrac{1}{2}(1) + \tfrac{1}{4}(2) + 2\cdot\tfrac{1}{8}(3) \\ &= 0.5 + 0.5 + 0.75 \\ &= 1.75 \text{ bits.} \end{aligned}$$

The common class is cheap to name (1 bit) and the rare ones cost 3 bits each, so the average falls between.

Python

```
import numpy as np

def entropy_py(p):
    # Entropy in bits of a vector of class proportions; terms with p = 0
    # are dropped, since 0 * log2(0) is read as 0.
    p = np.asarray(p, dtype=float)
    p = p[p > 0]
    return float(-np.sum(p * np.log2(p)))

entropy_py([1/2, 1/4, 1/8, 1/8])
```

```
1.75
```

R

```
entropy_bits <- function(p) {
  # Entropy in bits of a vector of class proportions; terms with p = 0
  # are dropped, since 0 * log2(0) is read as 0.
  p <- p[p > 0]
  -sum(p * log2(p))
}

entropy_bits(c(1/2, 1/4, 1/8, 1/8))
```

```
[1] 1.75
```

The code agrees with the hand calculation: $H = 1.75$ bits.

Exercise 1.2 (Compare with the uniform distribution)**.** Now take the same four classes in equal proportions, $p = (\frac{1}{4}, \frac{1}{4}, \frac{1}{4}, \frac{1}{4})$. Compute H.

Solution

Solution 1.2.

$$\begin{aligned} H &= -4\cdot\tfrac{1}{4}\log_2\tfrac{1}{4} \\ &= \log_2 4 \\ &= 2 \text{ bits.} \end{aligned}$$

Exercise 1.3 (Compare with the uniform distribution (part 2))**.** Use the setup of Exercise 1.2. Compare H with that of Exercise 1.1.

Solution

Solution 1.3. This entropy[a] of $H = 2$ bits is more than the 1.75 bits of Solution 1.1.

[a] https://morrison-lab.github.io/pds/expectation.html#def-entropy

Exercise 1.4 (Compare with the uniform distribution (part 3))**.** Use the setup of Exercise 1.2. Which node is easier to describe, and why?

Solution

Solution 1.4. With equal proportions every class is equally likely, so a reader who knows only which node a point fell into has no better guess than any other, and needs the full $\log_2 4 = 2$ bits to name the label. In the skewed node the reader can bet on the common class, so the node is less impure. No distribution[a] over four classes has more than $\log_2 4$ bits, the entropy[b] of the even split.

[a] https://morrison-lab.github.io/pds/random-variables.html#def-cdf
[b] https://morrison-lab.github.io/pds/expectation.html#def-entropy

Exercise 1.5 (Compute the entropy at the root of a small tree)**.** A training set holds all eight combinations of three binary features $x_1, x_2, x_3 \in \{0, 1\}$, labeled by the rule $y = 1$ if both $x_1 = 1$ and $x_2 = 1$, and $y = 0$ otherwise. The feature x_3 plays no part in the rule.

Table 2: The values of x_1, x_2, x_3 and y in each of eight records.

x_1	x_2	x_3	y
0	0	0	0
1	0	0	0
0	1	0	0
1	1	0	1
0	0	1	0
1	0	1	0
0	1	1	0
1	1	1	1

What is the entropy of the labels at the root, before any split?

Solution

Solution 1.5. Two of the eight points have $y = 1$, so the class proportions are $(\frac{1}{4}, \frac{3}{4})$ and

$$\begin{aligned} H(S) &= -\tfrac{1}{4}\log_2 \tfrac{1}{4} - \tfrac{3}{4}\log_2 \tfrac{3}{4} \\ &= 0.5 + 0.311 \\ &= 0.811 \text{ bits.} \end{aligned}$$

```r
# All eight combinations of the three features, with the AND label.
and_data <- expand.grid(x1 = 0:1, x2 = 0:1, x3 = 0:1)
and_data$y <- as.integer(and_data$x1 == 1 & and_data$x2 == 1)

# Entropy in bits of the labels in a node.
node_entropy <- function(y) {
  p <- table(y) / length(y)
  -sum(p * log2(p))
}

node_entropy(and_data$y)
```

```
[1] 0.8112781
```

Code and hand calculation agree: $H(S) = 0.811$ bits.

Exercise 1.6 (Choose the first split by information gain)**.** For the data of Exercise 1.5, compute the information gain of splitting on each of x_1, x_2 and x_3.

Solution

Solution 1.6. Splitting on x_1 sends four points to each branch. The branch $x_1 = 0$ holds only $y = 0$, so $H = 0$. The branch $x_1 = 1$ holds two points of each label, so $H = 1$. The weighted remainder is $\frac{4}{8}(0) + \frac{4}{8}(1) = 0.5$, and

$$\mathrm{IG}(S, x_1) = 0.811 - 0.5 = 0.311 \text{ bits.}$$

The rule is symmetric in x_1 and x_2, so $\mathrm{IG}(S, x_2) = 0.311$ too.
Splitting on x_3 sends four points to each branch, and each branch holds one point with $y = 1$ out of four. Both branches have $H = 0.811$, the same as the root, so the remainder is 0.811 and $\mathrm{IG}(S, x_3) = 0$.

```r
# Information gain of splitting the AND data on one binary feature:
# the root entropy minus the branch entropies, weighted by branch size.
info_gain <- function(data, feature) {
  branches <- split(data$y, data[[feature]])
  weights <- sapply(branches, length) / nrow(data)
  node_entropy(data$y) - sum(weights * sapply(branches, node_entropy))
}

sapply(c("x1", "x2", "x3"), info_gain, data = and_data)
```

```
       x1        x2        x3
0.3112781 0.3112781 0.0000000
```

Exercise 1.7 (Choose the first split by information gain (part 2))**.** Use the setup of Exercise 1.6. Which feature, or features, would the greedy rule split on first?

Solution

Solution 1.7. Features x_1 and x_2 tie, each with a gain of 0.311 bits, so the greedy rule may split on either first. The irrelevant feature x_3 gains nothing.

Exercise 1.8 (Finish the tree)**.** Split on x_1 first, as the gains in Exercise 1.6 allow. Which branch is already pure?

Solution

Solution 1.8. The branch $x_1 = 0$ holds only $y = 0$, so it is a pure leaf that predicts 0.

Exercise 1.9 (Finish the tree (part 2))**.** Use the setup of Exercise 1.8.
For the other branch, compute the information gain of each remaining feature.

Solution

Solution 1.9. The branch $x_1 = 1$ holds four points, two of each label, so its entropy[a] is 1 bit. Splitting it on x_2 gives two pure branches: $x_2 = 0$ holds only $y = 0$ and $x_2 = 1$ holds only $y = 1$. The remainder is 0, so the gain is the full 1 bit. Splitting it on x_3 leaves each branch with one point of each label, so the remainder is 1 and the gain is 0.

```
# Gains inside the x1 = 1 branch, for the two features still available.
branch <- and_data[and_data$x1 == 1, ]
sapply(c("x2", "x3"), info_gain, data = branch)
```

```
x2 x3
 1  0
```

[a] https://morrison-lab.github.io/pds/expectation.html#def-entropy

Exercise 1.10 (Finish the tree (part 3))**.** Use the setup of Exercise 1.8.
Finish the tree that splits on x_1 first.

Solution

Solution 1.10. Splitting on x_2 inside the $x_1 = 1$ branch finishes a tree with these leaves: $y = 0$ when $x_1 = 0$, $y = 0$ when $x_1 = 1$ and $x_2 = 0$, and $y = 1$ when $x_1 = 1$ and $x_2 = 1$.

Exercise 1.11 (Finish the tree (part 4))**.** Use the setup of Exercise 1.8.
How many leaves does the finished tree have?

Solution

Solution 1.11. Splitting on x_1 and then on x_2 gives a tree with three leaves, one for each case of the previous part.

Exercise 1.12 (Finish the tree (part 5))**.** Use the setup of Exercise 1.8.
Does the finished tree classify all eight training points correctly?

Solution

Solution 1.12. That is the rule $y = x_1 \wedge x_2$, so it classifies all eight points correctly, and it ignores x_3, as the greedy rule never rewarded a split on it.

1.3 Regression trees

The same procedure fits numeric targets once we change the impurity measure. A leaf predicts the sample mean of the training targets it holds,

$$\hat{y}_S = \frac{1}{|S|} \sum_{i \in S} y_i$$

so we measure a node's impurity by how far its targets spread around that sample mean:

$$V(S) = \frac{1}{|S|} \sum_{i \in S} (y_i - \hat{y}_S)^2$$

Putting V in place of H in the expression above gives the **variance reduction** of a split,

$$\mathrm{VR}(S, \text{split}) = V(S) - \sum_{b=1}^{B} \frac{|S_b|}{|S|} V(S_b)$$

and a regression tree splits to maximize it. Because the leaf predicts the sample mean, $|S|\,V(S)$ is exactly the squared error that leaf would incur, so maximizing variance reduction is minimizing training squared error one split at a time.

Statistics and salaries of major league baseball players on 322 rows make up the `Hitters` data [13, Sec. 10.9.1]. The data are in `data/hitters.csv`[3], one row per player. The columns this course uses are:

- `Salary`: the outcome[4], the player's salary in thousands of dollars, missing for some players.
- `CHits`: the number of hits during the player's career.
- `Years`: the number of years the player has played in the major leagues.
- The other numeric columns: batting and fielding counts for the previous season and for the career.

Dataset 1: The baseball (`Hitters`) data

1.3.1 R

```r
hitters_all <- readr::read_csv("data/hitters.csv", show_col_types = FALSE)
hitters_tree_data <- hitters_all[!is.na(hitters_all$Salary), ]
hitters_tree_data$log_salary <- log(hitters_tree_data$Salary)
# Grow the tree with no stopping rule,
# then keep the best subtree with two splits.
grow_all <- rpart::rpart.control(cp = 0, minsplit = 2)
hitters_big <- rpart::rpart(
  log_salary ~ Years + Hits,
  data = hitters_tree_data,
  control = grow_all
)
two_splits <- hitters_big$cptable[, "nsplit"] == 2
has_two_splits <- any(two_splits)
stopifnot(has_two_splits)
cp_two_splits <- hitters_big$cptable[two_splits, "CP"]
hitters_tree <- rpart::prune(hitters_big, cp = cp_two_splits)
plot(hitters_tree, margin = 0.1)
text(hitters_tree, digits = 4)
```

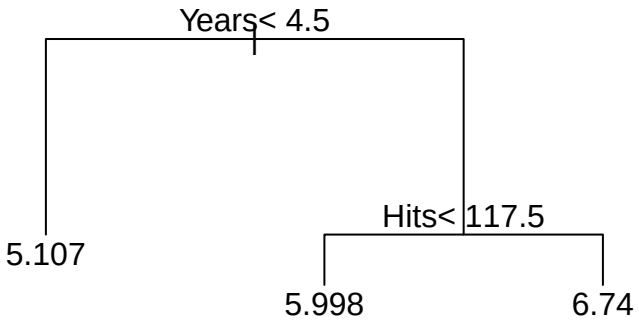
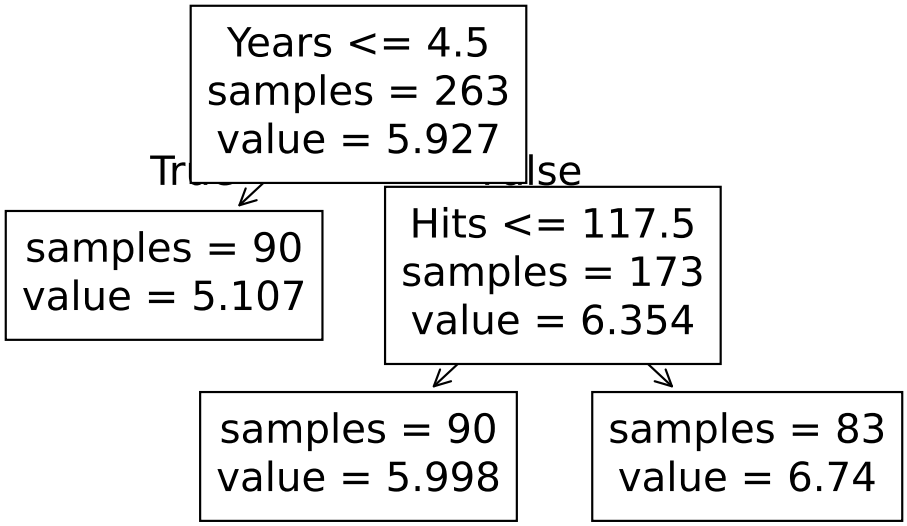


Figure 4: A regression tree for the log salary of a baseball player, from the number of years in the major leagues (`Years`) and the number of hits in the previous season (`Hits`). Each split sends a player who meets its condition to the left. Each leaf shows the sample mean log salary of the players in it.

1.3.2 Python

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.tree import DecisionTreeRegressor, plot_tree

hitters = pd.read_csv("data/hitters.csv").dropna(subset=["Salary"])
X_hitters = hitters[["Years", "Hits"]]
y_hitters = np.log(hitters["Salary"])
# Best-first growth with three leaves keeps the two splits that reduce
# the squared error most.
hitters_tree = DecisionTreeRegressor(max_leaf_nodes=3, random_state=0)
hitters_tree.fit(X_hitters, y_hitters);
fig, ax = plt.subplots(figsize=(8, 4.5));
plot_tree(hitters_tree, feature_names=["Years", "Hits"], precision=3, impurity=False, ax=ax);
```



Example 1.2 (A regression tree for baseball salaries)**.** Take the players of the `Hitters` data whose salary is recorded, and predict the natural log[a] of `Salary` from `Years`, the number of years a player has spent in the major leagues, and `Hits`, the number of hits the player made in the previous season. Salaries are skewed to the right, and the log pulls in their long right tail, so that a histogram[b] of log salaries is closer to a bell shape [13, Sec. 8.1.1]. Figure 4 shows a regression tree with three leaves, grown from the 263 players.

- The root sends a player with `Years < 4.5` to the left leaf, which predicts the sample mean[c] log salary of the 90 players there, 5.107. Salaries are in thousands of dollars, so a log salary ℓ is a salary of $1000\,e^{\ell}$ dollars, here about \$165,174.
- The right branch splits again, on `Hits < 117.5`. Its two leaves predict log salaries of 5.998 and 6.740, about \$402,623 and \$845,561.

`Years` and `Hits` are whole numbers, so each split falls halfway between two adjacent values in the data.

[a] https://morrison-lab.github.io/mds/algebra-exponentials.html#def-logarithm
[b] https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-histogram
[c] https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean

1.3.3 R

```r
plot(
  hitters_tree_data$Years, hitters_tree_data$Hits,
  pch = 20, col = "grey50", xlab = "Years", ylab = "Hits"
)
abline(v = 4.5, lwd = 2)
right_edge <- max(hitters_tree_data$Years) + 1
segments(4.5, 117.5, right_edge, 117.5, lwd = 2)
region_labels <- tibble::tribble(
  ~x, ~y, ~label,
  2.5, 200, "R1",
  15, 40, "R2",
  15, 220, "R3"
)
text(region_labels$x, region_labels$y, region_labels$label, cex = 1.3)
```

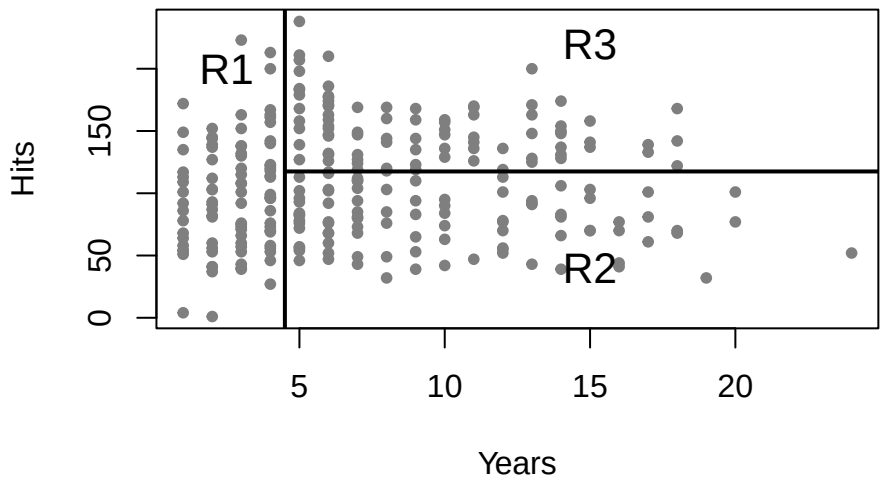
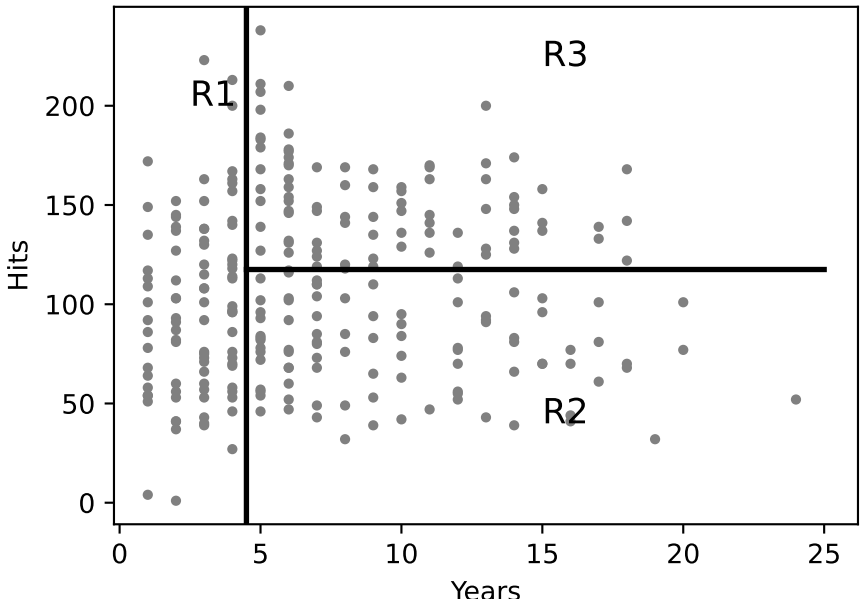


Figure 5: The three regions of the tree in the baseball salary example, in the plane of `Years` and `Hits`, with one point per player. R_1 holds the players with fewer than 4.5 years in the major leagues, R_2 the more experienced players with fewer than 117.5 hits, and R_3 the more experienced players with more hits.

1.3.4 Python

```
fig, ax = plt.subplots(figsize=(5, 3.5));
ax.scatter(hitters["Years"], hitters["Hits"], s=8, color="grey");
ax.axvline(4.5, color="black", linewidth=2);
ax.plot([4.5, hitters["Years"].max() + 1], [117.5, 117.5], color="black", linewidth=2);
for label, x, y in [("R1", 2.5, 200), ("R2", 15, 40), ("R3", 15, 220)]:
    ax.text(x, y, label, fontsize=13);
ax.set_xlabel("Years");
ax.set_ylabel("Hits");
```



Remark (The tree as a partition of the feature space)

Remark 1.1 (The tree as a partition of the feature space). The tree of Example 1.2 cuts the plane of `Years` and `Hits` into three regions, drawn in Figure 5:

$$R_1 = \{\underset{\sim}{x} : \text{Years} < 4.5\},$$
$$R_2 = \{\underset{\sim}{x} : \text{Years} \geq 4.5,\ \text{Hits} < 117.5\},$$
$$R_3 = \{\underset{\sim}{x} : \text{Years} \geq 4.5,\ \text{Hits} \geq 117.5\}.$$

The regions are the leaves of the tree, which James et al. [13, Sec. 8.1.1] also call "terminal nodes". They call the decision nodes "internal nodes", and the segments of the tree that join the nodes "branches". A tree is drawn upside down, with its root at the top and its leaves at the bottom.

Remark (Reading the baseball salary tree)

Remark 1.2 (Reading the baseball salary tree). The tree of Example 1.2 says that `Years` matters most for salary: players with less experience earn less than players with more. Among the less experienced players, the previous season's hits play little part in salary, since the tree does not split their leaf. Among players with five or more years in the major leagues, those with more hits in the previous season tend to earn more.
Three leaves are probably too simple a description of how hits, years and salary are related. Even so, the tree is easier to explain than a linear regression of the same data, and it fits in one small picture [13, Sec. 8.1.1].

Remark (The squared error a regression tree minimizes)

Remark 1.3 (The squared error a regression tree minimizes). Building a regression tree takes two steps [13, Sec. 8.1.1].

1. Divide the possible values of the features $x_1, \ldots, x_p$ into M regions $R_1, \ldots, R_M$ that do not overlap.
2. At each $\underset{\sim}{x}$ in R_m, predict the sample mean[a] $\hat{y}_{R_m}$ of the training targets in R_m.

For example, with two regions whose training targets have sample means 10 and 20, the tree predicts 10 anywhere in R_1 and 20 anywhere in R_2.

Table 3: The best first split of the baseball salary data on each feature: the cutpoint whose split lowers the residual sum of squares of the log salaries most, and the residual sum of squares after that split.

```
# The residual sum of squares after splitting y at each cutpoint of x,
# taken halfway between adjacent values in the data.
split_rss <- function(x, y) { # nolint: function_location_linter
  values <- unique(x)
  values <- sort(values)
  cutpoints <- (head(values, -1) + tail(values, -1)) / 2
  rss <- purrr::map_dbl(cutpoints, function(s) {
    left <- y[x < s]
    right <- y[x >= s]
    left_dev <- left - mean(left)
    right_dev <- right - mean(right)
    sum(left_dev^2) + sum(right_dev^2)
  })
  best <- which.min(rss)
  tibble::tibble(cutpoint = cutpoints[best], rss = rss[best])
}
log_salary <- hitters_tree_data$log_salary
root_dev <- log_salary - mean(log_salary)
root_rss <- sum(root_dev^2)
years_split <- split_rss(hitters_tree_data$Years, log_salary)
hits_split <- split_rss(hitters_tree_data$Hits, log_salary)
first_split <- dplyr::bind_rows(
  Years = years_split,
  Hits = hits_split,
  .id = "feature"
)
first_split
```

```
# A tibble: 2 x 3
  feature cutpoint   rss
  <chr>      <dbl> <dbl>
1 Years        4.5  115.
2 Hits       118.   161.
```

The regions could have any shape. A tree uses boxes, rectangles aligned with the axes, one side per feature, because boxes are simple and a model built from them can be read as a short list of rules. The aim is the boxes whose residual sum of squares is smallest,

$$\text{RSS} = \sum_{m=1}^{M} \sum_{i\,:\,\underset{\sim}{x}_i \in R_m} \left(y_i - \hat{y}_{R_m}\right)^2.$$

There are far too many ways to divide those values into M boxes to check each one, so a tree is grown one split at a time instead.

[a] https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean

1.3.5 R

1.3.6 Python

```
def split_rss(x, y):
    """The best cutpoint of x for splitting y, and the RSS after it."""
    x = x.to_numpy()
```

```
    y = y.to_numpy()
    values = np.unique(x)
    cutpoints = (values[1:] + values[:-1]) / 2
    rss = []
    for s in cutpoints:
        left = y[x < s]
        right = y[x >= s]
        rss.append(((left - left.mean()) ** 2).sum() + ((right - right.mean()) ** 2).sum())
    best = int(np.argmin(rss))
    return cutpoints[best], rss[best]

pd.DataFrame(
    [("Years", *split_rss(hitters["Years"], y_hitters)),
     ("Hits", *split_rss(hitters["Hits"], y_hitters))],
    columns=["feature", "cutpoint", "rss"],
)
```

	feature	cutpoint	rss
0	Years	4.5	115.058475
1	Hits	117.5	160.971530

Definition 1.1 (Recursive binary splitting)**.**
Recursive binary splitting grows a regression tree from training data $(\underset{\sim}{x}_1, y_1), \dots, (\underset{\sim}{x}_N, y_N)$. It starts with all N of them in one region, the root. For a region R, a feature index j and a cutpoint s, write

$$R_1(j,s) = \{\underset{\sim}{x} \in R : x_j < s\}, \qquad R_2(j,s) = \{\underset{\sim}{x} \in R : x_j \geq s\}.$$

The best split of R is the pair (j, s) that minimizes

$$\sum_{i\,:\,\underset{\sim}{x}_i \in R_1(j,s)} \left(y_i - \hat{y}_{R_1}\right)^2 + \sum_{i\,:\,\underset{\sim}{x}_i \in R_2(j,s)} \left(y_i - \hat{y}_{R_2}\right)^2,$$

where $\hat{y}_{R_1}$ and $\hat{y}_{R_2}$ are the sample means[a] of the training targets in $R_1(j,s)$ and $R_2(j,s)$. At each step, the method takes the current region whose best split lowers the residual sum of squares most, and replaces that region by its two halves. It repeats this step until a stopping rule holds, such as that no region holds more than five of the $\underset{\sim}{x}_i$.
The method is top-down because it starts at the root and works toward the leaves. It is greedy because each step takes the best split for that step, without looking ahead for a split that would give a better tree later.
Source: James et al. [13, Sec. 8.1.1].

[a] https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean

Example 1.3 (The first split of the baseball salary tree)**.** Recursive binary splitting starts the tree of Example 1.2 with all 263 players at the root. There, the residual sum of squares of the log[a] salaries around their sample mean[b] is 207.2. Table 3 searches every cutpoint of `Years` and of `Hits`.

- The best split on `Years` is at 4.5, and leaves a residual sum of squares of 115.1.
- The best split on `Hits` is at 117.5, and leaves 161.0.

The split on `Years` lowers the residual sum of squares more, so the root splits on `Years < 4.5`, as in Figure 4.

[a] https://morrison-lab.github.io/mds/algebra-exponentials.html#def-logarithm
[b] https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean

1.3.7 R

```r
four_splits <- hitters_big$cptable[, "nsplit"] == 4
cp_four_splits <- hitters_big$cptable[four_splits, "CP"]
hitters_tree_five <- rpart::prune(hitters_big, cp = cp_four_splits)
years_grid <- seq(1, 24, length.out = 120)
hits_grid <- seq(1, 238, length.out = 120)
n_years <- length(years_grid)
grid <- expand.grid(Years = years_grid, Hits = hits_grid)
grid_pred <- predict(hitters_tree_five, newdata = grid)
pred_matrix <- matrix(grid_pred, nrow = n_years)
shades <- grDevices::hcl.colors(12, "Blues 3", rev = TRUE)
# A pinwheel: five boxes, none of whose edges crosses the whole square.
pinwheel <- tibble::tribble(
  ~x0, ~y0, ~x1, ~y1, ~label,
  0, 0, 0.6, 0.3, "R1",
  0.6, 0, 1, 0.6, "R2",
  0.4, 0.6, 1, 1, "R3",
  0, 0.3, 0.4, 1, "R4",
  0.4, 0.3, 0.6, 0.6, "R5"
)
label_x <- (pinwheel$x0 + pinwheel$x1) / 2
label_y <- (pinwheel$y0 + pinwheel$y1) / 2
unit_range <- c(0, 1)
panel_grid <- c(2, 2)
panel_margins <- c(4, 4, 1, 1)
panel_par <- list(mfrow = panel_grid, mar = panel_margins)
withr::with_par(panel_par, {
  plot(
    unit_range, unit_range,
    type = "n", xlab = "x1", ylab = "x2", asp = 1
  )
  rect(pinwheel$x0, pinwheel$y0, pinwheel$x1, pinwheel$y1, lwd = 2)
  text(label_x, label_y, pinwheel$label)
  image(
    years_grid, hits_grid, pred_matrix,
    col = shades, xlab = "Years", ylab = "Hits"
  )
  points(hitters_tree_data$Years, hitters_tree_data$Hits, pch = 20, cex = 0.5)
  plot(hitters_tree_five, margin = 0.1, uniform = TRUE)
  text(hitters_tree_five, digits = 3, cex = 0.8)
  persp(
    years_grid, hits_grid, pred_matrix,
    theta = 30, phi = 25, xlab = "Years", ylab = "Hits",
    zlab = "Log salary", col = "lightblue", border = NA, shade = 0.5,
    ticktype = "detailed", cex.axis = 0.6
  )
})
```

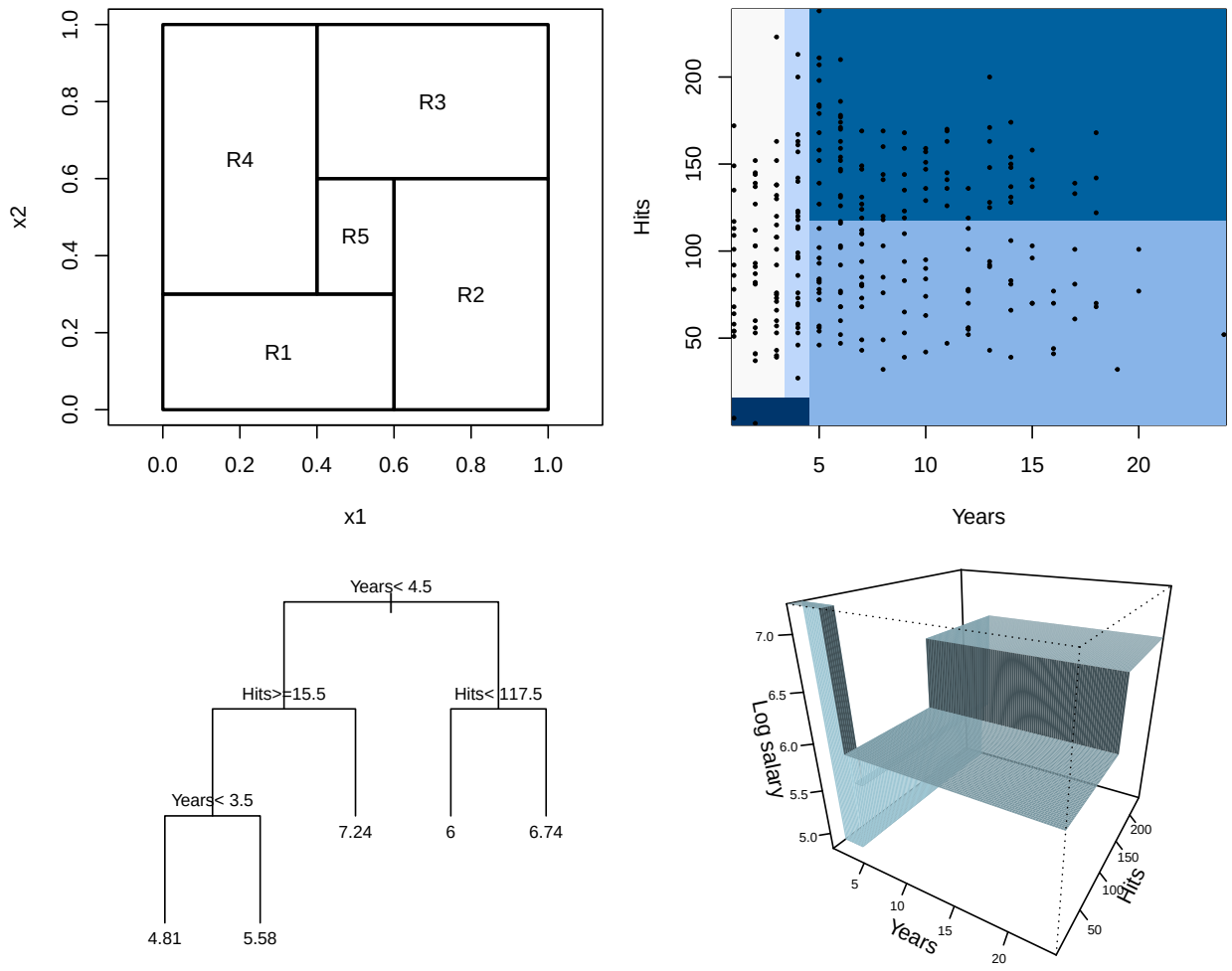


Figure 6: Top left: a partition of a square into five boxes that recursive binary splitting cannot produce, because no single cut crosses the whole square. Top right: the five regions of the baseball salary tree grown to four splits, shaded by the log salary each predicts, with one point per player. Bottom left: that tree. Bottom right: the tree's prediction surface over `Years` and `Hits`.

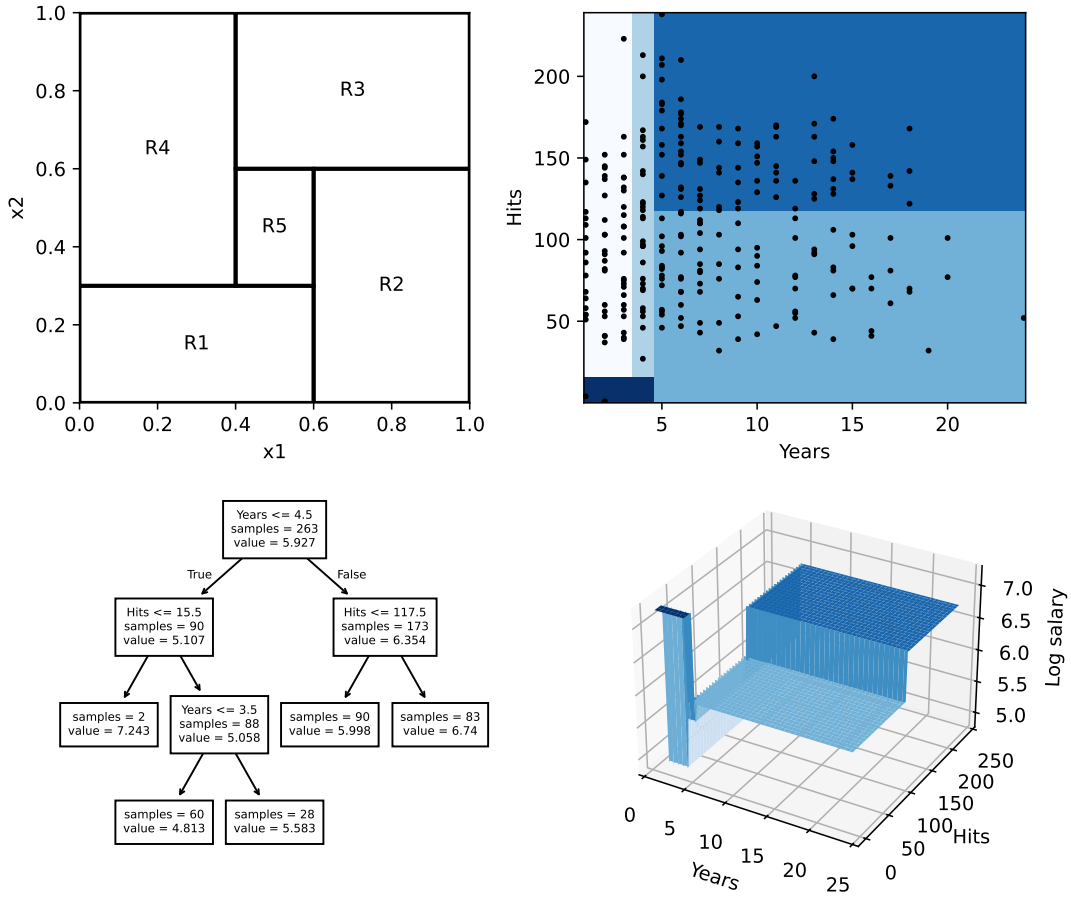
1.3.8 Python

```
from matplotlib.patches import Rectangle

hitters_tree_five = DecisionTreeRegressor(max_leaf_nodes=5, random_state=0)
hitters_tree_five.fit(X_hitters, y_hitters);
years_grid = np.linspace(1, 24, 120)
hits_grid = np.linspace(1, 238, 120)
yy, hh = np.meshgrid(years_grid, hits_grid)
grid = pd.DataFrame({"Years": yy.ravel(), "Hits": hh.ravel()})
pred_grid = hitters_tree_five.predict(grid).reshape(yy.shape)

fig = plt.figure(figsize=(9, 8));
ax = fig.add_subplot(2, 2, 1);
pinwheel = [(0, 0, 0.6, 0.3, "R1"), (0.6, 0, 1, 0.6, "R2"), (0.4, 0.6, 1, 1, "R3"),
            (0, 0.3, 0.4, 1, "R4"), (0.4, 0.3, 0.6, 0.6, "R5")]
```

```
for x0, y0, x1, y1, label in pinwheel:
    ax.add_patch(Rectangle((x0, y0), x1 - x0, y1 - y0, fill=False, linewidth=2));
    ax.text((x0 + x1) / 2, (y0 + y1) / 2, label, ha="center", va="center");
ax.set_aspect("equal");
ax.set_xlabel("x1");
ax.set_ylabel("x2");
ax = fig.add_subplot(2, 2, 2);
ax.pcolormesh(years_grid, hits_grid, pred_grid, cmap="Blues", shading="auto");
ax.scatter(hitters["Years"], hitters["Hits"], s=4, color="black");
ax.set_xlabel("Years");
ax.set_ylabel("Hits");
ax = fig.add_subplot(2, 2, 3);
plot_tree(hitters_tree_five, feature_names=["Years", "Hits"], precision=3, impurity=False, ax=ax, fontsi
ax = fig.add_subplot(2, 2, 4, projection="3d");
ax.plot_surface(yy, hh, pred_grid, cmap="Blues");
ax.set_xlabel("Years");
ax.set_ylabel("Hits");
ax.set_zlabel("Log salary");
```



Watch

Starmer builds regression trees[a] [14].

[a] https://www.youtube.com/watch?v=g9c66TUylZ4

Read

GeeksforGeeks fits a regression tree[a] with scikit-learn [15].

[a] https://www.geeksforgeeks.org/machine-learning/python-decision-tree-regression-using-sklearn/

1.4 When to stop growing

Nothing so far says when to stop, and the greedy rule will not stop on its own. It can keep splitting until every leaf holds a single point, at which stage the training error is zero and the tree has memorized the data rather than learned from it.

Two conditions end the recursion naturally: every leaf is pure, so no split can gain anything; or no candidate split gains enough to justify the extra depth. The first is reachable on any dataset without contradictory labels, and reaching it is the failure mode rather than the goal.

So we limit the growth deliberately instead, with one or both of two simple controls:

- a maximum depth, capping how many comparisons any prediction can involve
- a minimum number of points required in a leaf, or required in a node before it may be split at all

Both say the same thing from opposite directions — do not create a region so small that the points inside it are as likely to be noise as signal — and both are hyperparameters, which we choose by the procedure of Model Selection rather than from the data. Both encode **Occam's razor**, the principle that asks for the simplest hypothesis that fits the observations. For a tree, the simplest hypothesis is the smallest tree.

Read

GeeksforGeeks explains pruning decision trees[a] [16], including the depth and leaf-size limits above.

[a] https://www.geeksforgeeks.org/machine-learning/pruning-decision-trees/

1.5 Pruning

The two rules above stop the tree while it grows. The next two exercises ask whether stopping early can stop too soon.

Exercise 1.13 (Find the best first split when the label is an exclusive or)**.** A label is the exclusive or of two binary features x_1 and x_2 when it is 1 exactly when $x_1 \neq x_2$. Table 4 holds eight points labeled that way, two for each of the four combinations of feature values.

Table 4: Eight points whose label y is the exclusive or of x_1 and x_2, one point per line.

i	x_1	x_2	y
1	0	0	0
2	0	0	0
3	0	1	1
4	0	1	1
5	1	0	1
6	1	0	1

7	1	1	0
8	1	1	0

Compute the information gain of splitting the root on each of x_1 and x_2.

Hint

Choosing a split defines the information gain, and Measuring impurity defines the entropy[a] H that it uses.

[a] https://morrison-lab.github.io/pds/expectation.html#def-entropy

Solution

Solution 1.13. The root holds four points of each label, so $H = 1$ bit. Splitting on x_1 sends the four points with $x_1 = 0$ to one branch. Two of them have $x_2 = 1$ and so $y = 1$, and two have $y = 0$, so that branch also has $H = 1$. The branch $x_1 = 1$ is the same. The weighted remainder is $\frac{4}{8}(1) + \frac{4}{8}(1) = 1$, and

$$\mathrm{IG}(S, x_1) = 1 - 1 = 0 \text{ bits.}$$

Swapping x_1 and x_2 leaves every label unchanged, so $\mathrm{IG}(S, x_2) = 0$ too. No split of the root gains anything.

Exercise 1.14 (Look one split further)**.** Continuing from Exercise 1.13, grow a tree on the eight points labeled by an exclusive or that splits the root on x_1 anyway, and then splits each of the root's two branches on x_2.

Compare the number of training points that this four-leaf tree misclassifies with the number that the root alone misclassifies.

Hint

A leaf of a classification tree predicts the most common label among its training points, as in Decision trees.

Solution

Solution 1.14. The four-leaf tree misclassifies none of the eight training points, and the root alone misclassifies four. Each leaf holds the two points with one combination of x_1 and x_2, and those two points share a label, so every leaf is pure. The root predicts the same label for each of the eight points, and four of the points carry the other label.

A grower that stops when no split gains enough would have stopped at the root, because Solution 1.13 found that no split of the root gains anything. The first split looked worthless only because its value came from the splits below it.

So a rule that stops early can stop too soon. A safer plan grows a large tree first, and then cuts it back, undoing the splits that do not earn their keep.

Definition 1.2 (Cost-complexity pruning, weakest-link pruning)**.** Let T_0 be a tree grown on the training data to predict a numeric target, and let T be any tree obtained by collapsing some of the internal nodes of T_0 into leaves. Write $|T|$ for the number of leaves of T, R_m for the region of its mth leaf, and $\hat{y}_{R_m}$ for the sample mean[a] of the training targets in R_m. **Cost-complexity pruning**, or **weakest-link pruning**, of T_0 with penalty $\lambda \geq 0$ replaces T_0 by the tree T that minimizes the training squared error[b] plus λ per leaf,

$$C_\lambda(T) = \sum_{m=1}^{|T|} \sum_{i\,:\,\underset{\sim}{x}_i \in R_m} \left(y_i - \hat{y}_{R_m}\right)^2 + \lambda |T|.$$

James et al. [17] and Hastie et al. [18] write the penalty as α; these notes write λ, as for the penalties

of Regularization.
Source: James et al. [17, Sec. 8.1.1].

[a] https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample-mean
[b] https://morrison-lab.github.io/pds/expectation.html#def-squared-error-loss

Example 1.4 (Pruning a tree with three leaves)**.**
Consider a tree T_0 that splits its root into a leaf A and an internal node B, and splits B into leaves B_1 and B_2. Table 5 gives the training squared error[a] of each leaf, and of the leaf that collapsing a node would leave in its place.

Table 5: The training squared errors of the leaves of T_0 and of the leaves that collapsing B or the root would leave.

node	training squared error
A	4
B_1	2
B_2	3
B, collapsed	9
the root, collapsed	20

Collapsing internal nodes of T_0 gives three trees, and Table 6 gives their cost-complexity criteria.

Table 6: The trees obtained by collapsing internal nodes of T_0, with their training squared errors and their cost-complexity criteria.

tree	leaves	training squared error	C_λ
T_0	A, B_1, B_2	$4 + 2 + 3 = 9$	$9 + 3\lambda$
T_1	A, B	$4 + 9 = 13$	$13 + 2\lambda$
T_2	the root	20	$20 + \lambda$

T_0 beats T_1 when $9 + 3\lambda < 13 + 2\lambda$, that is, when $\lambda < 4$. T_1 beats T_2 when $13 + 2\lambda < 20 + \lambda$, that is, when $\lambda < 7$. So pruning keeps all of T_0 for $\lambda < 4$, collapses B for $4 < \lambda < 7$, and collapses T_0 to its root for $\lambda > 7$.

[a] https://morrison-lab.github.io/pds/expectation.html#def-squared-error-loss

Exercise 1.15 (Prune a tree whose middle subtree never wins)**.** Consider a tree T_0 that splits its root into an internal node C and a leaf D, and splits C into leaves C_1 and C_2. Table 7 gives the training squared error[a] of each leaf, and of the leaf that collapsing a node would leave in its place.

Table 7: The training squared errors of the leaves of T_0 and of the leaves that collapsing C or the root would leave.

node	training squared error
C_1	1
C_2	1
D	6
C, collapsed	8
the root, collapsed	18

Find, for each penalty $\lambda \geq 0$, the tree that cost-complexity pruning of T_0 chooses.

Hint

List the trees that collapsing internal nodes of T_0 gives, and compare their criteria C_λ as Example 1.4 does.

[a] https://morrison-lab.github.io/pds/expectation.html#def-squared-error-loss

Solution

Solution 1.15. Pruning keeps all of T_0 for $\lambda < 5$, collapses T_0 to its root for $\lambda > 5$, and never chooses the tree that collapses only C.
Collapsing internal nodes of T_0 gives three trees:

Table 8: The trees obtained by collapsing internal nodes of the tree in Table 7, with their training squared errors and their cost-complexity criteria.

tree	leaves	training squared error[a]	C_λ
T_0	C_1, C_2, D	$1 + 1 + 6 = 8$	$8 + 3\lambda$
T_1	C, D	$8 + 6 = 14$	$14 + 2\lambda$
T_2	the root	18	$18 + \lambda$

T_1 beats T_0 only when $14 + 2\lambda < 8 + 3\lambda$, that is, when $\lambda > 6$. T_1 beats T_2 only when $14 + 2\lambda < 18 + \lambda$, that is, when $\lambda < 4$. No λ satisfies both, so T_1 never wins. T_0 beats T_2 when $8 + 3\lambda < 18 + \lambda$, that is, when $\lambda < 5$.
Collapsing C alone raises the training squared error by $14 - 8 = 6$ to remove one leaf, while collapsing the root raises it by $18 - 8 = 10$ to remove two leaves, or 5 per leaf. Collapsing the root raises the training squared error less per leaf removed, so pruning jumps from T_0 straight to the root.

[a] https://morrison-lab.github.io/pds/expectation.html#def-squared-error-loss
[a] https://morrison-lab.github.io/pds/expectation.html#def-squared-error-loss

The trees that cost-complexity pruning chooses get smaller as the penalty grows, each inside the one before, so one pass over T_0 finds them all. The chosen tree loses leaves as λ grows from 0, and each step collapses the internal node whose collapse raises the training squared error[5] least per leaf removed, the weakest link that gives the method its second name [18, Sec. 9.2.2]. Solution 1.15 shows that a step can collapse more than one split at once.

The penalty λ is a hyperparameter, so we choose it by cross-validation and then prune the tree grown on all the training data with the chosen λ [17, Sec. 8.1.1], Algorithm 8.1. A classification tree is pruned the same way, with the training misclassification count, or a node impurity from Measuring impurity, in place of the training squared error; James et al. [17, Sec. 8.1.2] prefer the misclassification count when the goal is accurate prediction.

1.6 CART

CART stands for Classification And Regression Trees.

[5] https://morrison-lab.github.io/pds/expectation.html#def-squared-error-loss

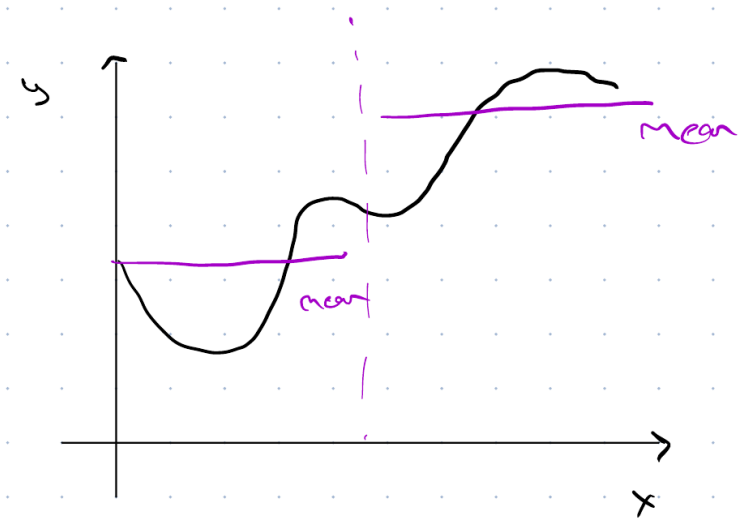


Figure 7: A black curve of y against x, cut by a dashed vertical line into two parts. A purple horizontal segment in each part marks the mean[6] of y there.

- Handles discrete and continuous features.
- Handles classification and regression with the same recursion, changing only the impurity measure.
- Captures non-linear structure, because the regions are assembled by recursive cuts rather than described by one global function.
- Overfits.
- Does not need to store all the data, unlike KNN and kernel smoothing.
- Fast.
- Interpretable — but not really.
- Subject to the CURSE of dimensionality.

The interpretability bullet needs explanation, because trees are usually promoted for it. A single fitted tree genuinely is as readable as a model gets: we can follow any prediction down the tree and report it as the few comparisons that produced it. The reservation concerns what that readability entitles you to conclude. A deep tree's path is long enough that following it is no longer an explanation. The greedy rule commits to one split before considering what comes below, so a small change to the data can change an early split and alter everything under it. And two correlated features compete for the same split, so a feature's absence from the tree is not evidence that it did not matter — only that the greedy rule chose its rival first.

Watch

Weinberger explains the curse of dimensionality[a] [2], Lecture 4.

[a] https://www.youtube.com/watch?v=BbYV8UfMJSA

Read

GeeksforGeeks explains:

- CART[a] [19]
- the curse of dimensionality[b] [20]

[a] https://www.geeksforgeeks.org/machine-learning/cart-classification-and-regression-tree-in-machine-learning/
[b] https://www.geeksforgeeks.org/machine-learning/curse-of-dimensionality-in-machine-learning/

2 Further reading

- James et al. [17], Chapter 8
- Breiman et al. [8] — the book that introduced classification and regression trees
- Zhou [21], Tree-Based Methods[7] — slides on *An Introduction to Statistical Learning*, Chapter 8, with code in both R and Python
- Starmer [22], on decision trees — an illustrated walk-through with little algebra

> **Watch**
>
> The Data Science Learning Community's book clubs discuss *An Introduction to Statistical Learning*, Chapter 8, in recorded meetings:
> - the R book club, in part 1[a] and part 2[b] [23];
> - the Python book club, in a meeting on Chapters 7 and 8[c] and one on Chapter 8[d] [24].
>
> [a] https://www.youtube.com/watch?v=nI3qp-wK58A&list=PL3x6DOfs2NGjlhpKMiWsDcKL2nngwXROq
> [b] https://www.youtube.com/watch?v=uGt4MP2YT8Q&list=PL3x6DOfs2NGjlhpKMiWsDcKL2nngwXROq
> [c] https://www.youtube.com/watch?v=FwiByW3ogwY&list=PL3x6DOfs2NGgbSNh0UG87rrM-z5LiKLZp
> [d] https://www.youtube.com/watch?v=M76X3Wnpzt8&list=PL3x6DOfs2NGgbSNh0UG87rrM-z5LiKLZp

References

[1] T. Hastie, R. Tibshirani, and J. Taylor, "Statistical learning with python." Stanford Online, lecture videos on YouTube, 2023. Available: https://www.youtube.com/playlist?list=PLoROMvodv4rPP6braWoRt5UCXYZ71GZIQ

[2] K. Weinberger, "CS4780: Machine learning for intelligent systems, spring 2017." Cornell University, lecture videos on YouTube, 2017. Available: https://www.youtube.com/playlist?list=PLl8OlHZGYOQ7bkVbuRthEsaLr7bONzbXS

[3] J. Starmer, "Decision and classification trees, clearly explained." StatQuest, video on YouTube, 2021. Available: https://www.youtube.com/watch?v=_L39rN6gz7Y

[4] GeeksforGeeks, "Decision tree in machine learning." GeeksforGeeks, tutorial article, 2026. Available: https://www.geeksforgeeks.org/machine-learning/decision-tree-introduction-example/

[5] N. Jaques, "CSE 446/546: Machine learning, lecture slides." University of Washington, 2026. Accessed: Oct. 12, 2026. [Online]. Available: https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/

[6] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 3, pp. 379–423, 1948, doi: 10.1002/j.1538-7305.1948.tb01338.x[8].

[7] J. R. Quinlan, "Induction of decision trees," *Machine Learning*, vol. 1, no. 1, pp. 81–106, 1986, doi: 10.1007/BF00116251[9].

[8] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and regression trees*. Belmont, CA: Wadsworth, 1984.

[9] J. Starmer, "Entropy (for data science) clearly explained." StatQuest, video on YouTube, 2021. Available: https://www.youtube.com/watch?v=YtebGVx-Fxw

[10] GeeksforGeeks, "Gini impurity and entropy in decision tree." GeeksforGeeks, tutorial article, 2025. Available: https://www.geeksforgeeks.org/machine-learning/gini-impurity-and-entropy-in-decision-tree-ml/

[11] A. Ng *et al.*, "CS229: Machine learning, autumn 2018." Stanford Online, lecture videos on YouTube, 2018. Available: https://www.youtube.com/playlist?list=PLoROMvodv4rMiGQp3WXShtMGgzqpfVfbU

[7] https://ucla-biostat-212a.github.io/2026winter/slides/08-tree/tree.html
[8] https://doi.org/10.1002/j.1538-7305.1948.tb01338.x
[9] https://doi.org/10.1007/BF00116251

[12] GeeksforGeeks, "Information gain and mutual information for machine learning." GeeksforGeeks, tutorial article, 2025. Available: https://www.geeksforgeeks.org/machine-learning/information-gain-and-mutual-information-for-machine-learning/

[13] G. James, D. Witten, T. Hastie, R. Tibshirani, and J. Taylor, *An introduction to statistical learning: With applications in Python*. Springer, 2023. doi: 10.1007/978-3-031-38747-0[10].

[14] J. Starmer, "Regression trees, clearly explained." StatQuest, video on YouTube, 2019. Available: https://www.youtube.com/watch?v=g9c66TUylZ4

[15] GeeksforGeeks, "Implementing decision tree regression using Scikit-Learn." GeeksforGeeks, tutorial article, 2026. Available: https://www.geeksforgeeks.org/machine-learning/python-decision-tree-regression-using-sklearn/

[16] GeeksforGeeks, "Pruning decision trees." GeeksforGeeks, tutorial article, 2026. Available: https://www.geeksforgeeks.org/machine-learning/pruning-decision-trees/

[17] G. James, D. Witten, T. Hastie, and R. Tibshirani, *An introduction to statistical learning with applications in R*, 2nd ed. Springer, 2021. Available: https://www.statlearning.com/

[18] T. Hastie, R. Tibshirani, and J. Friedman, *The elements of statistical learning: Data mining, inference, and prediction*, 2nd ed. Springer, 2009. Available: https://hastie.su.domains/ElemStatLearn/

[19] GeeksforGeeks, "CART (Classification And Regression Tree) in machine learning." GeeksforGeeks, tutorial article, 2025. Available: https://www.geeksforgeeks.org/machine-learning/cart-classification-and-regression-tree-in-machine-learning/

[20] GeeksforGeeks, "Curse of dimensionality in machine learning." GeeksforGeeks, tutorial article, 2025. Available: https://www.geeksforgeeks.org/machine-learning/curse-of-dimensionality-in-machine-learning/

[21] J. Zhou, "Biostat 212A: Statistical machine learning a, winter 2026." UCLA, course webpage with lecture slides, 2026. Available: https://ucla-biostat-212a.github.io/2026winter/

[22] J. Starmer, *The StatQuest illustrated guide to machine learning*. StatQuest Publications, 2022. Available: https://statquest.org/statquest-store/

[23] Data Science Learning Community, "Introduction to statistical learning using R (islr06)." Book club meeting recordings on YouTube, 2023. Available: https://www.youtube.com/playlist?list=PL3x6DOfs2NGjlhpKMiWsDcKL2nngwXROq

[24] Data Science Learning Community, "Introduction to statistical learning using Python (islp03)." Book club meeting recordings on YouTube, 2025. Available: https://www.youtube.com/playlist?list=PL3x6DOfs2NGgbSNh0UG87rrM-z5LiKLZp

[10] https://doi.org/10.1007/978-3-031-38747-0