

Ensemble Methods

Last modified: 2026-10-25

```
# The Python chunks on this page import numpy, pandas and scikit-learn;
# reticulate provisions them.
# The Julia chunks run through JuliaCall.
library(reticulate)
py_require(c("numpy", "pandas", "scipy", "scikit-learn"))
```

i Source

Adapted from Logan Sizemore's Spring 2025 lectures on bagging, random forests and boosting.

Non-Parametric Models ends by listing what is wrong with a decision tree: it overfits. A shallow tree cannot partition the input space finely enough and underfits. A deep one makes a separate region for each noisy point and overfits. The depth that balances the two is rarely deep enough to represent the boundary we wanted. Trees are fast, they handle mixed feature types, and they are about as interpretable as a model gets. On their own, however, they are usually beaten by almost anything else.

Ensembling fixes this. Rather than fitting one model, fit many and combine them [1]. Two strategies do this in opposite ways, and this chapter compares them:

- **Bagging** (short for *bootstrap aggregating*) fits many models *independently* and averages them, to reduce variance
- **Boosting** fits many models *sequentially*, each correcting its predecessor, to reduce bias [2]

i Watch

The CS229 lecture on trees ends with ensemble methods, from 52:53^a [3], Lecture 9.

^a<https://www.youtube.com/watch?v=wr9gUr-eWdA&t=3173s>

i Read

GeeksforGeeks explains ensemble learning^a [4].

^a<https://www.geeksforgeeks.org/machine-learning/a-comprehensive-guide-to-ensemble-learning/>

1 Bagging

Bagging — *bootstrap aggregating*, from Breiman [5] — borrows a basic fact about sample means [6]. The sample mean of several independent measurements of the same quantity varies less than any single one of them. Bagging applies that to predictions: fit the model several times, and average what the copies predict.

There is only one training set, so we cannot collect fresh ones. The bootstrap manufactures stand-ins for them instead.

1.1 The bootstrap

- Draw n points from the training set **with replacement**. The result is a training set of the original size that differs from it: some points appear twice or more, and some do not appear at all
- Repeat B times to get B bootstrap samples, and fit a deep tree to each
- To predict, average the B predictions (regression) or take the majority vote (classification):

$$\hat{f}_{\text{bag}}(x) = \frac{1}{B} \sum_{b=1}^B \hat{f}^{*b}(x)$$

- A given training point is left out of a given bootstrap sample with probability

$$\left(1 - \frac{1}{n}\right)^n \xrightarrow{n \rightarrow \infty} e^{-1} \approx 0.368$$

- so each bootstrap sample contains about 63.2% of the distinct training points. That figure comes up again in Section 1.3, because the other 36.8% turn out to be useful

i Watch

Starmer explains the bootstrap^a [7].

^a<https://www.youtube.com/watch?v=Xz0x-8-cgaQ>

i Read

GeeksforGeeks explains the bootstrap^a [8].

^a<https://www.geeksforgeeks.org/maths/bootstrap-method/>

1.2 Why averaging helps, and how much

- If the B trees were independent, each with variance σ^2 , the sample average would have variance σ^2/B — and we could drive it to zero by taking B large enough
- They are not independent. Each tree learns from an overlapping sample of the same data, so the trees make correlated errors. With pairwise correlation ρ , the variance of the sample average is

$$\rho \sigma^2 + \frac{1-\rho}{B} \sigma^2 \tag{1}$$

- The second term vanishes as B grows, but the first does not. **Adding trees cannot reduce the variance below $\rho \sigma^2$** , and the similarity of the trees to each other sets that lower limit
- This single expression explains the whole design of the next section. To do better than bagging you cannot add more trees — you have to make them less alike
- Bagging reduces variance but leaves bias where it was, so an ensemble of stumps still underfits and an ensemble of unpruned trees can still overfit

1.3 Out-of-bag error

- The 36.8% of points left out of each bootstrap sample are a validation set that needs no extra data:
 - For each training point x_i , collect the trees whose bootstrap sample did *not* contain it — about 37% of them
 - Average those trees' predictions for x_i and score that prediction against y_i
 - Average the resulting point-wise errors over the training set

- This *out-of-bag* (OOB) error estimates generalization error without a held-out split and without refitting. We therefore often tune bagged models on OOB error rather than by cross-validation

1.4 Feature importance

- A single tree can be read off the page; five hundred averaged trees cannot
- Instead, we use an aggregate: for each feature, sum the impurity reduction achieved by every split on that feature, across every tree. Features that repeatedly gave large reductions rank high
- Treat the ranking as a description of *this fitted ensemble*, not as a causal claim. Correlated features split the credit between them arbitrarily, and this particular measure is biased towards high-cardinality features, which offer more split points to try

i Watch

- The CS229 lecture on ensembles covers bagging and random forests, from 1:00:10^a [3], Lecture 9.
- Hastie and Tibshirani cover bagging^b [9], video 8.4, and fit tree ensembles in code in their Python lab^c.
- Weinberger teaches bagging, from 31:50^d [10], Lecture 30.

^a<https://www.youtube.com/watch?v=wr9gUr-eWdA&t=3610s>

^bhttps://www.youtube.com/watch?v=_cKAxjnInfA

^c<https://www.youtube.com/watch?v=AVTfC5WnDTo>

^d<https://www.youtube.com/watch?v=0LB1cy2sCXc&t=1910s>

i Read

GeeksforGeeks explains:

- bagging^a [11]
- feature importance in random forests^b [12]

^a<https://www.geeksforgeeks.org/machine-learning/what-is-bagging-classifier/>

^b<https://www.geeksforgeeks.org/machine-learning/feature-importance-with-random-forests/>

i Read

The University of Washington's CSE 446 slides cover the bootstrap^a [13], Lecture 16.

^a<https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/16-bootstrap.pdf>

1.5 Bagging on real data

Exercise 1.1 (Bagging trees for fuel economy). The Auto data of James et al. [14] give the fuel efficiency (mpg) of 392 cars, with seven other measurements of each car. Download `auto.csv`^a to load it in Python or R. Use every third car (the 1st, 4th, 7th, and so on) as test data and the other cars as training data, and predict mpg from cylinders, displacement, horsepower, weight, acceleration, year and origin.

Fit one fully grown regression tree to the training cars.

^a<https://morrisson-lab.github.io/lds/chapters/data/auto.csv>

Solution

Solution 1.1.

1.5.1 Python

```
import numpy as np
import pandas as pd
from scipy import stats
from sklearn.tree import DecisionTreeRegressor
```

```
auto_py = pd.read_csv("data/auto.csv").dropna()
print(auto_py.head(3))
```

	mpg	cylinders	displacement	...	year	origin	name
0	18.0	8	307.0	...	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	...	70	1	buick skylark 320
2	18.0	8	318.0	...	70	1	plymouth satellite

[3 rows x 9 columns]

```
cols = ["cylinders", "displacement", "horsepower", "weight",
        "acceleration", "year", "origin"]
is_test_py = (np.arange(len(auto_py)) % 3 == 0)
train_py, test_py = auto_py[~is_test_py], auto_py[is_test_py]
```

```
def mean_ci_py(sq):
    # A 95% confidence interval for the mean of the squared errors of the test cars,
    # which are independent of the trees: the mean plus or minus t_crit standard errors.
    n = len(sq)
    half = stats.t.ppf(0.975, df=n - 1) * sq.std(ddof=1) / np.sqrt(n)
    return sq.mean() - half, sq.mean() + half
```

```
single_py = DecisionTreeRegressor(random_state=0).fit(train_py[cols], train_py["mpg"])
sq_single_py = (single_py.predict(test_py[cols]) - test_py["mpg"].to_numpy()) ** 2
mse_single_py = sq_single_py.mean()
# sRMSE is the square root of the sMSE, and so are its interval's endpoints.
print("single tree sRMSE:", np.sqrt(mse_single_py).round(2),
      np.round(np.sqrt(mean_ci_py(sq_single_py)), 2))
```

single tree sRMSE: 3.9 [2.91 4.69]

1.5.2 R

```
library(rpart)
```

```
auto <- read.csv("data/auto.csv")
auto <- tidyr::drop_na(auto)
head(auto, 3)
```

```
# A tibble: 3 x 9
  mpg cylinders displacement horsepower weight acceleration year origin name
<dbl>   <int>      <dbl>      <int>   <int>      <dbl> <int> <int> <chr>
1   18       8        307        130   3504         12    70     1 chev~
2   15       8        350        165   3693        11.5   70     1 buic~
3   18       8        318        150   3436         11    70     1 plym~
```

```

cols <- c("cylinders", "displacement", "horsepower", "weight",
          "acceleration", "year", "origin")
is_test <- (seq_len(nrow(auto)) - 1) %% 3 == 0
train <- auto |>
  dplyr::filter(!is_test) |>
  dplyr::select(mpg, dplyr::all_of(cols))
test <- auto |>
  dplyr::filter(is_test) |>
  dplyr::select(mpg, dplyr::all_of(cols))

grow <- function(data) {
  rpart(mpg ~ ., data = data,
        control = rpart.control(minsplit = 2, cp = 0, xval = 0))
}

# The squared error of each test car, and a 95% confidence interval for their mean.
# The test cars are independent of the trees, so the mean of their squared errors
# is the test error, and its interval is the mean plus or minus t_crit standard errors.
sq_err <- function(pred) (pred - test$mpg)^2
mean_ci <- function(sq) {
  half <- qt(0.975, df = length(sq) - 1) * sd(sq) / sqrt(length(sq))
  c(lower = mean(sq) - half, upper = mean(sq) + half)
}
mse <- function(pred) mean(sq_err(pred))

sq_single <- sq_err(predict(grow(train), test))
mse_single <- mean(sq_single)
c(single_tree = sqrt(mse_single))

single_tree
3.802932

```

The two languages grow trees with different algorithms and draw different bootstrap samples, so their printed errors differ. The numbers below are from the R run.

The single fully grown tree has a test sample root mean squared error (sRMSE) of 3.80 mpg, with a 95% confidence interval of [2.79, 4.60]. The sRMSE is the square root of the sample mean squared error, so it is in the units of mpg and not in squared mpg (why we report sRMSE). Each interval is the t interval for the sample mean of the test cars' squared errors, the sample mean plus or minus 1.98 standard errors, with its endpoints square-rooted.

Exercise 1.2 (Bagging trees for fuel economy (part 2)). Use the setup of Exercise 1.1. Bag $B = 100$ fully grown trees, each fit to a bootstrap sample of the training cars.

Solution

Solution 1.2.

1.5.3 Python

```

rng = np.random.default_rng(571)
preds_py = []
for _ in range(100):
    boot = train_py.iloc[rng.integers(0, len(train_py), len(train_py))]
    tree = DecisionTreeRegressor(random_state=0).fit(boot[cols], boot["mpg"])
    preds_py.append(tree.predict(test_py[cols]))
running_py = np.cumsum(preds_py, axis=0) / np.arange(1, 101)[: , None]
sq_bag_py = (running_py - test_py["mpg"].to_numpy()) ** 2
mse_bag_py = sq_bag_py.mean(axis=1)
print("100 bagged trees sRMSE:", np.sqrt(mse_bag_py[-1]).round(2),
      np.round(np.sqrt(mean_ci_py(sq_bag_py[-1])), 2))

```

100 bagged trees sRMSE: 2.95 [1.95 3.69]

1.5.4 R

```

set.seed(571)
preds <- sapply(1:100, function(b) {
  boot <- dplyr::slice(train, sample(nrow(train), replace = TRUE))
  predict(grow(boot), test)
})
running <- t(apply(preds, 1, cumsum)) / rep(1:100, each = nrow(preds))
mse_bag <- apply(running, 2, mse)
c(bagged_100 = sqrt(mse_bag[100]))

```

```

bagged_100
2.963071

```

```

# The interval at each number of trees, for the figure: one column of running
# predictions per number of trees.
ci_bag <- t(apply(running, 2, function(p) mean_ci(sq_err(p))))
sq_bag_100 <- sq_err(running[, 100])

```

Averaging 100 bootstrap trees gives a test sRMSE of 2.96 mpg ([1.96, 3.70]), which is 22% smaller than the single tree's.

Exercise 1.3 (Bagging trees for fuel economy (part 3)). Use the setup of Exercise 1.1. Plot the test sample mean squared error (sMSE) of the bagged average against the number of trees.

Solution

Solution 1.3. The error falls quickly over the first ten trees and then flattens, as Figure 1 shows.

```

library(ggplot2)
# One row per number of trees: the test sRMSE and its 95% confidence interval.
bag_df <- tibble::tibble(
  trees = 1:100, rmse = sqrt(mse_bag),
  lower = sqrt(ci_bag[, "lower"]), upper = sqrt(ci_bag[, "upper"])
)
single_ci <- sqrt(mean_ci(sq_single))
ggplot(bag_df, aes(trees, rmse)) +
  geom_ribbon(aes(ymin = lower, ymax = upper), alpha = 0.2) +
  geom_line(aes(linetype = "bagged average"), linewidth = 0.9) +
  geom_hline(aes(yintercept = sqrt(mse_single), linetype = "single tree")) +
  geom_hline(yintercept = single_ci, linetype = "dotted", colour = "grey40") +
  scale_linetype_manual(
    values = c("bagged average" = "solid", "single tree" = "dashed")
  )

```

```
) +
labs(x = "number of bagged trees", y = "test sRMSE (mpg)",
     linetype = NULL) +
theme(legend.position = "bottom")
```

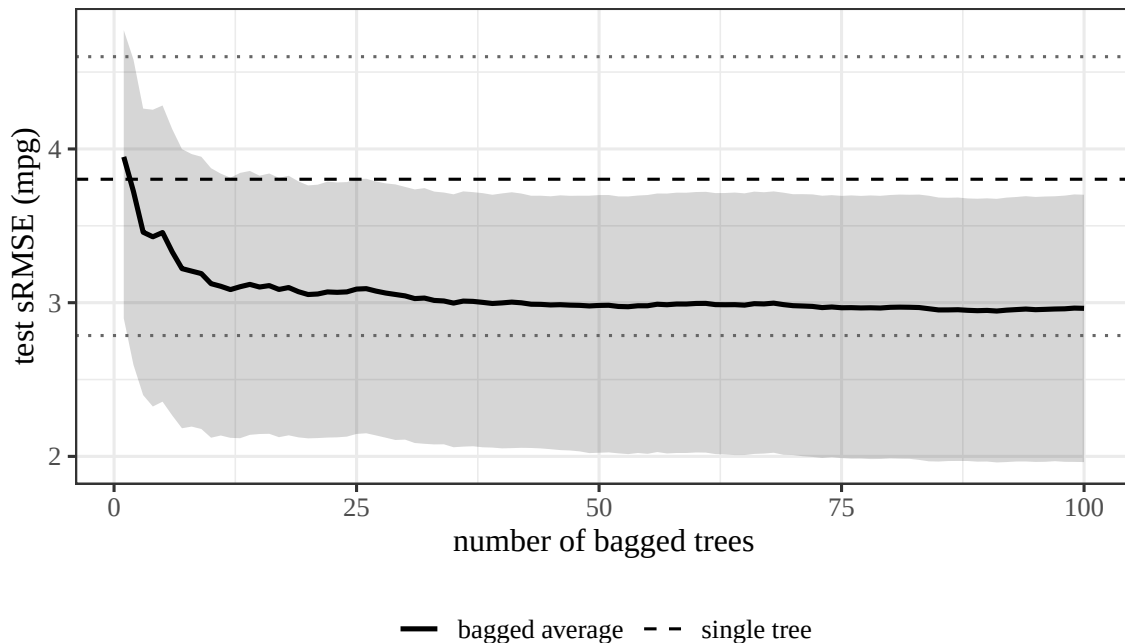


Figure 1: Test sample root mean squared error (sRMSE), in mpg, for predicting mpg from the other measurements of a car, against the number of bagged trees. The shaded band is a 95% confidence interval for each bagged error. The dashed line is a single fully grown tree, and the dotted lines bound its 95% confidence interval.

Exercise 1.4 (Bagging trees for fuel economy (part 4)). Use the setup of Exercise 1.1. Compare the test sMSE of the bagged average with that of the single tree.

Solution

Solution 1.4.

1.5.5 Python

```
# Both models predict the same test cars, so the 95% interval for the difference
# in their errors uses each car's difference in squared error.
print("single tree minus 100 bagged trees, mean squared error:",
      (mse_single_py - mse_bag_py[-1]).round(2),
      np.round(mean_ci_py(sq_single_py - sq_bag_py[-1]), 2))
```

single tree minus 100 bagged trees, mean squared error: 6.52 [2.81 10.23]

1.5.6 R

```
# Both models predict the same test cars, so the interval for the difference in
# their errors uses each car's difference in squared error.
diff_ci <- mean_ci(sq_single - sq_bag_100)
```

The two models predict the same 131 test cars, so the better comparison is of each car's difference in

squared error: the single tree's sMSE is larger by 5.7 mpg² on average, with a 95% confidence interval^a of [2.1, 9.3]. These intervals reflect which test cars were drawn. They do not reflect which training cars were drawn, or which bootstrap samples^b, so a different split could move both errors.

^a<https://morrison-lab.github.io/sds/inference.html#def-confidence-interval>

^b<https://morrison-lab.github.io/sds/exploratory-descriptive.html#def-sample>

2 Random forests

- Bagged trees are correlated because each follows the same greedy rule on nearly the same data: if one feature is strongly predictive, every tree splits on it first, and the trees agree from the root down
- A random forest breaks that agreement with a second source of randomness [15]. Bagging randomizes the *rows*; a random forest also randomizes the *columns*:
 - Fit each tree to its own bootstrap sample, as in bagging
 - **At each split**, draw a random subset of m features and choose the best split from that subset only
- The strongly predictive feature is now unavailable in a fraction of splits, so the trees use other features and diverge. That lowers ρ and therefore lowers the lower limit in Equation 1
- The trade is visible in the same expression: restricting the split candidates makes each individual tree slightly worse, raising σ^2 . The trade helps as long as ρ falls faster than σ^2 rises, and for trees it usually does
- The main settings to choose in advance, where p is the number of features:

Name	What it controls	Typical default
<code>n_estimators</code>	number of trees B	as many as you can afford
<code>max_features</code>	features per split m	$\sqrt{p}$ (classification), $p/3$ (regression)
<code>max_depth</code>	complexity of each tree	unlimited

- B is not a complexity parameter and does not need tuning against a validation set: the variance expression is monotone decreasing in B , so more trees never overfit, they only cost time

i Watch

- Weinberger teaches random forests^a [10], Lecture 31.
- Starmer explains building, using and evaluating random forests^b [16].

^a<https://www.youtube.com/watch?v=4EOCQJgqAOY>

^bhttps://www.youtube.com/watch?v=J4Wdy0Wc_xQ

i Read

GeeksforGeeks explains random forests^a [17].

^a<https://www.geeksforgeeks.org/machine-learning/random-forest-algorithm-in-machine-learning/>

i Read

The University of Washington's CSE 446 slides cover random forests^a [13], Lecture 17.

^a<https://courses.cs.washington.edu/courses/cse446/26sp/schedule/lectures/17-random%20forests.pdf>

3 Boosting

- Bagging and random forests reduce variance by averaging models that each see the whole problem. If every tree finds the same examples hard, averaging them changes nothing — the error is bias, and averaging does not touch bias
- Boosting addresses that case. It fits models **sequentially** and aims each new model at what the current ensemble gets wrong
- We keep the base learners (the individual models we combine) deliberately *weak*, usually shallow trees. The smallest is a stump: one split, two leaves. A stump alone does little better than guessing. Boosting needs nothing more of it, because its guarantee holds for any learner that beats chance by a margin on the weighted data [18]

i Watch

- Hastie and Tibshirani cover boosting^a [9], video 8.5.
- Weinberger teaches boosting^b [10], Lecture 32.

^a<https://www.youtube.com/watch?v=cdl4C2eCOHk>

^b<https://www.youtube.com/watch?v=dosOtgSdbnY>

3.1 AdaBoost

AdaBoost, short for *adaptive boosting*, made boosting practical [18], [19]. Here the labels are $y_i \in \{-1, +1\}$.

- **Initialize** equal weights $w_i^{(1)} = 1/n$ on every training point
- **For** $t = 1, \dots, T$:
 - Fit a weak learner h_t to the training set weighted by $w^{(t)}$
 - Compute its weighted error

$$\varepsilon_t = \sum_{i=1}^n w_i^{(t)} \mathbf{1}[y_i \neq h_t(x_i)]$$

- Compute the learner's coefficient

$$\alpha_t = \frac{1}{2} \ln \left(\frac{1 - \varepsilon_t}{\varepsilon_t} \right)$$

- Reweight and renormalize so the weights again sum to one

$$w_i^{(t+1)} \propto w_i^{(t)} \exp\{-\alpha_t y_i h_t(x_i)\}$$

- **Predict** with the weighted vote of the whole sequence

$$H(x) = \text{sign} \left(\sum_{t=1}^T \alpha_t h_t(x) \right)$$

Three facts follow from those formulas:

- α_t is large when ε_t is small and goes to zero as $\varepsilon_t \rightarrow 1/2$. An accurate learner gets a large weight in the vote, and a random-guessing learner gets none
- The exponent $-\alpha_t y_i h_t(x_i)$ is negative when h_t got point i right and positive when it got it wrong, so correct points lose weight and misclassified points gain it. We fit the next learner to a dataset that emphasizes exactly what the last one missed

- $\varepsilon_t > 1/2$ makes α_t negative. This is not a failure: a learner that is reliably wrong is a useful learner with its sign reversed

i Watch

- Weinberger teaches AdaBoost^a [10], Lecture 34.
- Starmer explains AdaBoost^b [20].

^a<https://www.youtube.com/watch?v=toOAToTaGV4>

^b<https://www.youtube.com/watch?v=LsK-xG1cLYA>

i Read

GeeksforGeeks explains AdaBoost^a [21].

^a<https://www.geeksforgeeks.org/machine-learning/adaboost-in-machine-learning/>

3.2 Gradient boosting

AdaBoost’s reweighting is specific to a particular loss. Gradient boosting generalizes the idea to any differentiable loss [22]. We state the generalization for squared error, where it is easiest to state.

- **Initialize** with the constant that minimizes the loss — for squared error, the sample mean of the training labels:

$$F_0(x) = \bar{y}$$

- **For** $t = 1, \dots, T$:
 - Compute the residuals of the current model

$$r_i^{(t)} = y_i - F_{t-1}(x_i)$$

- Fit a shallow tree h_t **to the residuals**, not to y
- Add it to the model, shrunk by a learning rate ν

$$F_t(x) = F_{t-1}(x) + \nu h_t(x)$$

- We keep the trees shallow, typically three to six levels. Depth is the main way gradient boosting overfits: with deep enough trees, the first few drive the training residuals to zero and every later tree fits noise
- The learning rate ν (commonly 0.1 or smaller) makes the ensemble additive rather than greedy. Small ν needs more trees and generalizes better — ν and T trade against each other, so we should tune them together

3.3 Why “gradient”

- For squared error $L = \frac{1}{2}(y - F(x))^2$, the derivative with respect to the model’s own output is

$$\begin{aligned} \frac{\partial L}{\partial F(x_i)} &= -(y_i - F(x_i)) \\ &= -r_i \end{aligned}$$

- So the residual **is** the negative gradient of the loss with respect to the prediction, and the update $F_t = F_{t-1} + \nu h_t$ is a gradient descent step — in the space of functions rather than the space of parameters

- Replace squared error with another differentiable loss and the procedure is unchanged: fit each tree to the negative gradient of that loss. Squared error is the case where the negative gradient happens to be the familiar residual
- The connection is more useful than it is elegant. It brings in everything known about gradient descent from Gradient Descent — what the learning rate does, why shrinkage acts as regularization, when the procedure converges

i Watch

- The CS229 lecture on ensembles introduces boosting, from 1:11:08^a [3], Lecture 9.
- Starmer explains gradient boosting^b [23] and works through the details for regression^c [24].

^a<https://www.youtube.com/watch?v=wr9gUr-eWdA&t=4268s>

^b<https://www.youtube.com/watch?v=3CC4N4z3GJc>

^c<https://www.youtube.com/watch?v=2xudPOBz-vs>

i Read

GeeksforGeeks explains gradient boosting^a [25].

^a<https://www.geeksforgeeks.org/machine-learning/ml-gradient-boosting/>

4 Bagging against boosting

	Bagging / random forest	Boosting
Models fitted	independently, in parallel	sequentially, each on the last's errors
Base learner	deep, low-bias, high-variance	shallow, high-bias, low-variance
Chiefly reduces	variance	bias
More models	never overfits, just costs time	<i>can</i> overfit; T needs tuning
Parallelizes	trivially	not across t
Built-in validation	out-of-bag error	none

The two are complementary rather than competing. A random forest is the safer default — it has one hyperparameter that matters, and running it longer cannot make it worse. A carefully tuned gradient-boosted ensemble usually predicts best on tabular data, but it needs a tuning budget.

i Read

GeeksforGeeks compares bagging and boosting^a [26].

^a<https://www.geeksforgeeks.org/machine-learning/bagging-vs-boosting-in-machine-learning/>

5 Ensembling neural networks

- The averaging behind bagging is not specific to trees. Training the same network several times from different initializations and averaging the predictions reliably improves both accuracy and calibration, for the same variance-reduction reason
- **Dropout** makes this affordable. During training, dropout zeroes each unit independently with probability $1 - p$ on each forward pass, so every example flows through a different subnetwork

- A network with n units has 2^n possible subnetworks, and all of them share one set of weights. Training with dropout therefore trains a huge ensemble, and using the full network at the end approximates averaging it. The cost is that of a single network
- Three details decide whether an implementation is correct:
 - We scale a kept unit's output by $1/p$ during training, so that the expected input to the next layer does not change with p
 - We switch dropout off when we predict. Every unit is active at test time, and we use the outputs as they are. (Some libraries do the opposite and rescale at test time instead of during training. The two conventions put the factor in different places but give the same trained model)
 - The letter p means opposite things in different sources. Here, and in the formulation that introduced dropout, p is the probability a unit is **kept**, so the scale factor is $1/p$. PyTorch's `nn.Dropout` takes p as the probability a unit is **zeroed**, and scales by $1/(1 - p)$. The mechanism is the same either way, but the letter is reversed, so `nn.Dropout(0.2)` is this section's $p = 0.8$

i Watch

Hutchinson teaches dropout^a (12 min) as an ensemble [27]. The video site requires a login, which enrolled students find in the syllabus.

^ahttps://facultyweb.cs.wvu.edu/~hutchib2/video_lectures/data371/#dropout

i Read

GeeksforGeeks explains dropout^a [28].

^a<https://www.geeksforgeeks.org/machine-learning/dropout-in-neural-networks/>

6 AdaBoost in code

The code in this section runs AdaBoost with decision stumps on a one-dimensional problem that no single stump can solve: the label is $+1$ when $|x| > 1$ and -1 otherwise, so the positive class lies on *both* sides of the negative one. The best possible single stump misclassifies three of the nine points. Look at the training error of the weighted vote as rounds accumulate.

6.1 Python

```
import numpy as np

def best_stump(x, y, w):
    """Weighted-error-minimizing stump over thresholds and both polarities."""
    best = None
    for threshold in np.unique(x):
        for polarity in (1, -1):
            pred = np.where(polarity * (x - threshold) >= 0, 1.0, -1.0)
            err = w[pred != y].sum()
            if best is None or err < best[0]:
                best = (err, threshold, polarity)
    return best

def adaboost(x, y, rounds):
    w = np.full(len(x), 1 / len(x))
```

```

stumps = []
for _ in range(rounds):
    err, threshold, polarity = best_stump(x, y, w)
    err = min(max(err, 1e-10), 1 - 1e-10)
    alpha = 0.5 * np.log((1 - err) / err)
    pred = np.where(polarity * (x - threshold) >= 0, 1.0, -1.0)
    w = w * np.exp(-alpha * y * pred)
    w = w / w.sum()
    stumps.append((alpha, threshold, polarity))
    margin = sum(a * np.where(p * (x - t) >= 0, 1.0, -1.0)
                 for a, t, p in stumps)
    print(len(stumps), round(err, 3),
          round(np.mean(np.sign(margin) != y), 3))

x = np.array([-3.0, -2.0, -1.5, -0.5, 0.0, 0.5, 1.5, 2.0, 3.0])
y = np.where(np.abs(x) > 1, 1.0, -1.0)
adaboost(x, y, 4)

```

```

1 0.333 0.333
2 0.25 0.333
3 0.167 0.0
4 0.2 0.0

```

6.2 R

```

stump_predict <- function(x, threshold, polarity) {
  ifelse(polarity * (x - threshold) >= 0, 1, -1)
}

best_stump <- function(x, y, w) {
  best <- NULL
  for (threshold in unique(x)) {
    for (polarity in c(1, -1)) {
      err <- sum(w[stump_predict(x, threshold, polarity) != y])
      if (is.null(best) || err < best$err) {
        best <- list(err = err, threshold = threshold, polarity = polarity)
      }
    }
  }
  best
}

adaboost <- function(x, y, rounds) {
  w <- rep(1 / length(x), length(x))
  margin <- rep(0, length(x))
  for (step in seq_len(rounds)) {
    fit <- best_stump(x, y, w)
    err <- min(max(fit$err, 1e-10), 1 - 1e-10)
    alpha <- 0.5 * log((1 - err) / err)
    pred <- stump_predict(x, fit$threshold, fit$polarity)
    w <- w * exp(-alpha * y * pred)
    w <- w / sum(w)
    margin <- margin + alpha * pred
    cat(step, round(err, 3), round(mean(sign(margin) != y), 3), "\n")
  }
}

```

```

}

x <- c(-3, -2, -1.5, -0.5, 0, 0.5, 1.5, 2, 3)
y <- ifelse(abs(x) > 1, 1, -1)
adaboost(x, y, 4)

```

```

1 0.333 0.333
2 0.25 0.333
3 0.167 0
4 0.2 0

```

6.3 Julia

```

stump_predict(x, threshold, polarity) =
    [polarity * (xi - threshold) >= 0 ? 1.0 : -1.0 for xi in x];

function best_stump(x, y, w)
    best = nothing
    for threshold in unique(x), polarity in (1, -1)
        pred = stump_predict(x, threshold, polarity)
        err = sum(w[pred .!= y])
        if best === nothing || err < best[1]
            best = (err, threshold, polarity)
        end
    end
    best
end;

function adaboost(x, y, rounds)
    w = fill(1 / length(x), length(x))
    margin = zeros(length(x))
    for r in 1:rounds
        err, threshold, polarity = best_stump(x, y, w)
        err = min(max(err, 1e-10), 1 - 1e-10)
        alpha = 0.5 * log((1 - err) / err)
        pred = stump_predict(x, threshold, polarity)
        w = w .* exp(-alpha .* y .* pred)
        w = w ./ sum(w)
        margin = margin .+ alpha .* pred
        println(r, " ", round(err, digits = 3), " ",
                round(sum(sign.(margin) .!= y) / length(y), digits = 3))
    end
end;

x = [-3.0, -2.0, -1.5, -0.5, 0.0, 0.5, 1.5, 2.0, 3.0];
y = [abs(xi) > 1 ? 1.0 : -1.0 for xi in x];
adaboost(x, y, 4)

```

```

1 0.333 0.333
2 0.25 0.333
3 0.167 0.0
4 0.2 0.0

```

The first stump's weighted error is $1/3$ — the best a single split can manage on this data — and the ensemble's training error reaches zero within a few rounds. We keep `err` away from 0 and 1 before taking the logarithm: a weak learner that happens to be perfect on the current weights would otherwise send α_t to infinity.

7 Further reading

- Breiman [5] — bagging, and why averaging reduces variance
- Breiman [29] — decorrelating the trees as well as resampling
- Schapire [30] — the first boosting algorithm, seven years before AdaBoost, and the proof that weak learnability implies strong
- Freund and Schapire [18] — AdaBoost, and the decision-theoretic argument behind the weight update
- Friedman [31] — the function-space gradient-descent view
- Zhou [32], Tree-Based Methods¹ — slides that go from a single tree through bagging and random forests to AdaBoost and gradient boosting, with code in both R and Python
- Hastie et al. [33], Chapters 10 and 15

i Watch

Section 8.2 of *An Introduction to Statistical Learning* covers bagging, random forests and boosting. The Data Science Learning Community’s book clubs discuss its Chapter 8 in recorded meetings:

- the R book club, in part 1^a and part 2^b [34];
- the Python book club, in a meeting on Chapters 7 and 8^c and one on Chapter 8^d [35].

^a<https://www.youtube.com/watch?v=nI3qp-wK58A&list=PL3x6DOfs2NGjIhpKMiWsDcKL2nngwXROq>

^b<https://www.youtube.com/watch?v=uGt4MP2YT8Q&list=PL3x6DOfs2NGjIhpKMiWsDcKL2nngwXROq>

^c<https://www.youtube.com/watch?v=FwiByW3ogwY&list=PL3x6DOfs2NGGbSNh0UG87rrM-z5LiKLZp>

^d<https://www.youtube.com/watch?v=M76X3Wnpzt8&list=PL3x6DOfs2NGGbSNh0UG87rrM-z5LiKLZp>

References

- [1] Wikipedia contributors, “Ensemble learning — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Ensemble_learning
- [2] Wikipedia contributors, “Boosting (machine learning) — Wikipedia, the free encyclopedia.” 2026. Available: [https://en.wikipedia.org/wiki/Boosting_\(machine_learning\)](https://en.wikipedia.org/wiki/Boosting_(machine_learning))
- [3] A. Ng *et al.*, “CS229: Machine learning, autumn 2018.” Stanford Online, lecture videos on YouTube, 2018. Available: <https://www.youtube.com/playlist?list=PLoROMvodv4rMiGQp3WXShtMGgzqpfVfbU>
- [4] GeeksforGeeks, “Ensemble learning.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/a-comprehensive-guide-to-ensemble-learning/>
- [5] L. Breiman, “Bagging predictors,” *Machine Learning*, vol. 24, no. 2, pp. 123–140, 1996, doi: 10.1007/BF00058655².
- [6] Wikipedia contributors, “Bootstrap aggregating — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Bootstrap_aggregating
- [7] J. Starmer, “Bootstrapping main ideas.” StatQuest, video on YouTube, 2021. Available: <https://www.youtube.com/watch?v=Xz0x-8-cgaQ>
- [8] GeeksforGeeks, “Bootstrap method.” GeeksforGeeks, tutorial article, 2025. Available: <https://www.geeksforgeeks.org/maths/bootstrap-method/>
- [9] 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>
- [10] K. Weinberger, “CS4780: Machine learning for intelligent systems, spring 2017.” Cornell University, lecture videos on YouTube, 2017. Available: <https://www.youtube.com/playlist?list=PLl8OIHZGYOQ7bkVbuRthEsaLr7bONzbXS>
- [11] GeeksforGeeks, “Bagging classifier.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/what-is-bagging-classifier/>

¹<https://ucla-biostat-212a.github.io/2026winter/slides/08-tree/tree.html>

²<https://doi.org/10.1007/BF00058655>

- [12] GeeksforGeeks, “Feature importance with random forests.” GeeksforGeeks, tutorial article, 2025. Available: <https://www.geeksforgeeks.org/machine-learning/feature-importance-with-random-forests/>
- [13] 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/>
- [14] 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/>
- [15] Wikipedia contributors, “Random forest — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Random_forest
- [16] J. Starmer, “Random forests part 1 - building, using and evaluating.” StatQuest, video on YouTube, 2018. Available: https://www.youtube.com/watch?v=J4Wdy0Wc_xQ
- [17] GeeksforGeeks, “Random forest algorithm in machine learning.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/random-forest-algorithm-in-machine-learning/>
- [18] Y. Freund and R. E. Schapire, “A decision-theoretic generalization of on-line learning and an application to boosting,” *Journal of Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, 1997, doi: 10.1006/jcss.1997.1504³.
- [19] Wikipedia contributors, “AdaBoost — Wikipedia, the free encyclopedia.” 2026. Available: <https://en.wikipedia.org/wiki/AdaBoost>
- [20] J. Starmer, “AdaBoost, clearly explained.” StatQuest, video on YouTube, 2019. Available: <https://www.youtube.com/watch?v=LsK-xG1cLYA>
- [21] GeeksforGeeks, “AdaBoost in machine learning.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/adaboost-in-machine-learning/>
- [22] Wikipedia contributors, “Gradient boosting — Wikipedia, the free encyclopedia.” 2026. Available: https://en.wikipedia.org/wiki/Gradient_boosting
- [23] J. Starmer, “Gradient boost, part 1: Regression main ideas.” StatQuest, video on YouTube, 2019. Available: <https://www.youtube.com/watch?v=3CC4N4z3GJc>
- [24] J. Starmer, “Gradient boost part 2 (of 4): Regression details.” StatQuest, video on YouTube, 2019. Available: <https://www.youtube.com/watch?v=2xudPOBz-vs>
- [25] GeeksforGeeks, “Gradient boosting in ML.” GeeksforGeeks, tutorial article, 2025. Available: <https://www.geeksforgeeks.org/machine-learning/ml-gradient-boosting/>
- [26] GeeksforGeeks, “Bagging vs boosting in machine learning.” GeeksforGeeks, tutorial article, 2026. Available: <https://www.geeksforgeeks.org/machine-learning/bagging-vs-boosting-in-machine-learning/>
- [27] B. Hutchinson, “DATA 471/571 (machine learning) and CSCI 481/581 (deep learning) video lectures.” Western Washington University. Accessed: Sep. 28, 2026. [Online]. Available: https://facultyweb.cs.wvu.edu/~hutchib2/video_lectures/data371/
- [28] GeeksforGeeks, “Dropout in neural networks.” GeeksforGeeks, tutorial article, 2025. Available: <https://www.geeksforgeeks.org/machine-learning/dropout-in-neural-networks/>
- [29] L. Breiman, “Random forests,” *Machine Learning*, vol. 45, no. 1, pp. 5–32, 2001, doi: 10.1023/A:1010933404324⁴.
- [30] R. E. Schapire, “The strength of weak learnability,” *Machine Learning*, vol. 5, no. 2, pp. 197–227, 1990, doi: 10.1007/BF00116037⁵.
- [31] J. H. Friedman, “Greedy function approximation: A gradient boosting machine,” *The Annals of Statistics*, vol. 29, no. 5, pp. 1189–1232, 2001, doi: 10.1214/aos/1013203451⁶.
- [32] 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/>
- [33] 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/>
- [34] 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=PL3x6DOfs2NGjIhpKMiWsDcKL2nngwXROq>

³<https://doi.org/10.1006/jcss.1997.1504>

⁴<https://doi.org/10.1023/A:1010933404324>

⁵<https://doi.org/10.1007/BF00116037>

⁶<https://doi.org/10.1214/aos/1013203451>

- [35] 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>