Data Science for Economics

Lecture 4: Decision Trees Contd.

Dawie van Lill

Packages and topics

Here are the packages that we will be using for this session.

```
if (!require("pacman")) install.packages("pacman")
pacman::p_load(dplyr, ggplot2, rpart, caret, rpart.plot,
               vip, pdp, doParallel, foreach, tidyr,
               ipred, ranger, gbm, xgboost, AmesHousing)
```

New topics for today

1. Random forests
2. Gradient boosting

Random

Random forests

Random forests are a modification of bagged decision trees.

Build large collection of de-correlated trees to improve predictive performance.

When bagging trees, a problem still exists.

Model building steps independent but trees in bagging are not independent.

This characteristic is known as **tree correlation**.

Random forests help to reduce tree correlation.

Injects more randomness into the tree-growing process.

Uses **split-variable randomization**

- Each time split is performed, search for the split variable is limited to random subset of m_{try} of original p features
- Defaults values are $m_{try} = \frac{p}{3}$ (regression) and $m_{try} = \sqrt{p}$ (classification)

Random forest algorithm

```
Given a training data set
Select number of trees to build (n_trees)
for i = 1 to n_trees do
|  Generate a bootstrap sample of the original data
|  Grow a regression/classification tree to the bootstrapped data
|  for each split do
|  | Select m_try variables at random from all p variables
|  | Pick the best variable/split-point among the m_try
|  | Split the node into two child nodes
|  end
| Use typical tree model stopping criteria to determine when a
| tree is complete (but do not prune)
end
Output ensemble of trees
```

When $m_{try} = p$ we have bagging.

Algorithm randomly selects bootstrap sample to train on **and** a random sample of features to use for each split.

Lessens tree correlation and increases predictive power.

Ames housing example (OOB)

```r
# Create training (70%) set for the Ames housing data.
set.seed(123)
ames <- AmesHousing::make_ames()
split  <- rsample::initial_split(ames, prop = 0.7,
                                 strata = "Sale_Price")
ames_train  <- rsample::training(split)

# number of features
n_features <- length(setdiff(names(ames_train), "Sale_Price"))

# train a default random forest model
ames_rf1 <- ranger(
  Sale_Price ~ .,
  data = ames_train,
  mtry = floor(n_features / 3),
  respect.unordered.factors = "order",
  seed = 123)

# get OOB RMSE
(default_rmse <- sqrt(ames_rf1$prediction.error))
```

```
## [1] 25896.47
```

Hyperparameters

There are several tunable hyperparameters that we can consider in this model.

The main hyperparameters to consider include:

1. The number of trees in the forest
2. The number of features to consider at any given split: m_{try}
3. The complexity of each tree
4. The sampling scheme
5. The splitting rule to use during tree construction

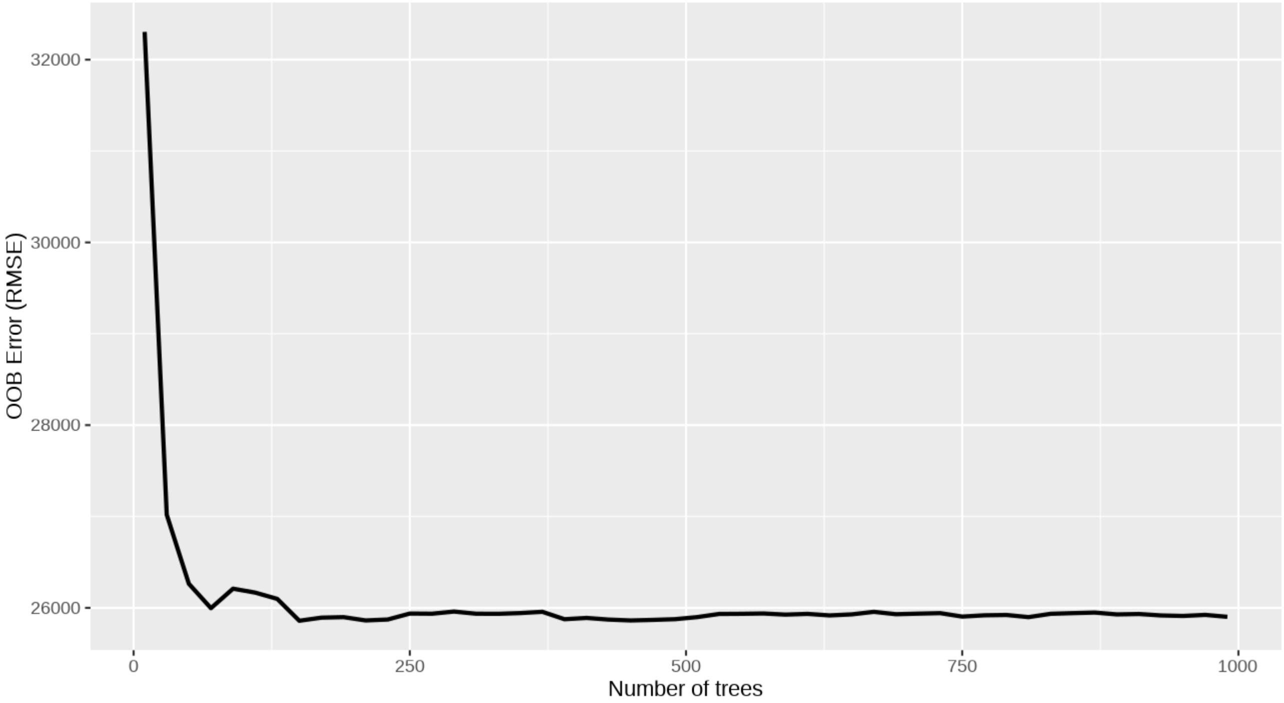
We will quickly discuss each of these hyperparameters.

Grid search can also be used to find best combination of hyperparameters.

We will briefly touch on grid search.

Number of trees

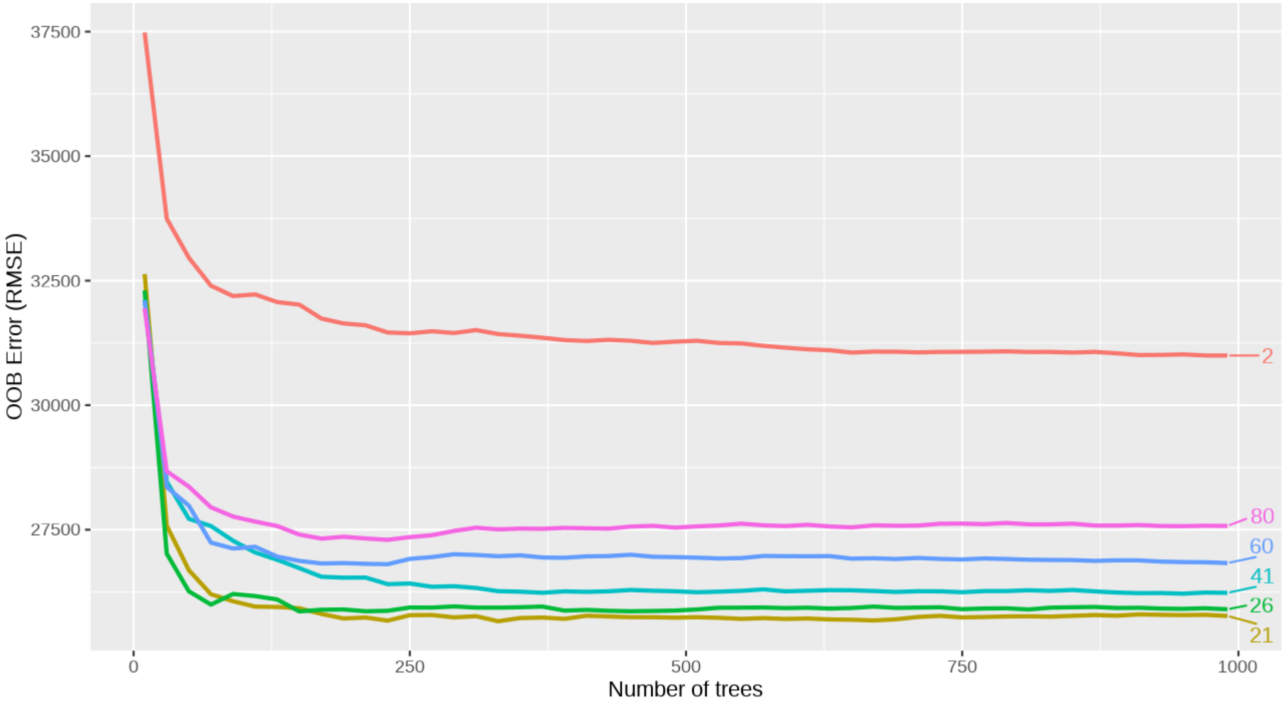
Number of trees needs to be sufficiently large to stabilize the error rate



Good rule of thumb is to start with 10 times number of features.

Split-variable randomization

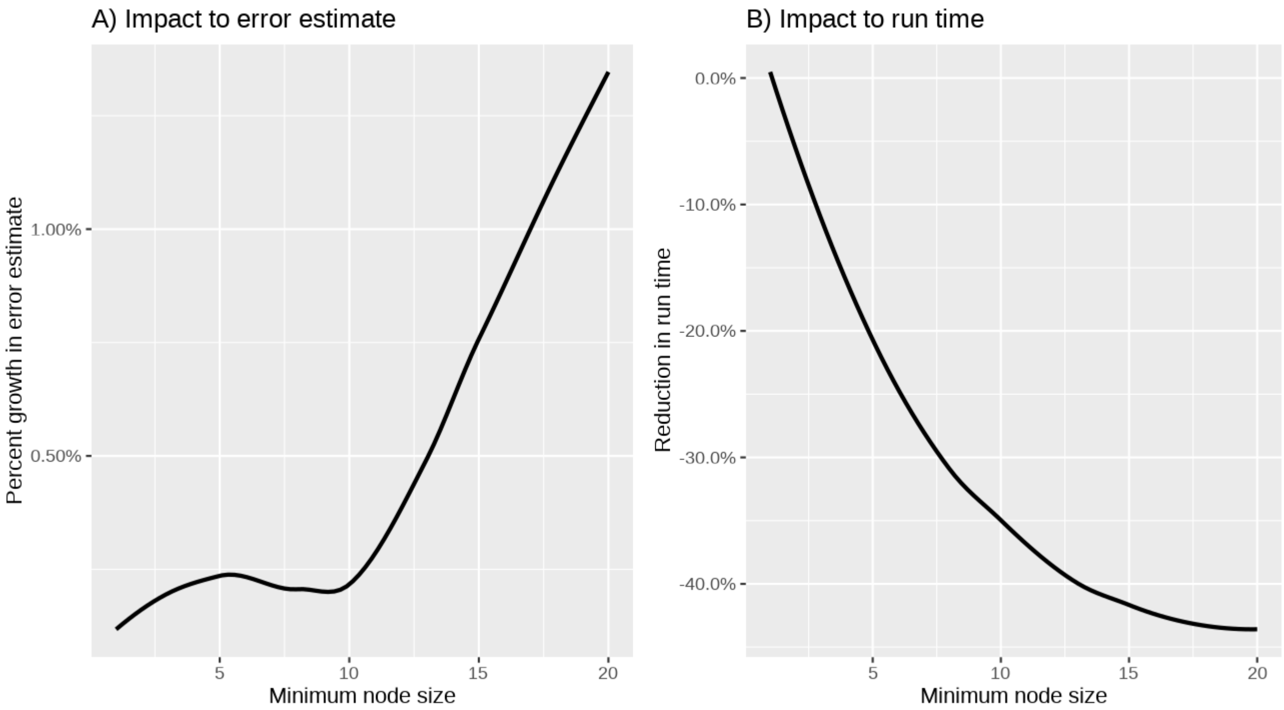
Hyperparameter m_{try} controls split-variable randomization.



Few (many) relevant predictors, higher (lower) m_{try}

Tree complexity

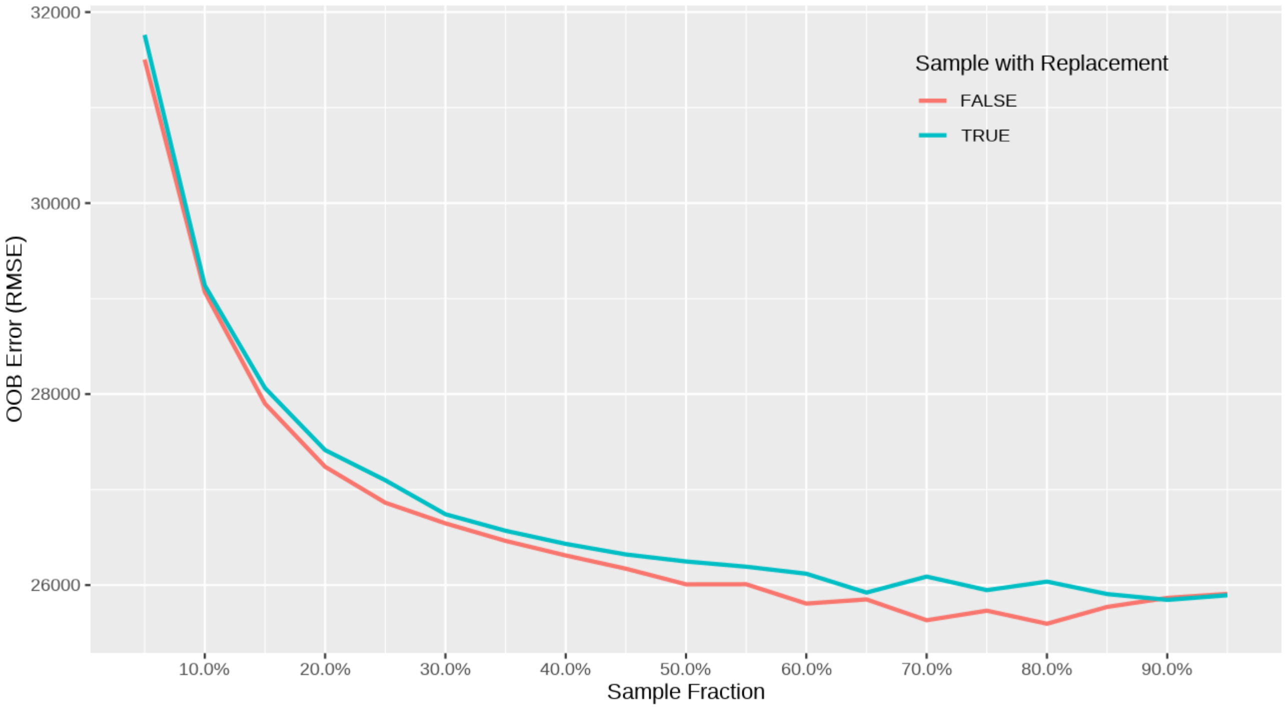
We can control depth and complexity of individual trees with **node size** the most common to control complexity.



If computation time a concern then try increasing node size!

Sampling schedule

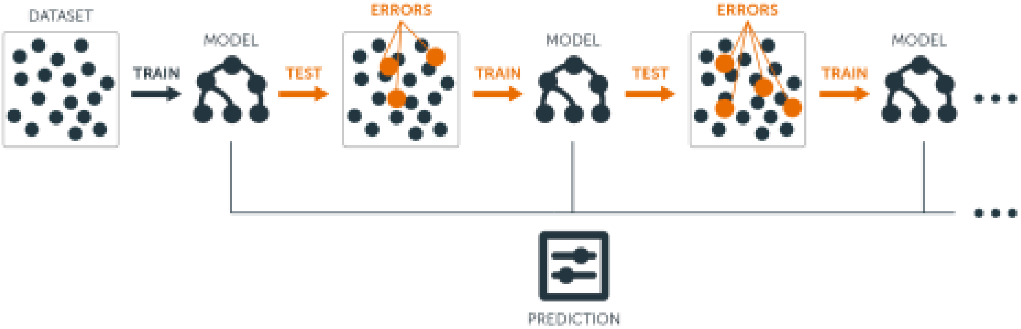
Can also adjust sample size and decide on whether to sample with or without replacement.



Gradient boosting

GBMs build an ensemble of **shallow trees in sequence** with each tree learning and improving on the previous one.

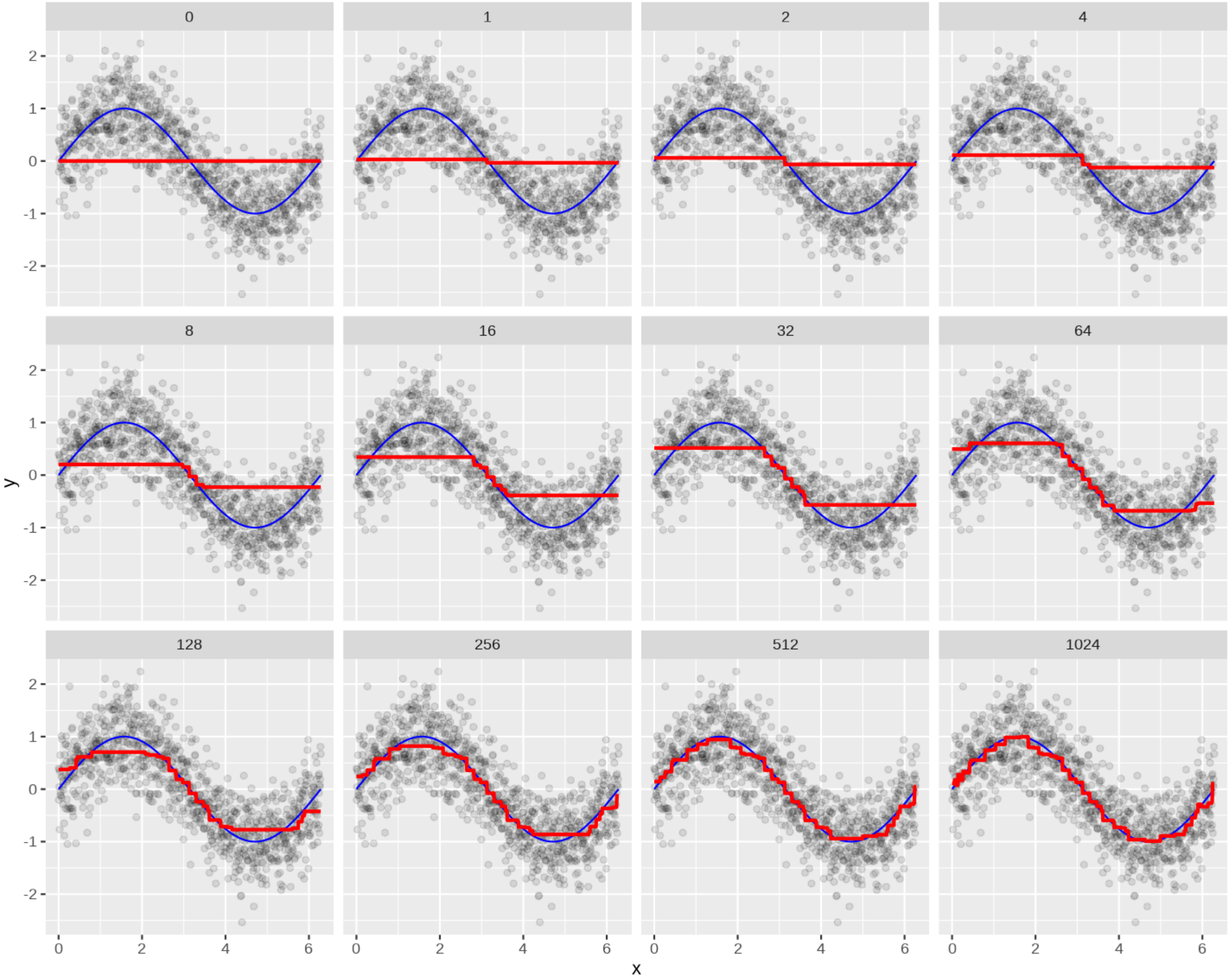
The main idea of boosting is to add new models to the ensemble **sequentially**.



Boosting attacks bias-variance-tradeoff by starting with a *weak* model.

Boosts the performance by building new trees.

- New tree in sequence tries to fix where previous one made biggest mistakes.
- Focus on training rows with largest prediction errors.


0
1
2
4
8
16
32
64
128
256
512
1024
y
x
2
1
0
-1
-2
0
2
4
6

Gradient descent

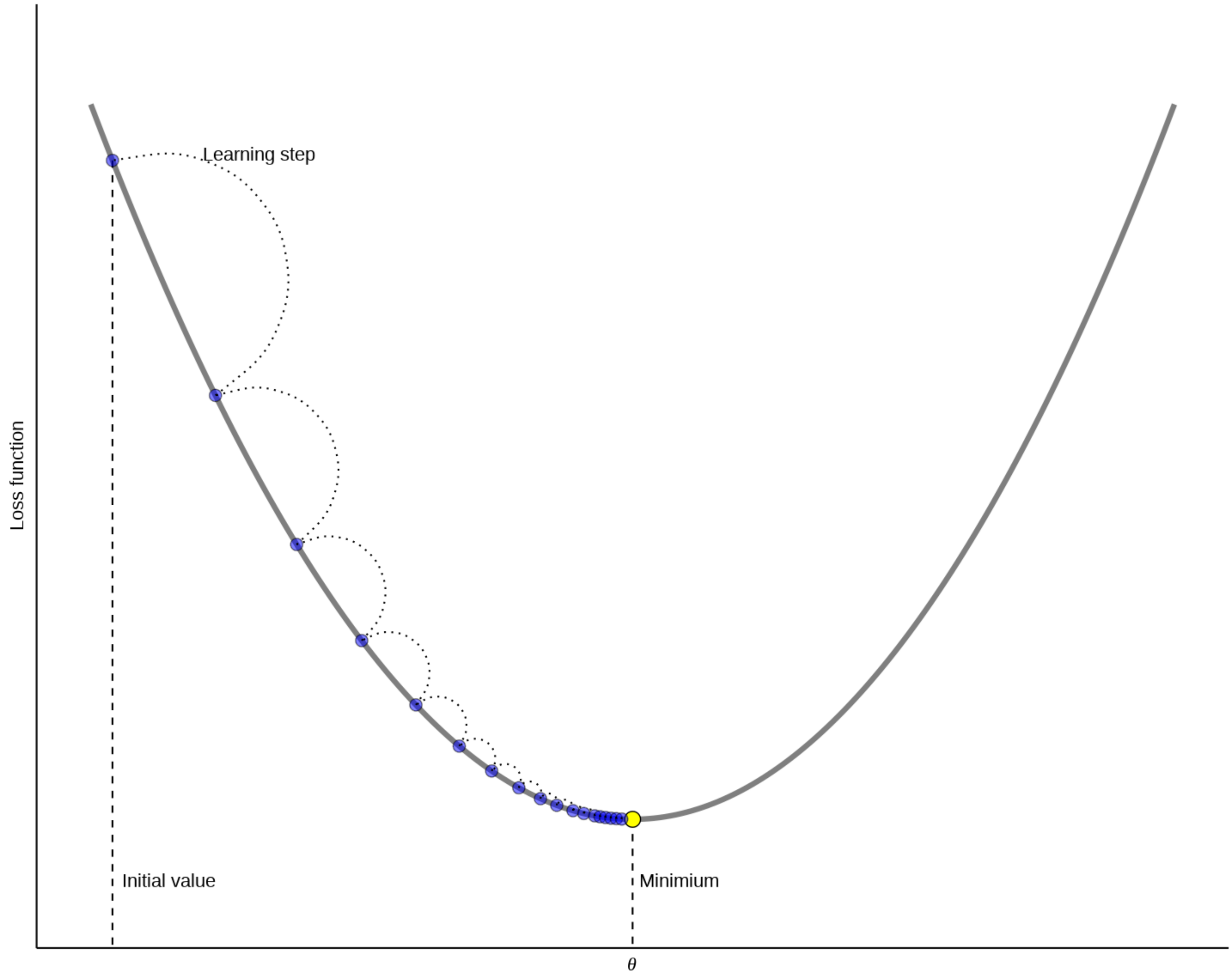
Gradient boosting machine comes from the fact that this procedure can be generalized to loss functions other than SSE.

Gradient boosting is considered a **gradient descent** algorithm.

Idea is to tweak parameter(s) iteratively to minimize a cost function.

Gradient descent measures the local gradient of the loss (cost) function for a given set of parameters Θ and takes steps in the direction of the descending gradient.

Once the gradient is zero, we have reached a minimum.


Learning step
Loss function
Initial value
Minimium
θ

Gradient descent

Gradient descent can be performed on any loss function that is differentiable.

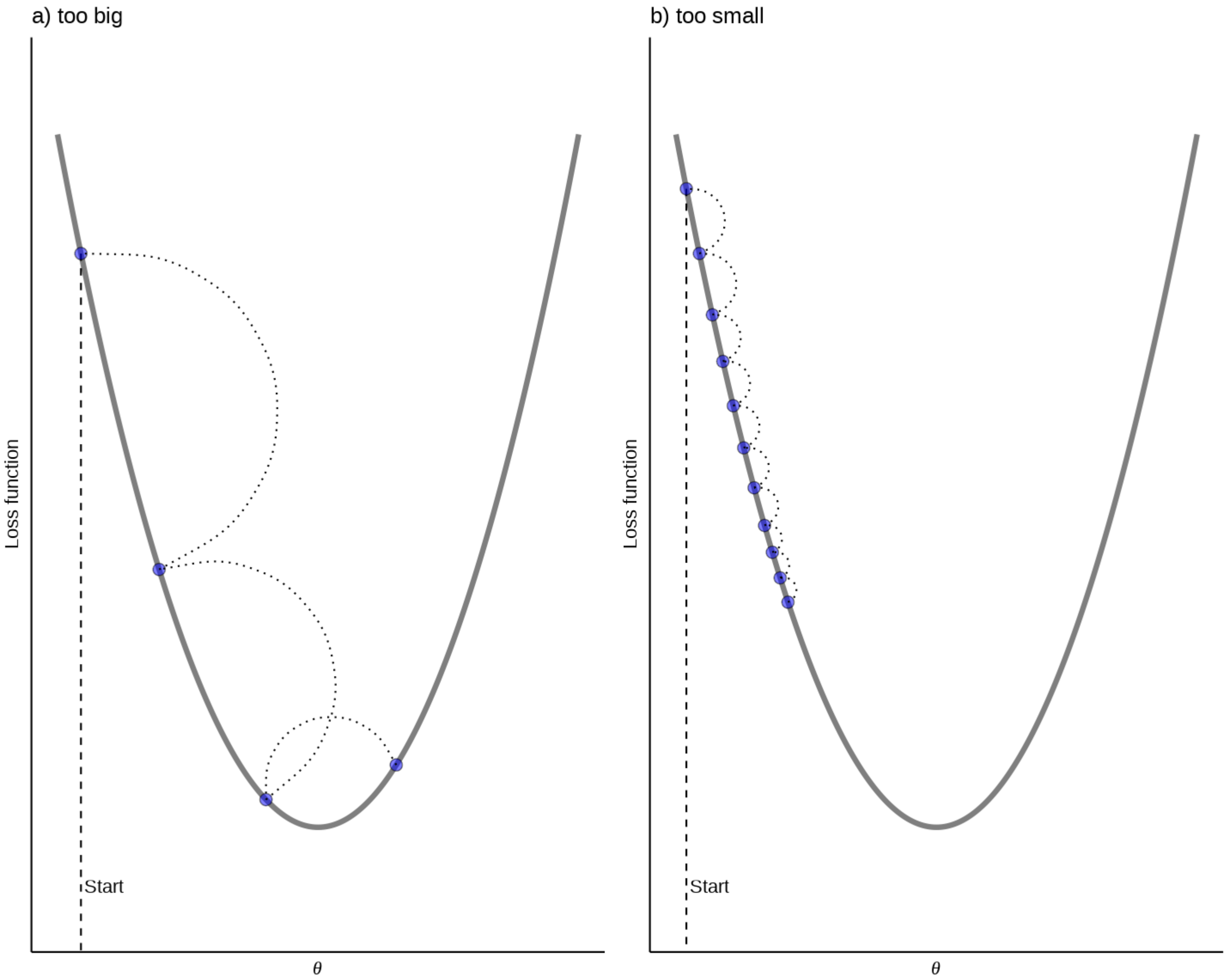
Allows GBMs to optimize different loss functions as desired.

Important parameter in gradient descent is the size of the steps.

This is controlled by the **learning rate**.

If the learning rate is too small, then the algorithm will take many iterations (steps) to find the minimum.

On the other hand, if the learning rate is too high, you might jump across the minimum and end up further away than when you started.


a) too big
Loss function
Start
θ
b) too small
Loss function
Start
θ

Gradient descent

Not all cost functions are convex (i.e., bowl shaped).

There may be local minimas, plateaus, and other irregular terrain of the loss function that makes finding the global minimum difficult.

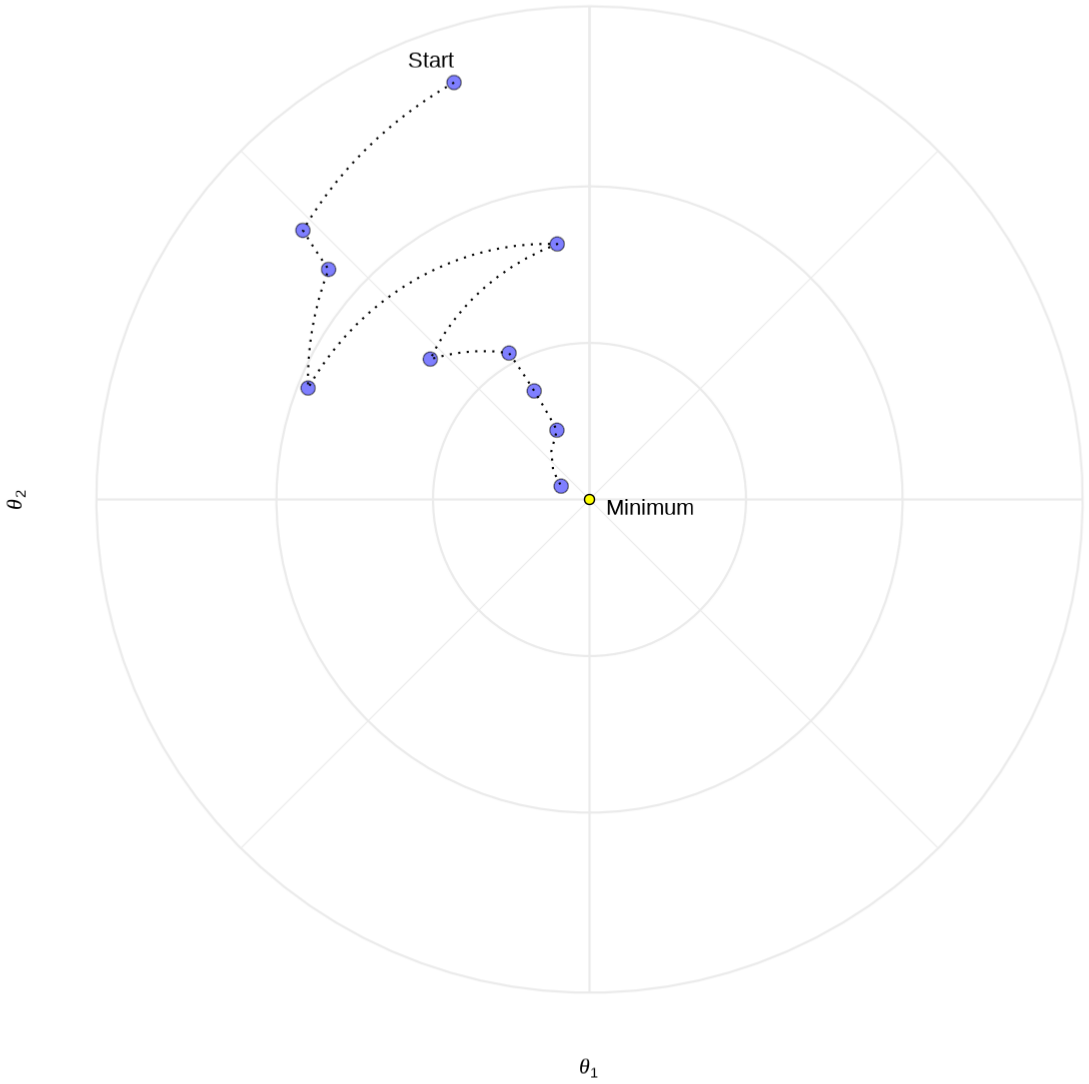
Stochastic gradient descent can help us address this problem.

Sample fraction of training observations and grow next tree using the subsample.

Several hyperparameter tuning options in stochastic gradient boosting.

- Some control gradient descent, while others control tree growing process.

If properly tuned (e.g., with k-fold CV) GBMs can lead to some of the most flexible and accurate predictive models you can build!


Start
Minimum
θ_2
θ_1

Basic GBM

Simple GBM model contains two categories of hyperparameters: *boosting hyperparameters* and *tree-specific hyperparameters*.

Two main boosting hyperparameters are:

1. Number of trees
2. Learning rate (shrinkage)

Two main tree hyperparameters are:

1. Tree depth
2. Minimum number of observations in terminal nodes

Ames housing example (GBM)

```r
# run a basic GBM model
set.seed(123)  # for reproducibility
ames_gbm1 <- gbm(
  formula = Sale_Price ~ .,
  data = ames_train,
  distribution = "gaussian",  # SSE loss function
  n.trees = 5000,
  shrinkage = 0.1,
  interaction.depth = 3,
  n.minobsinnode = 10,
  cv.folds = 10
)

# find index for number trees with minimum CV error
best <- which.min(ames_gbm1$cv.error)

# get MSE and compute RMSE
sqrt(ames_gbm1$cv.error[best])
```

General tuning strategy

Tuning can require much more strategy than a random forest model. A good approach is:

1. Choose a relatively high learning rate. Generally the default value of 0.1 works.
2. Determine the optimum number of trees for this learning rate.
3. Fix tree hyperparameters. Tune learning rate and assess speed vs. performance.
4. Tune tree-specific parameters for decided learning rate.
5. Lower the learning rate to assess for any improvements in accuracy.
6. Use final hyperparameter settings and increase CV procedures to get more robust estimates.

This process is attempted in the textbook, see for reference.

Stochastic GBMs

Few variants of stochastic gradient boosting

All of these variants have additional hyperparameters.

- Subsample rows before creating each tree
- Subsample columns before creating each tree
- Subsample columns before considering each split in each tree

When adding in a stochastic procedure, you can either include it in step 4 or step 6 of our general tuning strategy from before.

XGBoost

xgboost provides the traditional boosting and tree-based hyperparameters.

However, **xgboost** also provides additional hyperparameters that can help reduce the chances of overfitting, leading to less prediction variability and, therefore, improved accuracy.

See sections on *regularisation* and *dropout* with **xgboost**.

Our tuning strategy with **xgboost** is the following.

1. Increase number of trees and tune learning rate with early stopping
2. Tune tree-specific hyperparameters
3. Explore stochastic GBM attributes
4. If substantial overfitting occurs explore regularization hyperparameters
5. If you find hyperparameter values that are substantially different from default settings, be sure to retune the learning rate
6. Obtain final “optimal” model

XGBoost

xgboost requires some additional data preparation.

Need to encode our categorical variables numerically.

```
library(recipes)
xgb_prep <- recipe(Sale_Price ~ ., data = ames_train) %>%
  step_integer(all_nominal()) %>%
  prep(training = ames_train, retain = TRUE) %>%
  juice()

X <- as.matrix(xgb_prep[setdiff(names(xgb_prep), "Sale_Price")])
Y <- xgb_prep$Sale_Price
```

Next we will go through a series of grid searches to find model hyperparameters.

XGBoost

```
set.seed(123)
ames_xgb <- xgb.cv(
  data = X,
  label = Y,
  nrounds = 6000,
  objective = "reg:linear",
  early_stopping_rounds = 50,
  nfold = 10,
  params = list(
    eta = 0.1,
    max_depth = 3,
    min_child_weight = 3,
    subsample = 0.8,
    colsample_bytree = 1.0),
  verbose = 0
)

# minimum test CV RMSE
min(ames_xgb$evaluation_log$test_rmse_mean)
```

Minimum test CV RMSE is about 20488.

XGBoost

Next, we assess if overfitting is limiting our model’s performance by performing a grid search that examines various regularisation parameters.

```r
# hyperparameter grid
hyper_grid <- expand.grid(
  eta = 0.01,
  max_depth = 3,
  min_child_weight = 3,
  subsample = 0.5,
  colsample_bytree = 0.5,
  gamma = c(0, 1, 10, 100, 1000),
  lambda = c(0, 1e-2, 0.1, 1, 100, 1000, 10000),
  alpha = c(0, 1e-2, 0.1, 1, 100, 1000, 10000),
  rmse = 0,          # a place to dump RMSE results
  trees = 0          # a place to dump required number of trees
)
```

```r
# grid search
for(i in seq_len(nrow(hyper_grid))) {
  set.seed(123)
  m <- xgb.cv(
    data = X,
    label = Y,
    nrounds = 4000,
    objective = "reg:linear",
    early_stopping_rounds = 50,
    nfold = 10,
    verbose = 0,
    params = list(
      eta = hyper_grid$eta[i],
      max_depth = hyper_grid$max_depth[i],
      min_child_weight = hyper_grid$min_child_weight[i],
      subsample = hyper_grid$subsample[i],
      colsample_bytree = hyper_grid$colsample_bytree[i],
      gamma = hyper_grid$gamma[i],
      lambda = hyper_grid$lambda[i],
      alpha = hyper_grid$alpha[i]
    )
  )
  hyper_grid$rmse[i] <- min(m$evaluation_log$test_rmse_mean)
  hyper_grid$trees[i] <- m$best_iteration
}
```

XGBoost final model

```r
# optimal parameter list
params <- list(
  eta = 0.01,
  max_depth = 3,
  min_child_weight = 3,
  subsample = 0.5,
  colsample_bytree = 0.5
)

# train final model
xgb.fit.final <- xgboost(
  params = params,
  data = X,
  label = Y,
  nrounds = 3944,
  objective = "reg:linear",
  verbose = 0
)
```