Machine learning

Random forests and gradient-boosted trees

July 14th, 2021

Decision trees review

Decision trees partition training data into **homogenous nodes / subgroups** with similar response values

Pros

- Decision trees are **very easy to explain** to non-statisticians.
- Easy to visualize and thus easy to interpret **without assuming a parametric form**

Cons

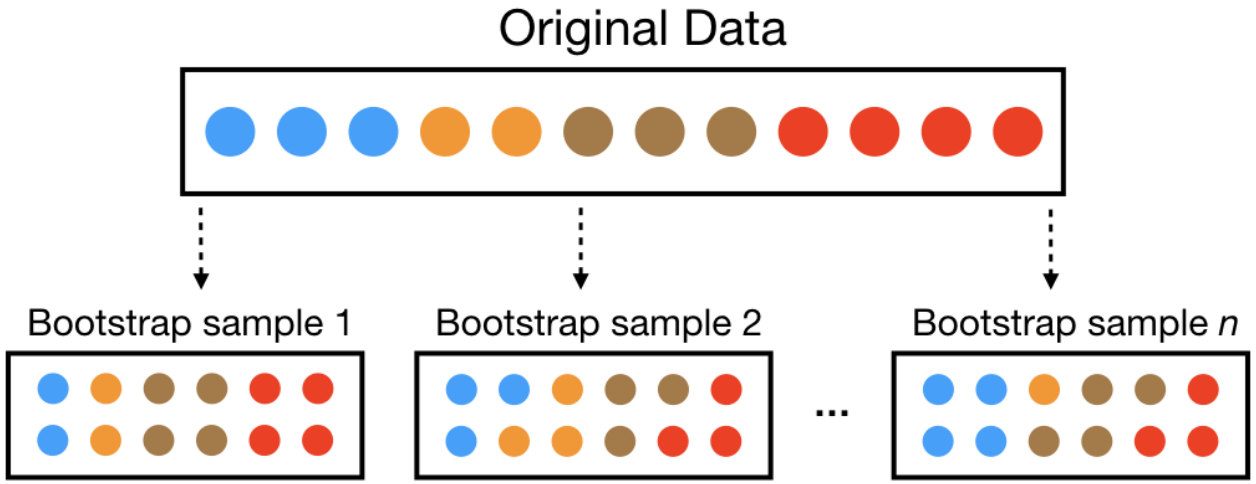
- High variance, i.e. split a dataset in half and grow tress in each half, the result will be very different
- Related note - **they generalize poorly resulting in higher test set error rates**

But there are several ways we can overcome this via **ensemble models**

Bagging

Bootstrap aggregation (aka bagging) is a general approach for overcoming high variance

- **Bootstrap**: sample the training data *with replacement*



- **Aggregation**: Combine the results from many trees together, each constructed with a different bootstrapped sample of the data

Bagging algorithm

Start with a **specified number of trees** B:

- For each tree b in $1, \ldots, B$:
    - Construct a bootstrap sample from the training data
    - Grow a deep, unpruned, complicated (aka really overfit!) tree

To generate a prediction for a new point:

- **Regression**: take the **average** across the B trees
- **Classification**: take the **majority vote** across the B trees
    - assuming each tree predicts a single class (could use probabilities instead...)

Improves prediction accuracy via **wisdom of the crowds** - but at the expense of interpretability

- Easy to read one tree, but how do you read $B = 500$?

But we can still use the measures of **variable importance** and **partial dependence** to summarize our models

Random forests algorithm

Random forests are **an extension of bagging**

- For each tree b in $1, \ldots, B$:
  - Construct a bootstrap sample from the training data
  - Grow a deep, unpruned, complicated (aka really overfit!) tree **but with a twist**
  - **At each split**: limit the variables considered to a **random subset** m_{try} of original p variables

Predictions are made the same way as bagging:

- **Regression**: take the **average** across the B trees
- **Classification**: take the **majority vote** across the B trees

Split-variable randomization adds more randomness to make **each tree more independent of each other**

Introduce m_{try} as a tuning parameter: typically use $p/3$ (regression) or $\sqrt{p}$ (classification)

- $m_{try} = p$ is bagging

Example data: MLB 2021 batting statistics

Downloaded MLB 2021 batting statistics leaderboard from Fangraphs

```
library(tidyverse)
mlb_data <- read_csv("http://www.stat.cmu.edu/cmsac/sure/2021/materials/data/fg_batting_2021.csv"
  janitor::clean_names() %>%
  mutate_at(vars(bb_percent:k_percent), parse_number)
model_mlb_data <- mlb_data %>%
  dplyr::select(-name, -team, -playerid)
head(model_mlb_data)
```

```
## # A tibble: 6 × 20
##       g    pa    hr     r   rbi    sb bb_percent k_per…¹   iso babip   avg   obp
##   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>      <dbl>   <dbl> <dbl> <dbl> <dbl> <dbl>
## 1    82   354    27    66    69     2       14.4    17.2 0.336 0.346 0.336 0.438
## 2    68   288    27    66    58    18       12.5    28.1 0.395 0.333 0.302 0.385
## 3    79   347    16    61    52     0       13.5    17   0.231 0.324 0.298 0.398
## 4    82   372    21    63    54    10        8.9    23.9 0.256 0.329 0.286 0.349
## 5    78   342    23    67    51    16       13.2    24.3 0.313 0.306 0.278 0.386
## 6    82   322    31    60    67    12       11.2    28   0.418 0.29  0.277 0.363
## # … with 8 more variables: slg <dbl>, w_oba <dbl>, xw_oba <dbl>, w_rc <dbl>,
## #   bs_r <dbl>, off <dbl>, def <dbl>, war <dbl>, and abbreviated variable name
## #   ¹k_percent
```

Example using `ranger`

`ranger` package is a popular / fast implementation (see `randomForest` for the original)

```
library(ranger)
init_mlb_rf <- ranger(war ~ ., data = model_mlb_data, num.trees = 50, importance = "impurity")
init_mlb_rf
```

```
## Ranger result
##
## Call:
##  ranger(war ~ ., data = model_mlb_data, num.trees = 50, importance = "impurity")
##
## Type:                             Regression
## Number of trees:                  50
## Sample size:                      135
## Number of independent variables:  19
## Mtry:                             4
## Target node size:                 5
## Variable importance mode:         impurity
## Splitrule:                        variance
## OOB prediction error (MSE):       0.1667404
## R squared (OOB):                  0.8536123
```

Out-of-bag estimate

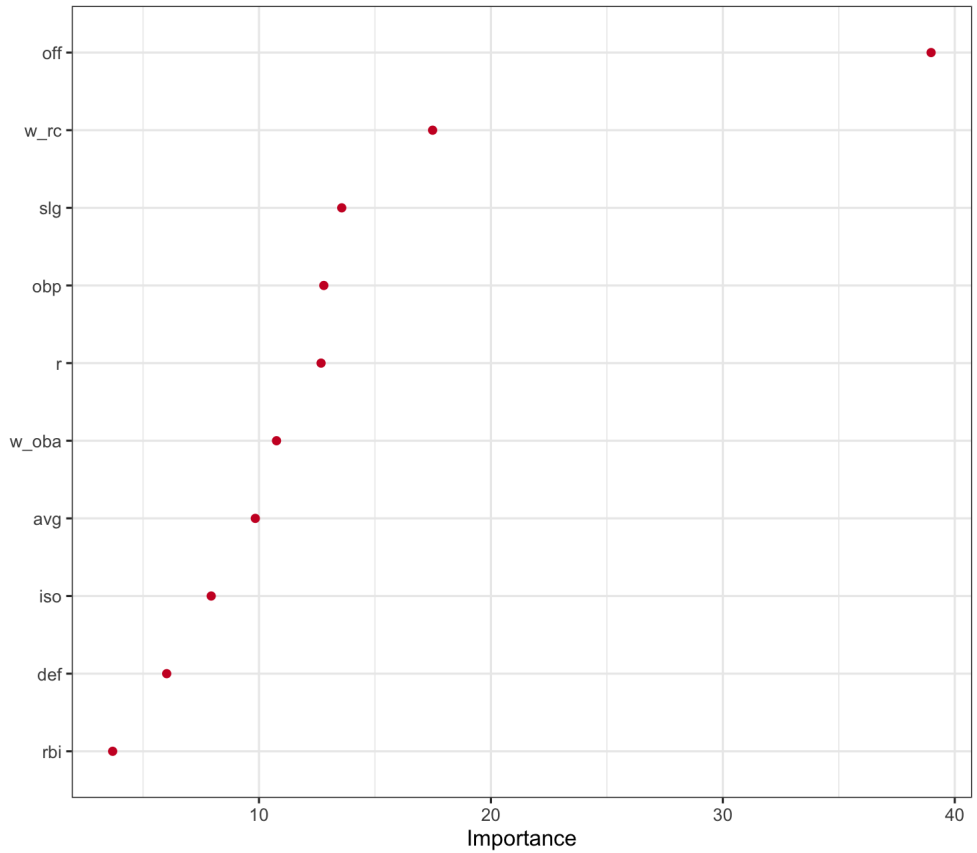
Since the trees are constructed via bootstrapped data (samples with replacements) - each sample *is likely to have duplicate observations / rows*

Out-of-bag (OOB) - original observations not contained in a single bootstrap sample

- Can use the OOB samples to estimate predictive performance (OOB becomes better with larger datasets)
- On average $\approx 63\%$ of original data ends up in any particular bootstrap sample

Variable importance

```
library(vip)
vip(init_mlb_rf, geom = "point") + theme_bw()
```



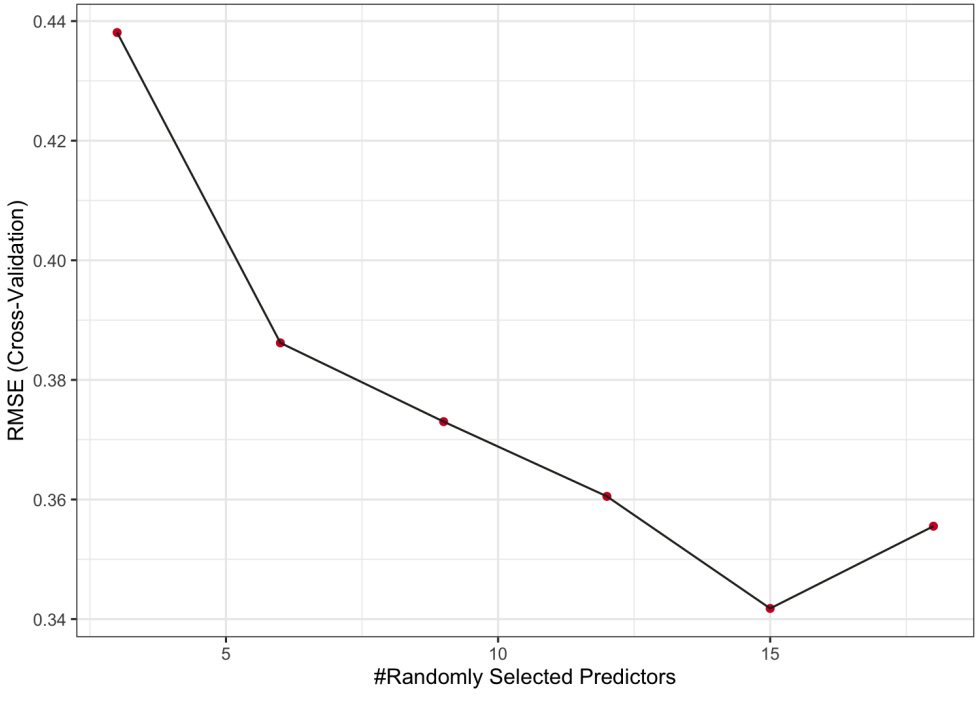
Tuning random forests

Unfortunately `caret` does not let you know tune number of trees - typically the error goes down with more (*Exercise: check out CV performance as a function of the number trees on your own, compare with OOB error*)

- **Important**: m_{try}
- Marginal: tree complexity, splitting rule, sampling scheme

```
library(caret)
rf_tune_grid <-
  expand.grid(mtry = seq(3, 18, by = 3),
              splitrule = "variance",
              min.node.size = 5)
set.seed(1917)
caret_mlb_rf <-
  train(war ~ ., data = model_mlb_data,
        method = "ranger", num.trees = 50,
        trControl = trainControl(method = "cv
        tuneGrid = rf_tune_grid)
```

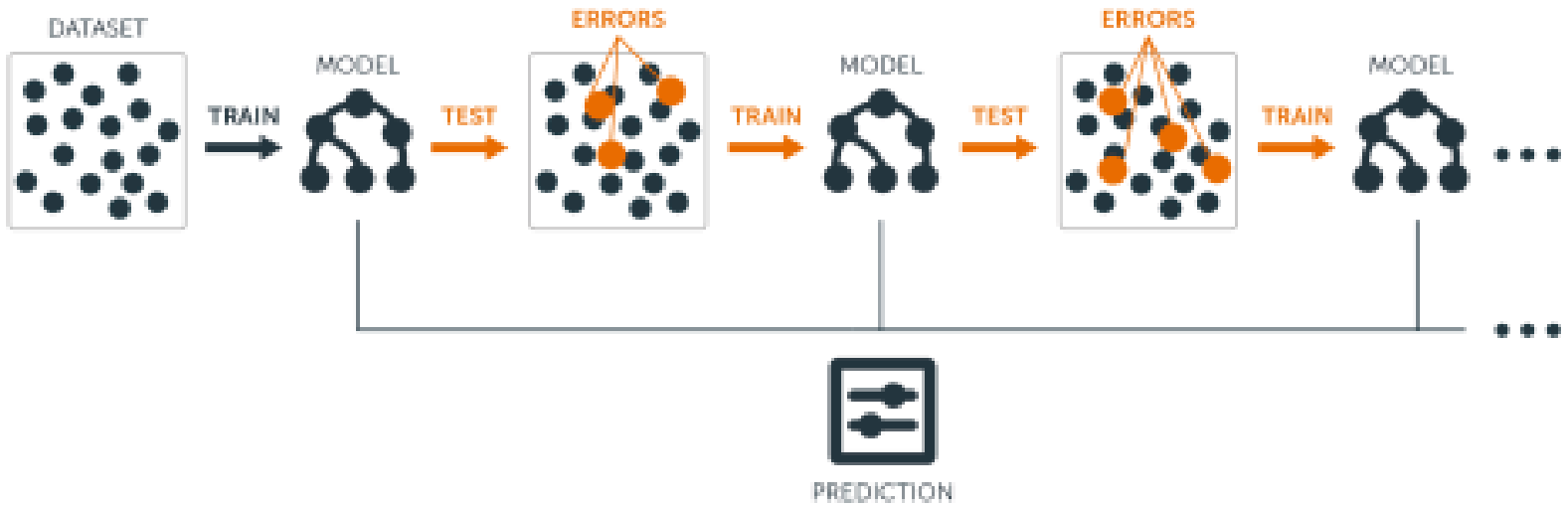
```
ggplot(caret_mlb_rf) + theme_bw()
```



Boosting

Build ensemble models **sequentially**

- start with a **weak learner**, e.g. small decision tree with few splits
- each model in the sequence *slightly* improves upon the predictions of the previous models **by focusing on the observations with the largest errors / residuals**



Boosted trees algorithm

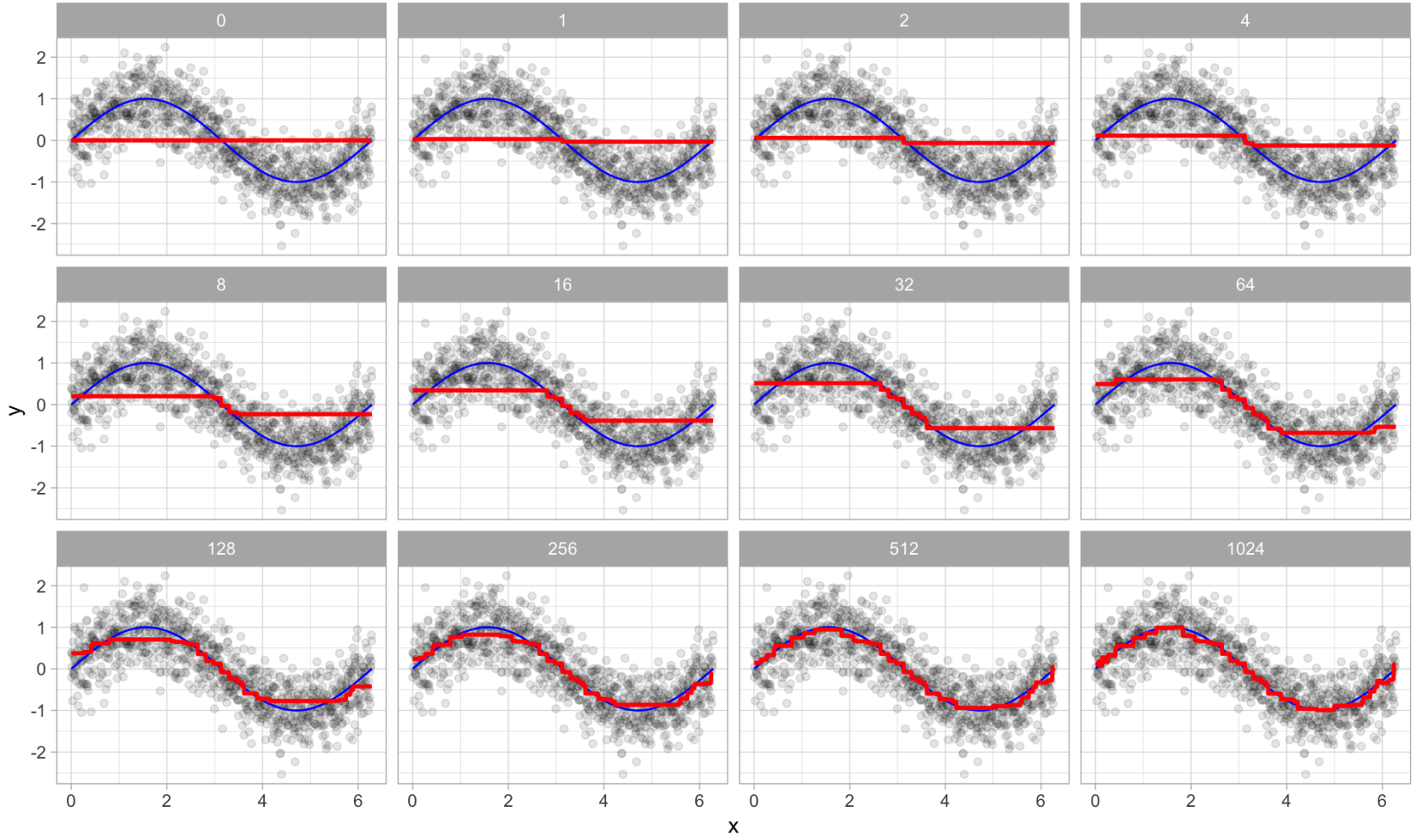
Write the prediction at step t of the search as $\hat{y}_i^{(t)}$, start with $\hat{y}_i^{(0)} = 0$

- Fit the first decision tree f_1 to the data: $\hat{y}_i^{(1)} = f_1(x_i) = \hat{y}_i^{(0)} + f_1(x_i)$
- Fit the next tree f_2 to the residuals of the previous: $y_i - \hat{y}_i^{(1)}$
- Add this to the prediction: $\hat{y}_i^{(2)} = \hat{y}_i^{(1)} + f_2(x_i) = f_1(x_i) + f_2(x_i)$
- Fit the next tree f_3 to the residuals of the previous: $y_i - \hat{y}_i^{(2)}$
- Add this to the prediction: $\hat{y}_i^{(3)} = \hat{y}_i^{(2)} + f_3(x_i) = f_1(x_i) + f_2(x_i) + f_3(x_i)$

Continue until some stopping criteria to reach final model as a **sum of trees**:

$$\hat{y}_i = f(x_i) = \sum_{b=1}^{B} f_b(x_i)$$

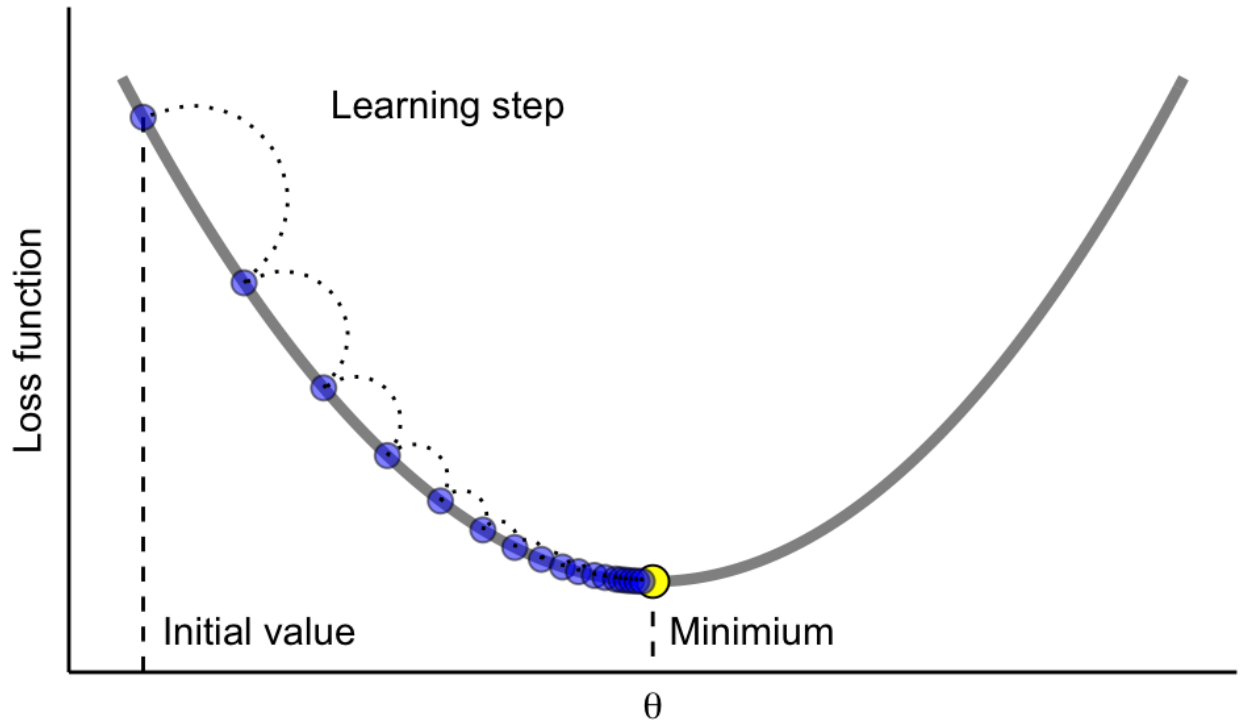
Visual example of boosting in action



Gradient boosted trees

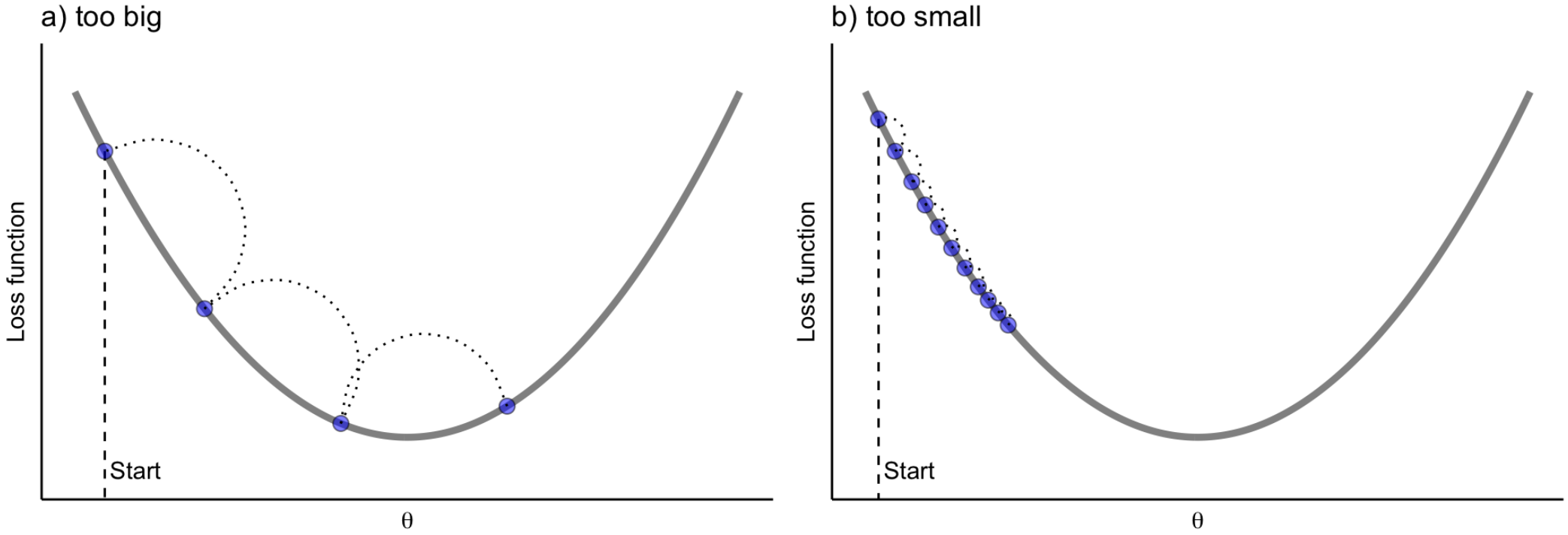
Regression boosting algorithm can be generalized to other loss functions via **gradient descent** - leading to gradient boosted trees, aka **gradient boosting machines (GBMs)**

Update the model parameters in the direction of the loss function's descending gradient



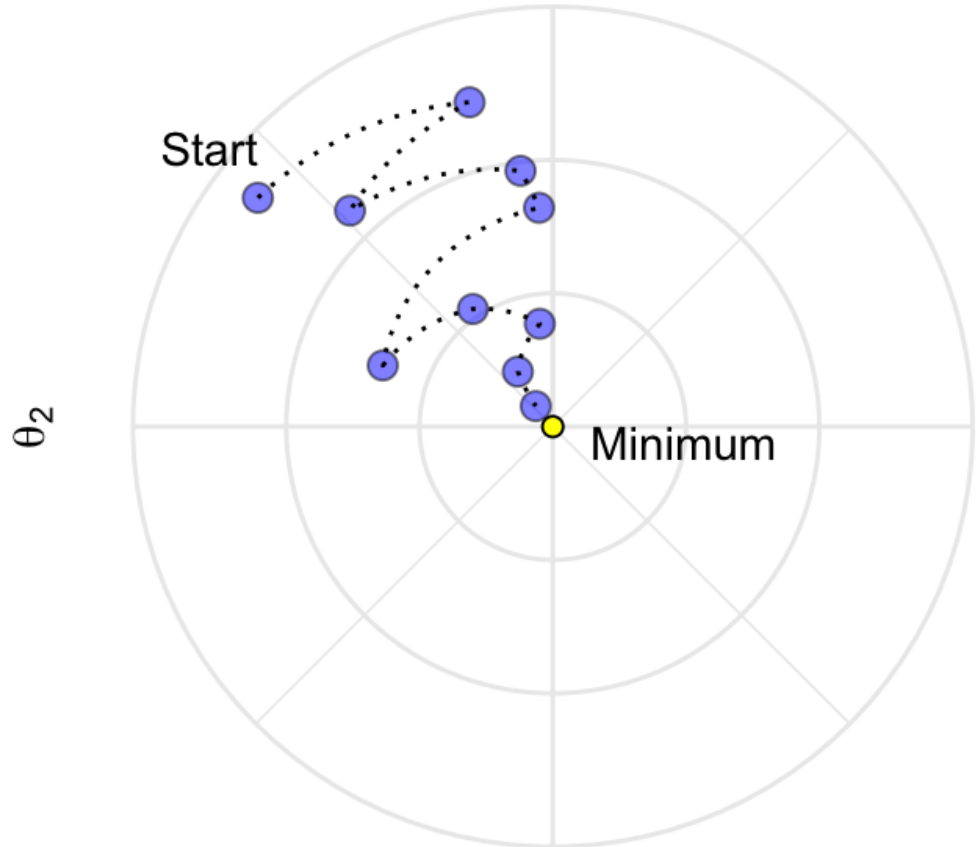
Tune the learning rate in gradient descent

We need to control how much we update by in each step - **the learning rate**



Stochastic gradient descent can help with complex loss functions

Can take random samples of the data when updating - makes algorithm faster and adds randomness to get closer to global minimum (no guarantees!)



eXtreme gradient boosting with XGBoost

Tuning GBMs with `xgboost`

XGBoost (extreme gradient boosting) is a very powerful, efficient boosting library that is available to use within `R` via the `xgboost` package

What we have to consider tuning (our **hyperparameters**):

- number of trees B (`nrounds`)
- learning rate (`eta`), i.e. how much we update in each step
- these two really have to be tuned together
- complexity of the trees (depth, number of observations in nodes)
- XGBoost also provides more **regularization** (via `gamma`) and early stopping

More work to tune properly as compared to random forests

- But GBMs have more flexibility in their usage for particular objective functions
- *Insert with great power comes great responsibility meme*

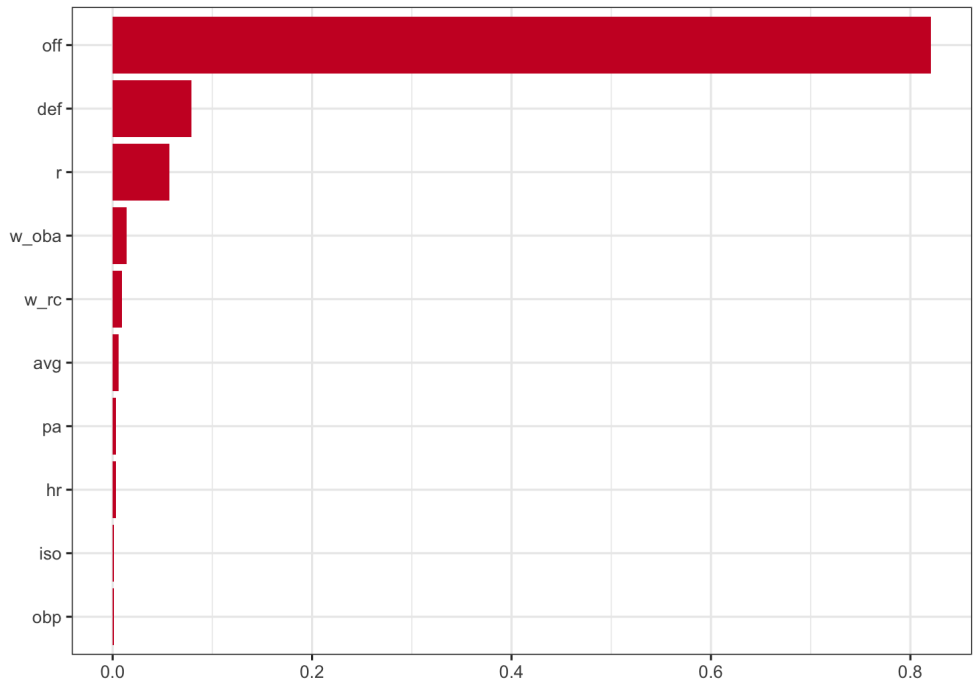
XGBoost example

```
library(xgboost)
xgboost_tune_grid <- expand.grid(nrounds = seq(from = 20, to = 200, by = 20),
                                 eta = c(0.025, 0.05, 0.1, 0.3), gamma = 0,
                                 max_depth = c(1, 2, 3, 4), colsample_bytree = 1,
                                 min_child_weight = 1, subsample = 1)
xgboost_tune_control <- trainControl(method = "cv", number = 5, verboseIter = FALSE)
set.seed(1937)
xgb_tune <- train(x = as.matrix(dplyr::select(model_mlb_data, -war)),
                  y = model_mlb_data$war, trControl = xgboost_tune_control,
                  tuneGrid = xgboost_tune_grid,
                  objective = "reg:squarederror", method = "xgbTree",
                  verbose = TRUE)
```

```
## [20:50:12] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` insteac
## [20:50:12] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` insteac
## [20:50:12] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` insteac
## [20:50:12] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` insteac
## [20:50:12] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` insteac
## [20:50:12] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` insteac
## [20:50:12] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` insteac
## [20:50:12] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` insteac
## [20:50:12] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` insteac
## [20:50:12] WARNING: src/c_api/c_api.cc:935: `ntree_limit` is deprecated, use `iteration_range` insteac
```

XGBoost example

```
xgb_fit_final <- xgboost(data = as.matrix(dplyr::select(model_mlb_data, -war)),
                         label = model_mlb_data$war, objective = "reg:squarederror",
                         nrounds = xgb_tune$bestTune$nrounds,
                         params = as.list(dplyr::select(xgb_tune$bestTune,
                                                        -nrounds)),
                         verbose = 0)
vip(xgb_fit_final) + theme_bw()
```



XGBoost example

```
library(pdp)
partial(xgb_fit_final, pred.var = "off", train = as.matrix(dplyr::select(model_mlb_data, -war)),
        plot.engine = "ggplot2", plot = TRUE,
        type = "regression") + theme_bw()
```