Gradient Boosting

Data Sets

Attrition

```
attrition <- attrition %>% mutate_if(is.ordered, factor, order = F)
attrition_h2o <- as.h2o(attrition)

churn <- initial_split(attrition, prop = .7, strata = "Attrition")

churn_train <- training(churn)
churn_test <- testing(churn)

rm(churn)
```

Ames, Iowa housing data.

```
set.seed(123)

ames <- AmesHousing::make_ames()
ames_h2o <- as.h2o(ames)

ames_split <- initial_split(ames, prop =.7, strata = "Sale_Price")

ames_train <- training(ames_split)
ames_test <- testing(ames_split)

rm(ames_split)

h2o.init(max_mem_size = "10g", strict_version_check = F)
```

```
 Connection successful!

R is connected to the H2O cluster:
    H2O cluster uptime:         1 minutes 20 seconds
    H2O cluster timezone:       America/New_York
    H2O data parsing timezone:  UTC
    H2O cluster version:        3.28.0.2
    H2O cluster version age:    10 days
    H2O cluster name:           H2O_started_from_R_brandon_fkm502
    H2O cluster total nodes:    1
    H2O cluster total memory:   15.71 GB
    H2O cluster total cores:    16
    H2O cluster allowed cores:  16
    H2O cluster healthy:        TRUE
```

```
    H2O Connection ip:          localhost
    H2O Connection port:        54321
    H2O Connection proxy:       NA
    H2O Internal Security:      FALSE
    H2O API Extensions:         Amazon S3, XGBoost, Algos, AutoML, Core V3, TargetEncoder, Core
    R Version:                  R version 3.6.2 (2019-12-12)
```

```
train_h2o <- as.h2o(ames_train)
response <- "Sale_Price"
predictors <- setdiff(colnames(ames_train), response)
```

Gradient Boosting Overview

Whereas random forests build an ensemble of deep independent trees, GBMs build an ensemble of shallow trees in sequence with each tree learning and improving on the previous one. Although shallow trees by themselves are rather weak predictive models, they can be "boosted" to produce a powerful "committee"

Boosting vs Bagging

Boosting is a general concept that aggregates the predictions of simpler models. It is typically used with models that have high bias and low variance. This works particularly well with decision trees.

Boosting is a sequential algorithm that attacks the bias-variance trade-off by starting with a *weak* model, and sequentially boost its performance by continuiing to build new trees, where each new tree in the sequence tries to "fix up" where the previous one made the biggest error.

Base Learners

Boosting is a framework that iteratively improves any weak learning model. Many gradient boosting applications allow you to "plug in" various classes of weak learners at your disposal.

In practice, these are almost always decision trees as base learners.

Traning Weak Models

A weak model is one whose error rate is only slighly better than random guessing. The idea behind boosting is that eadch model in the sequence slighly improves upon the performance of the previous one.

Sequential Training

Boosted trees are grown sequentially; each tree is grown using information from previously grown trees to improve performance.

Basic GBM Algorithm

- 1.) Fit a decision tree to the data: $F_1(x) = y$
- 2.) We then fit the next decision tree to the residuals of the previous: $h_1(x) = y - F_1(x)$
- 3.) Add this new tree to our algorithm: $F_2(x) = F_1(x) + h_1(x)$
- 4.) Fit the next decision tree to the residuals of F_2 : $h_2(x) = y - F_2(x)$
- 5.) Add this new tree to our algorithm: $F_3(x) = F_2(x) + h_1(x)$
- 6.) Continue this process until some mechanism (i.e., cross validation) tells us to stop.

The final model here is a stagewise additive model of b individual trees:

$f(x) = \sum_{b=1}^{B} f^b(x)$

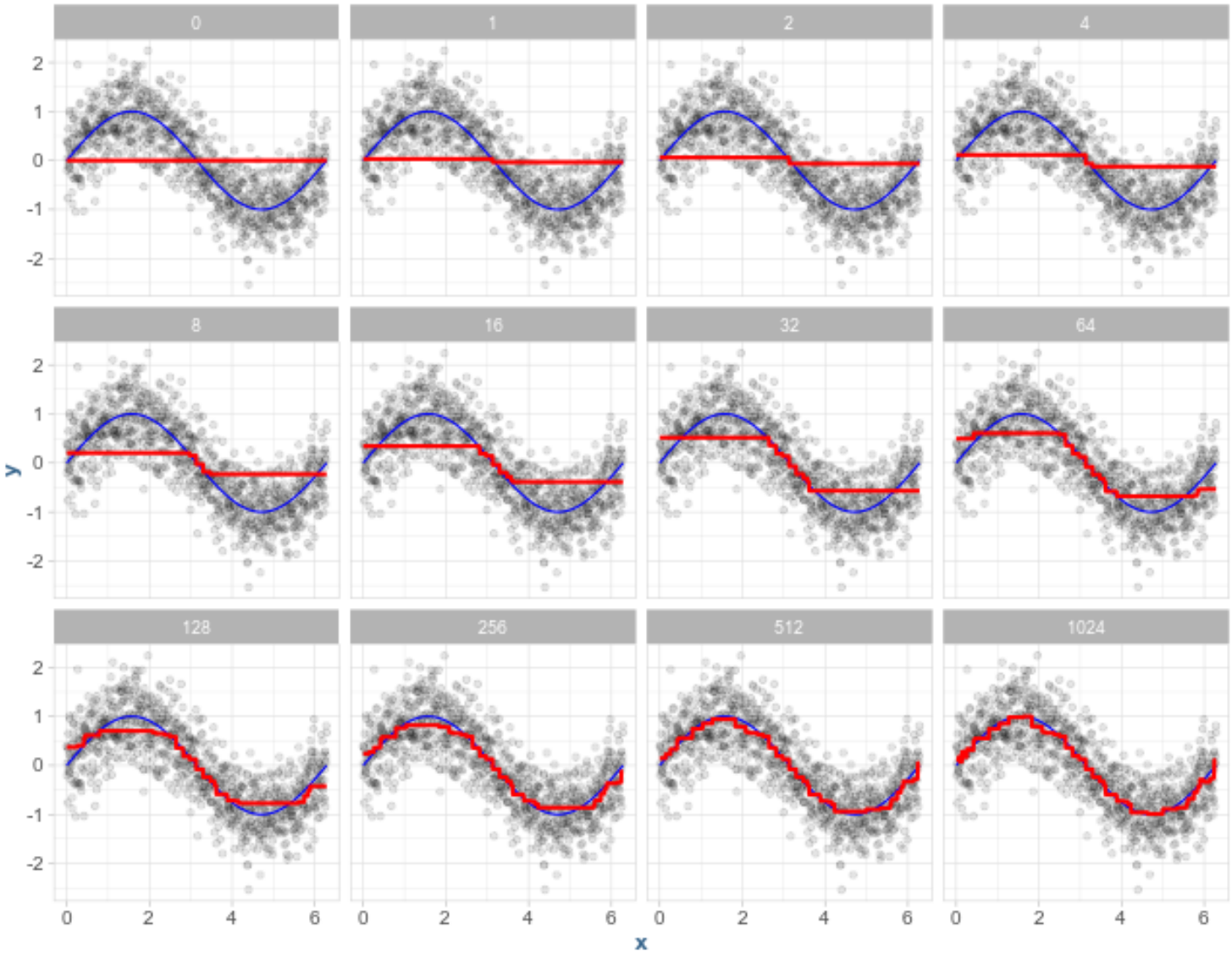
Visually, the process looks like this:

```
# Simulate sine wave data
set.seed(1112)  # for reproducibility
df <- tibble::tibble(
  x = seq(from = 0, to = 2 * pi, length = 1000),
  y = sin(x) + rnorm(length(x), sd = 0.5),
  truth = sin(x)
)

# Function to boost `rpart::rpart()` trees
rpartBoost <- function(x, y, data, num_trees = 100, learn_rate = 0.1, tree_depth = 6) {
  x <- data[[deparse(substitute(x))]]
  y <- data[[deparse(substitute(y))]]
  G_b_hat <- matrix(0, nrow = length(y), ncol = num_trees + 1)
  r <- y
  for(tree in seq_len(num_trees)) {
    g_b_tilde <- rpart(r ~ x, control = list(cp = 0, maxdepth = tree_depth))
    g_b_hat <- learn_rate * predict(g_b_tilde)
    G_b_hat[, tree + 1] <- G_b_hat[, tree] + matrix(g_b_hat)
    r <- r - g_b_hat
    colnames(G_b_hat) <- paste0("tree_", c(0, seq_len(num_trees)))
  }
  cbind(df, as.data.frame(G_b_hat)) %>%
    gather(tree, prediction, starts_with("tree")) %>%
    mutate(tree = stringr::str_extract(tree, "\\d+") %>% as.numeric())
}

# Plot boosted tree sequence
rpartBoost(x, y, data = df, num_trees = 2^10, learn_rate = 0.05, tree_depth = 1) %>%
  filter(tree %in% c(0, 2^c(0:10))) %>%
```

```
  ggplot(aes(x, prediction)) +
    ylab("y") +
    geom_point(data = df, aes(x, y), alpha = .1) +
    geom_line(data = df, aes(x, truth), color = "blue") +
    geom_line(colour = "red", size = 1) +
    facet_wrap(~ tree, nrow = 3)
```

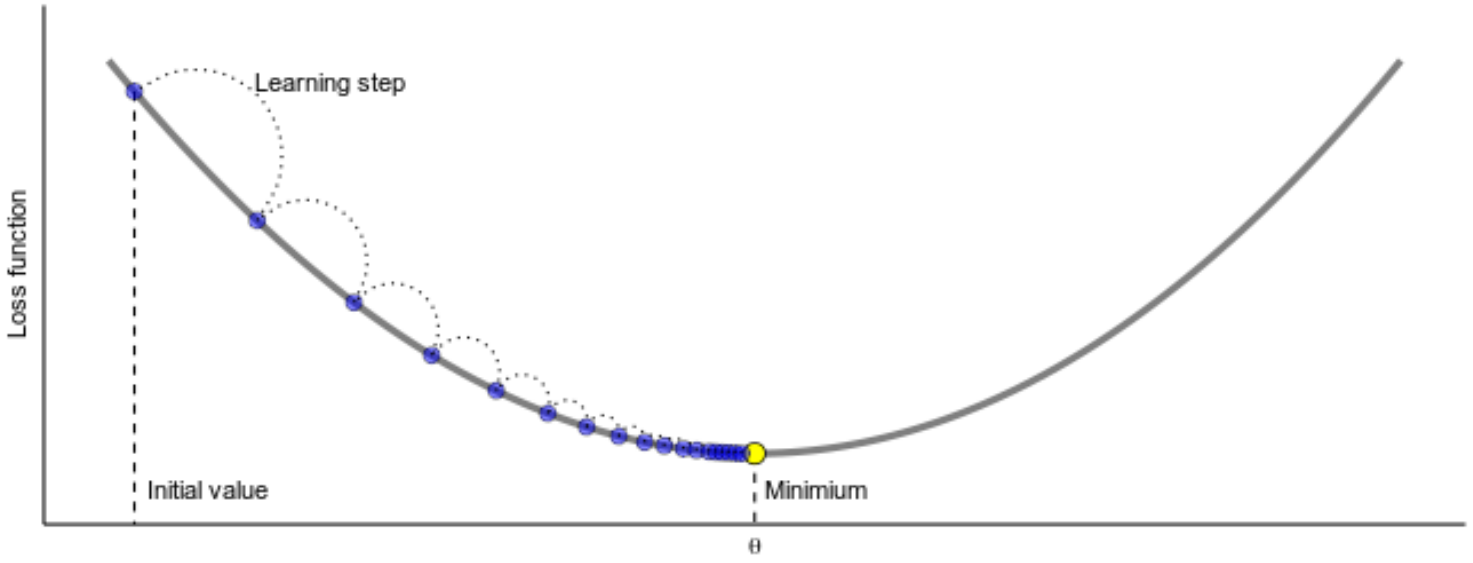


The Gradient in GBM is a gradient descent algorithm (from Calculus), where we can use it to find the minima/maximua of a function. Here, we apply the gradient descent to the loss function in the algorithm.

```
# create data to plot
x <- seq(-5, 5, by = .05)
y <- x^2 + 3
df <- data.frame(x, y)
```

```
step <- 5
step_size <- .2
for(i in seq_len(18)) {
  next_step <- max(step) + round(diff(range(max(step), which.min(df$y))) * step_size, 0)
  step <- c(step, next_step)
  next
}
steps <- df[step, ] %>%
  mutate(x2 = lag(x), y2 = lag(y)) %>%
  dplyr::slice(1:18)
# plot
ggplot(df, aes(x, y)) +
  geom_line(size = 1.5, alpha = .5) +
  theme_classic() +
  scale_y_continuous("Loss function", limits = c(0, 30)) +
  xlab(expression(theta)) +
  geom_segment(data = df[c(5, which.min(df$y)), ], aes(x = x, y = y, xend = x, yend = -Inf), l
  geom_point(data = filter(df, y == min(y)), aes(x, y), size = 4, shape = 21, fill = "yellow")
  geom_point(data = steps, aes(x, y), size = 3, shape = 21, fill = "blue", alpha = .5) +
  geom_curve(data = steps, aes(x = x, y = y, xend = x2, yend = y2), curvature = 1, lty = "dotte
  theme(
    axis.ticks = element_blank(),
    axis.text = element_blank()
  ) +
  annotate("text", x = df[5, "x"], y = 1, label = "Initial value", hjust = -0.1, vjust = .8) +
  annotate("text", x = df[which.min(df$y), "x"], y = 1, label = "Minimium", hjust = -0.1, vjust
  annotate("text", x = df[5, "x"], y = df[5, "y"], label = "Learning step", hjust = -.8, vjust
```

```
Warning: Removed 1 rows containing missing values (geom_curve).
```



Gradient descent is the process of gradually decreasing the cost function (i.e, MSE) by tweaking parameter(s) iteratively until you have reached a minimuim.

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

An important parameter is the step size with is controlled by the learning rate. If the learning rate is too small, then the algorithm will take many interations to find the minimum. If its too big, it can stip over a possible minimum value.

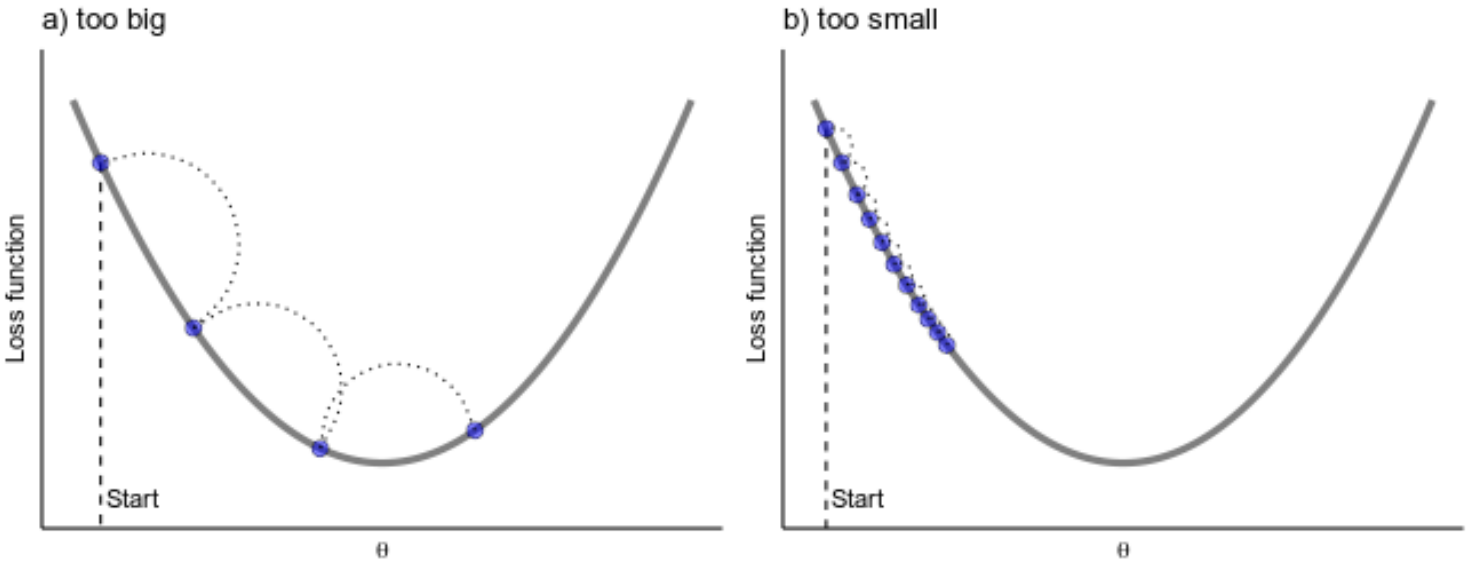
Visually:

```
# create too small of a learning rate
step <- 5
step_size <- .05
for(i in seq_len(10)) {
  next_step <- max(step) + round(diff(range(max(step), which.min(df$y))) * step_size, 0)
  step <- c(step, next_step)
  next
}
too_small <- df[step, ] %>%
  mutate(x2 = lag(x), y2 = lag(y))
# plot
p1 <- ggplot(df, aes(x, y)) +
  geom_line(size = 1.5, alpha = .5) +
  theme_classic() +
  scale_y_continuous("Loss function", limits = c(0, 30)) +
  xlab(expression(theta)) +
  geom_segment(data = too_small[1, ], aes(x = x, y = y, xend = x, yend = -Inf), lty = "dashed")
  geom_point(data = too_small, aes(x, y), size = 3, shape = 21, fill = "blue", alpha = .5) +
  geom_curve(data = too_small, aes(x = x, y = y, xend = x2, yend = y2), curvature = 1, lty = "d
  theme(
    axis.ticks = element_blank(),
    axis.text = element_blank()
  ) +
  annotate("text", x = df[5, "x"], y = 1, label = "Start", hjust = -0.1, vjust = .8) +
  ggtitle("b) too small")
# create too large of a learning rate
too_large <- df[round(which.min(df$y) * (1 + c(-.9, -.6, -.2, .3)), 0), ] %>%
  mutate(x2 = lag(x), y2 = lag(y))
# plot
p2 <- ggplot(df, aes(x, y)) +
  geom_line(size = 1.5, alpha = .5) +
  theme_classic() +
  scale_y_continuous("Loss function", limits = c(0, 30)) +
  xlab(expression(theta)) +
  geom_segment(data = too_large[1, ], aes(x = x, y = y, xend = x, yend = -Inf), lty = "dashed")
  geom_point(data = too_large, aes(x, y), size = 3, shape = 21, fill = "blue", alpha = .5) +
  geom_curve(data = too_large, aes(x = x, y = y, xend = x2, yend = y2), curvature = 1, lty = "d
```

```
  theme(
    axis.ticks = element_blank(),
    axis.text = element_blank()
  ) +
  annotate("text", x = too_large[1, "x"], y = 1, label = "Start", hjust = -0.1, vjust = .8) +
  ggtitle("a) too big")
gridExtra::grid.arrange(p2, p1, nrow = 1)
```

```
Warning: Removed 1 rows containing missing values (geom_curve).
```

```
Warning: Removed 1 rows containing missing values (geom_curve).
```



Not all loss functions are convex, however. There are many local minimas, plateaus, and other irregular terain of the loss function that makes finding the global minimum difficult.

Stochastic gradient descent can help us address this problem by sampling a fraction of the training observations (typically w/o replacement), and growing the next tree using that subsample.

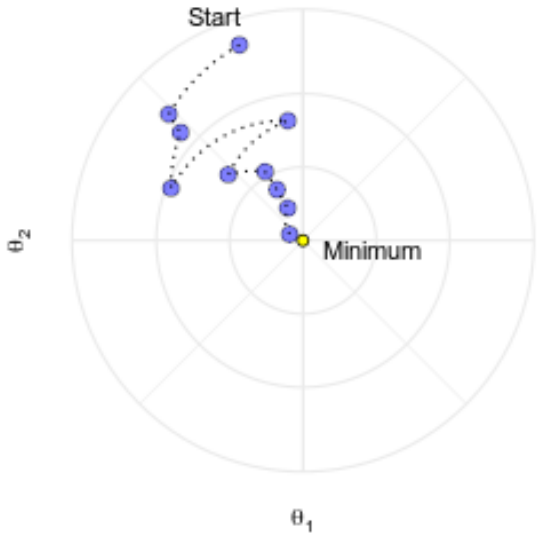
Visually:

```
# create random walk data
set.seed(123)
x <- sample(seq(3, 5, by = .05), 10, replace = TRUE)
set.seed(123)
y <- seq(2, 28, length.out = 10)

random_walk <- data.frame(
  x = x,
  y = y[order(y, decreasing = TRUE)]
)
```

```
optimal <- data.frame(x = 0, y = 0)

# plot
ggplot(df, aes(x, y)) +
  coord_polar() +
  theme_minimal() +
  theme(
    axis.ticks = element_blank(),
    axis.text = element_blank()
  ) +
  xlab(expression(theta[1])) +
  ylab(expression(theta[2])) +
  geom_point(data = random_walk, aes(x, y), size = 3, shape = 21, fill = "blue", alpha = .5) +
  geom_point(data = optimal, aes(x, y), size = 2, shape = 21, fill = "yellow") +
  geom_path(data = random_walk, aes(x, y), lty = "dotted") +
  annotate("text", x = random_walk[1, "x"], y = random_walk[1, "y"], label = "Start", hjust = 1
  annotate("text", x = optimal[1, "x"], y = optimal[1, "y"], label = "Minimum", hjust = -.2, vj
  ylim(c(0, 28)) +
  xlim(-5, 5)
```



Stochastic gradent descent will often find a near-optimal solution by jumping out of local minimas and off plateaus.

Basic GBM

The first boosting algorithm was AdaBoost, which was then generalized into a regression framework.

Hyperparameters

Typically there are two kinds of hyperparamters in GBM, boosting and tree-specific.

- 1.) Number of Trees

Total number of trees in the ensemble.

- 2.) Learning Rate

Determines the contribution of each tree on the final outcome and control how quickly the algorithm proceeds down the gradient descent.

Implementation:

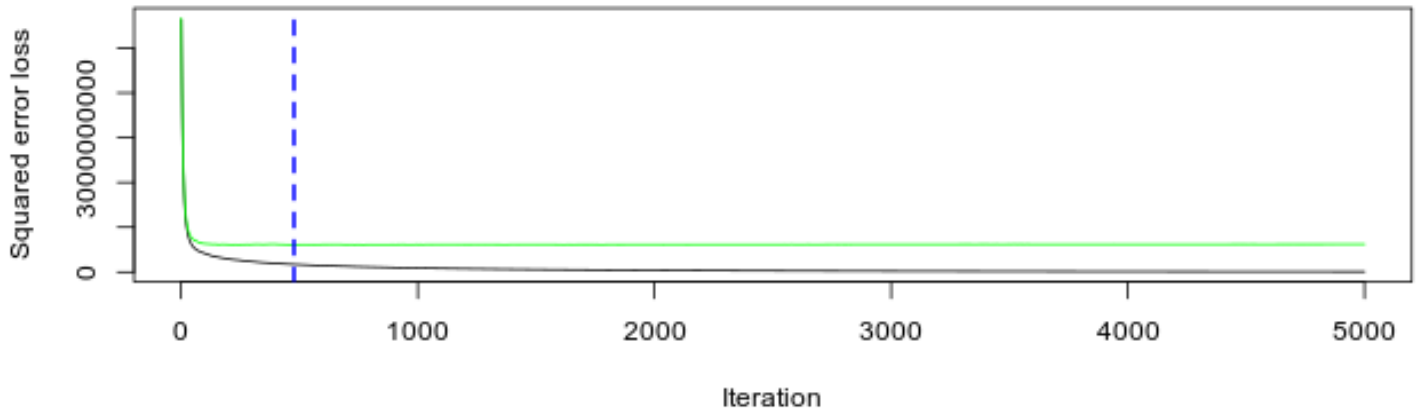
```
# run a basic GBM model
set.seed(123)  # for reproducibility
ames_gbm1 <- gbm(
  formula = Sale_Price ~ .,
  data = ames_train,
  distribution = "gaussian",  # SSE loss function
  n.trees = 5000,
  shrinkage = 0.1,
  interaction.depth = 3,
  n.minobsinnode = 10,
  cv.folds = 10
)

# 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])
```

```
[1] 24586.21
```

```
# plot error curve
gbm.perf(ames_gbm1, method = "cv")
```




```
[1] 477
```

General Tuning Strategy

```
# create grid search
hyper_grid <- expand.grid(
  learning_rate = c(0.3, 0.1, 0.05, 0.01, 0.005),
  RMSE = NA,
  trees = NA,
  time = NA
)

# execute grid search
for(i in seq_len(nrow(hyper_grid))) {

  # fit gbm
  set.seed(123)  # for reproducibility
  train_time <- system.time({
    m <- gbm(
      formula = Sale_Price ~ .,
      data = ames_train,
      distribution = "gaussian",
      n.trees = 5000,
      shrinkage = hyper_grid$learning_rate[i],
      interaction.depth = 3,
      n.minobsinnode = 10,
      cv.folds = 10
   )
```

```
  })

  # add SSE, trees, and training time to results
  hyper_grid$RMSE[i]  <- sqrt(min(m$cv.error))
  hyper_grid$trees[i] <- which.min(m$cv.error)
  hyper_grid$Time[i]  <- train_time[["elapsed"]]

}

# results
arrange(hyper_grid, RMSE)
```

```
  learning_rate     RMSE trees time   Time
1         0.050 24047.81  2627   NA 46.792
2         0.010 24363.16  4525   NA 45.239
3         0.005 24438.93  4552   NA 48.358
4         0.100 24586.21   477   NA 46.647
5         0.300 26797.92   543   NA 46.035
```

Exaustive Method

```
# search grid
hyper_grid <- expand.grid(
  n.trees = 6000,
  shrinkage = 0.01,
  interaction.depth = c(3, 5, 7),
  n.minobsinnode = c(5, 10, 15)
)

# create model fit function
model_fit <- function(n.trees, shrinkage, interaction.depth, n.minobsinnode) {
  set.seed(123)
  m <- gbm(
    formula = Sale_Price ~ .,
    data = ames_train,
    distribution = "gaussian",
    n.trees = n.trees,
    shrinkage = shrinkage,
    interaction.depth = interaction.depth,
    n.minobsinnode = n.minobsinnode,
    cv.folds = 10
  )
  # compute RMSE
  sqrt(min(m$cv.error))
}
```

```
# perform search grid with functional programming
hyper_grid$rmse <- purrr::pmap_dbl(
  hyper_grid,
  ~ model_fit(
    n.trees = ..1,
    shrinkage = ..2,
    interaction.depth = ..3,
    n.minobsinnode = ..4
    )
)

# results
arrange(hyper_grid, rmse)
```

```
  n.trees shrinkage interaction.depth n.minobsinnode     rmse
1    6000      0.01                 7              5 23520.33
2    6000      0.01                 7             15 23724.03
3    6000      0.01                 5             15 23770.08
4    6000      0.01                 5              5 23850.41
5    6000      0.01                 7             10 23855.54
6    6000      0.01                 5             10 23874.26
7    6000      0.01                 3             15 24173.12
8    6000      0.01                 3              5 24178.92
9    6000      0.01                 3             10 24346.85
```

Stochastic GBMs

```
# refined hyperparameter grid
hyper_grid <- list(
  sample_rate = c(0.5, 0.75, 1),              # row subsampling
  col_sample_rate = c(0.5, 0.75, 1),          # col subsampling for each split
  col_sample_rate_per_tree = c(0.5, 0.75, 1)  # col subsampling for each tree
)

# random grid search strategy
search_criteria <- list(
  strategy = "RandomDiscrete",
  stopping_metric = "mse",
  stopping_tolerance = 0.001,
  stopping_rounds = 10,
  max_runtime_secs = 60*60
)
```

```
# perform grid search
grid <- h2o.grid(
  algorithm = "gbm",
  grid_id = "gbm_grid",
  x = predictors,
  y = response,
  training_frame = train_h2o,
  hyper_params = hyper_grid,
  ntrees = 6000,
  learn_rate = 0.01,
  max_depth = 7,
  min_rows = 5,
  nfolds = 10,
  stopping_rounds = 10,
  stopping_tolerance = 0,
  search_criteria = search_criteria,
  seed = 123
)

# collect the results and sort by our model performance metric of choice
grid_perf <- h2o.getGrid(
  grid_id = "gbm_grid",
  sort_by = "mse",
  decreasing = FALSE
)

grid_perf
```

```
H2O Grid Details
================

Grid ID: gbm_grid
Used hyper parameters:
  -  col_sample_rate
  -  col_sample_rate_per_tree
  -  sample_rate
Number of models: 27
Number of failed models: 0

Hyper-Parameter Search Summary: ordered by increasing mse
  col_sample_rate col_sample_rate_per_tree sample_rate         model_ids
1             0.5                       0.5         0.5 gbm_grid_model_26
2             1.0                       0.5         0.5 gbm_grid_model_15
3            0.75                       0.5         0.5 gbm_grid_model_23
```

```
4              0.5                       0.5        0.75 gbm_grid_model_14
5              0.5                      0.75         0.5  gbm_grid_model_2
                   mse
1 5.0407401685574996E8
2  5.080430332075919E8
3  5.158362981815745E8
4 5.2119479263464874E8
5  5.225352739929353E8

---
   col_sample_rate col_sample_rate_per_tree sample_rate         model_ids
22            0.75                       1.0        0.75 gbm_grid_model_10
23            0.75                      0.75         1.0  gbm_grid_model_8
24             1.0                       1.0        0.75 gbm_grid_model_17
25             1.0                      0.75         1.0 gbm_grid_model_25
26            0.75                       1.0         1.0  gbm_grid_model_4
27             1.0                       1.0         1.0  gbm_grid_model_6
                   mse
22 5.842831556947927E8
23 5.860399896049552E8
24 6.073509333529812E8
25 6.175457748070993E8
26 6.266430997495539E8
27 6.658372458693743E8
```

```r
# Grab the model_id for the top model, chosen by cross validation error
best_model_id <- grid_perf@model_ids[[1]]
best_model <- h2o.getModel(best_model_id)

# Now let's get performance metrics on the best model
h2o.performance(model = best_model, xval = TRUE)
```

```
H2ORegressionMetrics: gbm
** Reported on cross-validation data. **
** 10-fold cross-validation on training data (Metrics computed for combined holdout predictions

MSE:  504074017
RMSE:  22451.59
MAE:  13455.52
RMSLE:  0.118753
Mean Residual Deviance :  504074017
R^2 :  0.9217076
```

XGBoost

Regularization

XGBoost has multiple tuning parameters to help prevent overfitting.

γ = Lagrangian multiplier which controls the complexity of a given tree.

α = L_1 regularization (simmilar to ridge regresssion)

λ = L_2 regularization (similar to lasso)

Dropout

Loosly defined as a regularization technique:

```
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
```

```
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)
```

```
[1] 22940.82
```

```
# 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
)

# 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
}

# results
hyper_grid %>%
  filter(rmse > 0) %>%
  arrange(rmse) %>%
```

```
  glimpse()
```

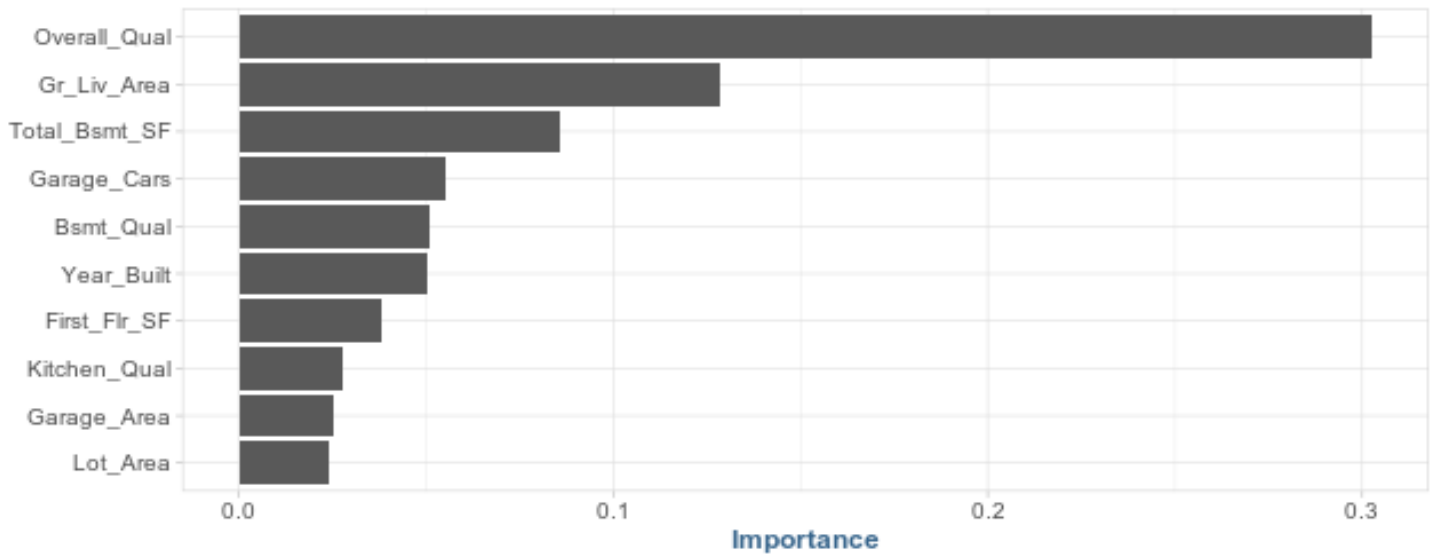
```
Observations: 245
Variables: 10
$ eta              <dbl> 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0....
$ max_depth        <dbl> 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,...
$ min_child_weight <dbl> 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,...
$ subsample        <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, ...
$ colsample_bytree <dbl> 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, 0.5, ...
$ gamma            <dbl> 100, 10, 1000, 1, 0, 0, 1000, 10, 100, 1, 1000, 1,...
$ lambda           <dbl> 1.0, 1.0, 1.0, 1.0, 1.0, 0.1, 0.1, 0.1, 0.1, 0.1, ...
$ alpha            <dbl> 10000.0, 10000.0, 10000.0, 10000.0, 10000.0, 10000...
$ rmse             <dbl> 22937.12, 22937.12, 22937.12, 22937.12, 22937.12, ...
$ trees            <dbl> 3909, 3909, 3909, 3909, 3909, 3996, 3996, 3996, 39...
```

```
# 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
)
```

Feature Interpretation

```
# variable importance plot
vip::vip(xgb.fit.final)
```




```
h2o.shutdown(prompt = FALSE)
```

```
# clean up
rm(list = ls())
```