**How Gradient Boosting works**

Let’s look at how Gradient Boosting works. Most of the magic is described in the name: “Gradient” plus “Boosting”.

**Boosting** builds models from individual so called “weak learners” in an iterative way. In the [Random Forests](https://shirinsplayground.netlify.com/2018/10/ml_basics_rf/) part, I had already discussed the differences between **Bagging** and **Boosting** as tree ensemble methods. In boosting, the individual models are not built on completely random subsets of data and features but sequentially by putting more weight on instances with wrong predictions and high errors. The general idea behind this is that instances, which are hard to predict correctly (“difficult” cases) will be focused on during learning, so that the model learns from past mistakes. When we train each ensemble on a subset of the training set, we also call this **Stochastic Gradient Boosting**, which can help improve generalizability of our model.

The **gradient** is used to minimize a **loss function**, similar to how [Neural Nets](https://shirinsplayground.netlify.com/2018/11/neural_nets_explained/) utilize gradient descent to optimize (“learn”) weights. In each round of training, the weak learner is built and its predictions are compared to the correct outcome that we expect. The distance between prediction and truth represents the error rate of our model. These errors can now be used to calculate the gradient. The gradient is nothing fancy, it is basically the partial derivative of our loss function – so it describes the steepness of our error function. The gradient can be used to find the direction in which to change the model parameters in order to (maximally) reduce the error in the next round of training by “descending the gradient”.

In Neural nets, gradient descent is used to look for the minimum of the loss function, i.e. learning the model parameters (e.g. weights) for which the prediction error is lowest in **a single model**. In Gradient Boosting we are combining the predictions of **multiple models**, so we are not optimizing the model parameters directly but the boosted model predictions. Therefore, the gradients will be added to the running training process by fitting the next tree also to these values.

Because we apply gradient descent, we will find **learning rate** (the “step size” with which we descend the gradient), **shrinkage** (reduction of the learning rate) and **loss function** as hyperparameters in Gradient Boosting models – just as with Neural Nets. Other [hyperparameters](https://xgboost.readthedocs.io/en/latest/parameter.html#general-parameters) of Gradient Boosting are similar to those of Random Forests:

* the number of iterations (i.e. the number of trees to ensemble),
* the number of observations in each leaf,
* tree complexity and depth,
* the proportion of samples and
* the proportion of features on which to train on.

**Gradient Boosting Machines vs. XGBoost**

[XGBoost](https://github.com/dmlc/xgboost) stands for Extreme Gradient Boosting; it is a specific implementation of the Gradient Boosting method which uses more accurate approximations to find the best tree model. It employs a number of nifty tricks that make it exceptionally successful, particularly with structured data. The most important are

1.) computing **second-order gradients, i.e. second partial derivatives** of the loss function (similar to **Newton’s method**), which provides more information about the direction of gradients and how to get to the minimum of our loss function. While regular gradient boosting uses the loss function of our base model (e.g. decision tree) as a proxy for minimizing the error of the overall model, XGBoost uses the 2nd order derivative as an approximation.

2.) And advanced **regularization** (L1 & L2), which improves model generalization.

XGBoost has additional advantages: training is very fast and can be parallelized / distributed across clusters.

**Code in R**

Here is a very quick run through how to train Gradient Boosting and XGBoost models in R with caret, xgboost and h2o.

**Data**

First, data: I’ll be using the ISLR package, which contains a number of datasets, one of them is College.

Statistics for a large number of US Colleges from the 1995 issue of US News and World Report.

library(tidyverse)

library(ISLR)

ml\_data <- College

ml\_data %>%

glimpse()

## Observations: 777

## Variables: 18

## $ Private Yes, Yes, Yes, Yes, Yes, Yes, Yes, Yes, Yes, Yes, ...

## $ Apps 1660, 2186, 1428, 417, 193, 587, 353, 1899, 1038, ...

## $ Accept 1232, 1924, 1097, 349, 146, 479, 340, 1720, 839, 4...

## $ Enroll 721, 512, 336, 137, 55, 158, 103, 489, 227, 172, 4...

## $ Top10perc 23, 16, 22, 60, 16, 38, 17, 37, 30, 21, 37, 44, 38...

## $ Top25perc 52, 29, 50, 89, 44, 62, 45, 68, 63, 44, 75, 77, 64...

## $ F.Undergrad 2885, 2683, 1036, 510, 249, 678, 416, 1594, 973, 7...

## $ P.Undergrad 537, 1227, 99, 63, 869, 41, 230, 32, 306, 78, 110,...

## $ Outstate 7440, 12280, 11250, 12960, 7560, 13500, 13290, 138...

## $ Room.Board 3300, 6450, 3750, 5450, 4120, 3335, 5720, 4826, 44...

## $ Books 450, 750, 400, 450, 800, 500, 500, 450, 300, 660, ...

## $ Personal 2200, 1500, 1165, 875, 1500, 675, 1500, 850, 500, ...

## $ PhD 70, 29, 53, 92, 76, 67, 90, 89, 79, 40, 82, 73, 60...

## $ Terminal 78, 30, 66, 97, 72, 73, 93, 100, 84, 41, 88, 91, 8...

## $ S.F.Ratio 18.1, 12.2, 12.9, 7.7, 11.9, 9.4, 11.5, 13.7, 11.3...

## $ perc.alumni 12, 16, 30, 37, 2, 11, 26, 37, 23, 15, 31, 41, 21,...

## $ Expend 7041, 10527, 8735, 19016, 10922, 9727, 8861, 11487...

## $ Grad.Rate 60, 56, 54, 59, 15, 55, 63, 73, 80, 52, 73, 76, 74...

**Gradient Boosting in caret**

The most flexible R package for machine learning is caret. If you go to the [Available Models section in the online documentation](https://topepo.github.io/caret/available-models.html) and search for “Gradient Boosting”, this is what you’ll find:

| **Model** | **method Value** | **Type** | **Libraries** | **Tuning Parameters** |
| --- | --- | --- | --- | --- |
| eXtreme Gradient Boosting | xgbDART | Classification, Regression | xgboost, plyr | nrounds, max\_depth, eta, gamma, subsample, colsample\_bytree, rate\_drop, skip\_drop, min\_child\_weight |
| eXtreme Gradient Boosting | xgbLinear | Classification, Regression | xgboost | nrounds, lambda, alpha, eta |
| eXtreme Gradient Boosting | xgbTree | Classification, Regression | xgboost, plyr | nrounds, max\_depth, eta, gamma, colsample\_bytree, min\_child\_weight, subsample |
| Gradient Boosting Machines | gbm\_h2o | Classification, Regression | h2o | ntrees, max\_depth, min\_rows, learn\_rate, col\_sample\_rate |
| Stochastic Gradient Boosting | gbm | Classification, Regression | gbm, plyr | n.trees, interaction.depth, shrinkage, n.minobsinnode |

A table with the different Gradient Boosting implementations, you can use with caret. Here I’ll show a very simple Stochastic Gradient Boosting example:

library(caret)

# Partition into training and test data

set.seed(42)

index <- createDataPartition(ml\_data$Private, p = 0.7, list = FALSE)

train\_data <- ml\_data[index, ]

test\_data <- ml\_data[-index, ]

# Train model with preprocessing & repeated cv

model\_gbm <- caret::train(Private ~ .,

data = train\_data,

method = "gbm",

preProcess = c("scale", "center"),

trControl = trainControl(method = "repeatedcv",

number = 5,

repeats = 3,

verboseIter = FALSE),

verbose = 0)

model\_gbm

## Stochastic Gradient Boosting

##

## 545 samples

## 17 predictor

## 2 classes: 'No', 'Yes'

##

## Pre-processing: scaled (17), centered (17)

## Resampling: Cross-Validated (5 fold, repeated 3 times)

## Summary of sample sizes: 437, 436, 435, 436, 436, 436, ...

## Resampling results across tuning parameters:

##

## interaction.depth n.trees Accuracy Kappa

## 1 50 0.9217830 0.7940197

## 1 100 0.9327980 0.8264864

## 1 150 0.9370795 0.8389860

## 2 50 0.9352501 0.8321982

## 2 100 0.9358337 0.8356107

## 2 150 0.9333816 0.8301596

## 3 50 0.9364511 0.8357210

## 3 100 0.9400927 0.8463975

## 3 150 0.9346048 0.8330068

##

## Tuning parameter 'shrinkage' was held constant at a value of 0.1

##

## Tuning parameter 'n.minobsinnode' was held constant at a value of 10

## Accuracy was used to select the optimal model using the largest value.

## The final values used for the model were n.trees = 100,

## interaction.depth = 3, shrinkage = 0.1 and n.minobsinnode = 10.

With predict(), we can use this model to make predictions on test data. Here, I’ll be feeding this directly to the confusionMatrix function:

caret::confusionMatrix(

data = predict(model\_gbm, test\_data),

reference = test\_data$Private

)

## Confusion Matrix and Statistics

##

## Reference

## Prediction No Yes

## No 57 6

## Yes 6 163

##

## Accuracy : 0.9483

## 95% CI : (0.9114, 0.973)

## No Information Rate : 0.7284

## P-Value [Acc > NIR] : <2e-16

##

## Kappa : 0.8693

## Mcnemar's Test P-Value : 1

##

## Sensitivity : 0.9048

## Specificity : 0.9645

## Pos Pred Value : 0.9048

## Neg Pred Value : 0.9645

## Prevalence : 0.2716

## Detection Rate : 0.2457

## Detection Prevalence : 0.2716

## Balanced Accuracy : 0.9346

##

## 'Positive' Class : No

##

**The xgboost library**

We can also directly work with the [xgboost](https://xgboost.readthedocs.io/en/latest/R-package/xgboostPresentation.html) package in R. It’s a bit more involved but also includes advanced possibilities.

The easiest way to work with xgboost is with the xgboost() function. The four most important arguments to give are

* data: a **matrix** of the training data
* label: the response variable in numeric format (for binary classification 0 & 1)
* objective: defines what learning task should be trained, here binary classification
* nrounds: number of boosting iterations

library(xgboost)

xgboost\_model <- xgboost(data = as.matrix(train\_data[, -1]),

label = as.numeric(train\_data$Private)-1,

max\_depth = 3,

objective = "binary:logistic",

nrounds = 10,

verbose = FALSE,

prediction = TRUE)

xgboost\_model

## ##### xgb.Booster

## raw: 6.7 Kb

## call:

## xgb.train(params = params, data = dtrain, nrounds = nrounds,

## watchlist = watchlist, verbose = verbose, print\_every\_n = print\_every\_n,

## early\_stopping\_rounds = early\_stopping\_rounds, maximize = maximize,

## save\_period = save\_period, save\_name = save\_name, xgb\_model = xgb\_model,

## callbacks = callbacks, max\_depth = 3, objective = "binary:logistic",

## prediction = TRUE)

## params (as set within xgb.train):

## max\_depth = "3", objective = "binary:logistic", prediction = "TRUE", silent = "1"

## xgb.attributes:

## niter

## callbacks:

## cb.evaluation.log()

## # of features: 17

## niter: 10

## nfeatures : 17

## evaluation\_log:

## iter train\_error

## 1 0.064220

## 2 0.051376

## ---

## 9 0.036697

## 10 0.033028

We can again use predict(); because here, we will get prediction probabilities, we need to convert them into labels to compare them with the true class:

predict(xgboost\_model,

as.matrix(test\_data[, -1])) %>%

as.tibble() %>%

mutate(prediction = round(value),

label = as.numeric(test\_data$Private)-1) %>%

count(prediction, label)

## # A tibble: 4 x 3

## prediction label n

##

## 1 0 0 56

## 2 0 1 6

## 3 1 0 7

## 4 1 1 163

Alternatively, we can use xgb.train(), which is more flexible and allows for more advanced settings compared to xgboost(). Here, we first need to create a so called DMatrix from the data. Optionally, we can define a watchlist for evaluating model performance during the training run. I am also creating a parameter set as a list object, which I am feeding to the params argument.

dtrain <- xgb.DMatrix(as.matrix(train\_data[, -1]),

label = as.numeric(train\_data$Private)-1)

dtest <- xgb.DMatrix(as.matrix(test\_data[, -1]),

label = as.numeric(test\_data$Private)-1)

params <- list(max\_depth = 3,

objective = "binary:logistic",

silent = 0)

watchlist <- list(train = dtrain, eval = dtest)

bst\_model <- xgb.train(params = params,

data = dtrain,

nrounds = 10,

watchlist = watchlist,

verbose = FALSE,

prediction = TRUE)

bst\_model

## ##### xgb.Booster

## raw: 6.7 Kb

## call:

## xgb.train(params = params, data = dtrain, nrounds = 10, watchlist = watchlist,

## verbose = FALSE, prediction = TRUE)

## params (as set within xgb.train):

## max\_depth = "3", objective = "binary:logistic", silent = "0", prediction = "TRUE", silent = "1"

## xgb.attributes:

## niter

## callbacks:

## cb.evaluation.log()

## # of features: 17

## niter: 10

## nfeatures : 17

## evaluation\_log:

## iter train\_error eval\_error

## 1 0.064220 0.099138

## 2 0.051376 0.077586

## ---

## 9 0.036697 0.060345

## 10 0.033028 0.056034

The model can be used just as before:

predict(bst\_model,

as.matrix(test\_data[, -1])) %>%

as.tibble() %>%

mutate(prediction = round(value),

label = as.numeric(test\_data$Private)-1) %>%

count(prediction, label)

## # A tibble: 4 x 3

## prediction label n

##

## 1 0 0 56

## 2 0 1 6

## 3 1 0 7

## 4 1 1 163

The third option, is to use xgb.cv, which will perform cross-validation. This function does not return a model, it is rather used to find optimal hyperparameters, particularly for nrounds.

cv\_model <- xgb.cv(params = params,

data = dtrain,

nrounds = 100,

watchlist = watchlist,

nfold = 5,

verbose = FALSE,

prediction = TRUE) # prediction of cv folds

Here, we can see after how many rounds, we achieved the smallest test error:

cv\_model$evaluation\_log %>%

filter(test\_error\_mean == min(test\_error\_mean))

## iter train\_error\_mean train\_error\_std test\_error\_mean test\_error\_std

## 1 17 0.0082568 0.002338999 0.0550458 0.01160461

## 2 25 0.0018350 0.001716352 0.0550458 0.01004998

## 3 29 0.0009176 0.001123826 0.0550458 0.01421269

## 4 32 0.0009176 0.001123826 0.0550458 0.01535140

## 5 33 0.0004588 0.000917600 0.0550458 0.01535140

## 6 80 0.0000000 0.000000000 0.0550458 0.01004998

**H2O**

H2O is another popular package for machine learning in R. We will first set up the session and create training and test data:

library(h2o)

h2o.init(nthreads = -1)

## Connection successful!

##

## R is connected to the H2O cluster:

## H2O cluster uptime: 2 hours 50 minutes

## H2O cluster timezone: Europe/Berlin

## H2O data parsing timezone: UTC

## H2O cluster version: 3.20.0.8

## H2O cluster version age: 2 months and 8 days

## H2O cluster name: H2O\_started\_from\_R\_shiringlander\_lci733

## H2O cluster total nodes: 1

## H2O cluster total memory: 3.31 GB

## H2O cluster total cores: 8

## H2O cluster allowed cores: 8

## H2O cluster healthy: TRUE

## H2O Connection ip: localhost

## H2O Connection port: 54321

## H2O Connection proxy: NA

## H2O Internal Security: FALSE

## H2O API Extensions: XGBoost, Algos, AutoML, Core V3, Core V4

## R Version: R version 3.5.1 (2018-07-02)

h2o.no\_progress()

data\_hf <- as.h2o(ml\_data)

splits <- h2o.splitFrame(data\_hf,

ratios = 0.75,

seed = 1)

train <- splits[[1]]

test <- splits[[2]]

response <- "Private"

features <- setdiff(colnames(train), response)

**Gradient Boosting**

The [Gradient Boosting](http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/gbm.html) implementation can be used as such:

h2o\_gbm <- h2o.gbm(x = features,

y = response,

training\_frame = train,

nfolds = 3) # cross-validation

h2o\_gbm

## Model Details:

## ==============

##

## H2OBinomialModel: gbm

## Model ID: GBM\_model\_R\_1543499512871\_1815

## Model Summary:

## number\_of\_trees number\_of\_internal\_trees model\_size\_in\_bytes min\_depth

## 1 50 50 13001 5

## max\_depth mean\_depth min\_leaves max\_leaves mean\_leaves

## 1 5 5.00000 8 21 15.74000

##

##

## H2OBinomialMetrics: gbm

## \*\* Reported on training data. \*\*

##

## MSE: 0.00244139

## RMSE: 0.04941043

## LogLoss: 0.02582422

## Mean Per-Class Error: 0

## AUC: 1

## Gini: 1

##

## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:

## No Yes Error Rate

## No 160 0 0.000000 =0/160

## Yes 0 419 0.000000 =0/419

## Totals 160 419 0.000000 =0/579

##

## Maximum Metrics: Maximum metrics at their respective thresholds

## metric threshold value idx

## 1 max f1 0.671121 1.000000 246

## 2 max f2 0.671121 1.000000 246

## 3 max f0point5 0.671121 1.000000 246

## 4 max accuracy 0.671121 1.000000 246

## 5 max precision 0.996764 1.000000 0

## 6 max recall 0.671121 1.000000 246

## 7 max specificity 0.996764 1.000000 0

## 8 max absolute\_mcc 0.671121 1.000000 246

## 9 max min\_per\_class\_accuracy 0.671121 1.000000 246

## 10 max mean\_per\_class\_accuracy 0.671121 1.000000 246

##

## Gains/Lift Table: Extract with `h2o.gainsLift(, )` or `h2o.gainsLift(, valid=, xval=)`

##

## H2OBinomialMetrics: gbm

## \*\* Reported on cross-validation data. \*\*

## \*\* 3-fold cross-validation on training data (Metrics computed for combined holdout predictions) \*\*

##

## MSE: 0.05688845

## RMSE: 0.238513

## LogLoss: 0.2007733

## Mean Per-Class Error: 0.09630817

## AUC: 0.9668929

## Gini: 0.9337858

##

## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:

## No Yes Error Rate

## No 133 27 0.168750 =27/160

## Yes 10 409 0.023866 =10/419

## Totals 143 436 0.063903 =37/579

##

## Maximum Metrics: Maximum metrics at their respective thresholds

## metric threshold value idx

## 1 max f1 0.400785 0.956725 265

## 2 max f2 0.132011 0.972352 287

## 3 max f0point5 0.725883 0.953442 229

## 4 max accuracy 0.400785 0.936097 265

## 5 max precision 0.997925 1.000000 0

## 6 max recall 0.009298 1.000000 381

## 7 max specificity 0.997925 1.000000 0

## 8 max absolute\_mcc 0.400785 0.837212 265

## 9 max min\_per\_class\_accuracy 0.811928 0.906250 224

## 10 max mean\_per\_class\_accuracy 0.725883 0.912552 229

##

## Gains/Lift Table: Extract with `h2o.gainsLift(, )` or `h2o.gainsLift(, valid=, xval=)`

## Cross-Validation Metrics Summary:

## mean sd cv\_1\_valid cv\_2\_valid

## accuracy 0.939574 6.4933195E-4 0.9390863 0.9408602

## auc 0.9701875 0.007612803 0.9708713 0.98301804

## err 0.060425993 6.4933195E-4 0.060913704 0.059139784

## err\_count 11.666667 0.33333334 12.0 11.0

## f0point5 0.95418453 0.006589541 0.9537167 0.96582466

## f1 0.95859224 4.803105E-4 0.9577465 0.9594096

## f2 0.96321476 0.006296414 0.96181047 0.95307916

## lift\_top\_group 1.3816328 0.012157884 1.3971632 1.3576642

## logloss 0.20019953 0.016917419 0.2080731 0.16776533

## max\_per\_class\_error 0.12948361 0.029007828 0.125 0.08163265

## mcc 0.84875494 0.001501441 0.84894496 0.85125524

## mean\_per\_class\_accuracy 0.9184681 0.009156114 0.9197695 0.93363625

## mean\_per\_class\_error 0.08153185 0.009156114 0.0802305 0.066363774

## mse 0.056778133 0.0035938106 0.06340453 0.05105359

## precision 0.95136136 0.010758136 0.951049 0.9701493

## r2 0.7161539 0.014445015 0.68836546 0.7368911

## recall 0.96641994 0.010696565 0.964539 0.9489051

## rmse 0.23804487 0.007509063 0.25180256 0.22595042

## specificity 0.87051636 0.029007828 0.875 0.9183673

## cv\_3\_valid

## accuracy 0.93877554

## auc 0.9566731

## err 0.06122449

## err\_count 12.0

## f0point5 0.94301224

## f1 0.95862067

## f2 0.9747546

## lift\_top\_group 1.3900709

## logloss 0.22476016

## max\_per\_class\_error 0.18181819

## mcc 0.84606457

## mean\_per\_class\_accuracy 0.9019987

## mean\_per\_class\_error 0.09800129

## mse 0.055876285

## precision 0.9328859

## r2 0.72320527

## recall 0.9858156

## rmse 0.23638165

## specificity 0.8181818

We can calculate performance on test data with h2o.performance():

h2o.performance(h2o\_gbm, test)

## H2OBinomialMetrics: gbm

##

## MSE: 0.03509102

## RMSE: 0.187326

## LogLoss: 0.1350709

## Mean Per-Class Error: 0.05216017

## AUC: 0.9770811

## Gini: 0.9541623

##

## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:

## No Yes Error Rate

## No 48 4 0.076923 =4/52

## Yes 4 142 0.027397 =4/146

## Totals 52 146 0.040404 =8/198

##

## Maximum Metrics: Maximum metrics at their respective thresholds

## metric threshold value idx

## 1 max f1 0.580377 0.972603 136

## 2 max f2 0.214459 0.979730 146

## 3 max f0point5 0.907699 0.979827 127

## 4 max accuracy 0.580377 0.959596 136

## 5 max precision 0.997449 1.000000 0

## 6 max recall 0.006710 1.000000 187

## 7 max specificity 0.997449 1.000000 0

## 8 max absolute\_mcc 0.580377 0.895680 136

## 9 max min\_per\_class\_accuracy 0.821398 0.952055 131

## 10 max mean\_per\_class\_accuracy 0.821398 0.956797 131

##

## Gains/Lift Table: Extract with `h2o.gainsLift(, )` or `h2o.gainsLift(, valid=, xval=)`

**XGBoost**

Alternatively, we can also use the [XGBoost](http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/xgboost.html) implementation of H2O:

h2o\_xgb <- h2o.xgboost(x = features,

y = response,

training\_frame = train,

nfolds = 3)

h2o\_xgb

## Model Details:

## ==============

##

## H2OBinomialModel: xgboost

## Model ID: XGBoost\_model\_R\_1543499512871\_2178

## Model Summary:

## number\_of\_trees

## 1 50

##

##

## H2OBinomialMetrics: xgboost

## \*\* Reported on training data. \*\*

##

## MSE: 0.25

## RMSE: 0.5

## LogLoss: 0.6931472

## Mean Per-Class Error: 0.5

## AUC: 0.5

## Gini: 0

##

## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:

## No Yes Error Rate

## No 0 160 1.000000 =160/160

## Yes 0 419 0.000000 =0/419

## Totals 0 579 0.276339 =160/579

##

## Maximum Metrics: Maximum metrics at their respective thresholds

## metric threshold value idx

## 1 max f1 0.500000 0.839679 0

## 2 max f2 0.500000 0.929047 0

## 3 max f0point5 0.500000 0.765996 0

## 4 max accuracy 0.500000 0.723661 0

## 5 max precision 0.500000 0.723661 0

## 6 max recall 0.500000 1.000000 0

## 7 max specificity 0.500000 0.000000 0

## 8 max absolute\_mcc 0.500000 0.000000 0

## 9 max min\_per\_class\_accuracy 0.500000 0.000000 0

## 10 max mean\_per\_class\_accuracy 0.500000 0.500000 0

##

## Gains/Lift Table: Extract with `h2o.gainsLift(, )` or `h2o.gainsLift(, valid=, xval=)`

##

## H2OBinomialMetrics: xgboost

## \*\* Reported on cross-validation data. \*\*

## \*\* 3-fold cross-validation on training data (Metrics computed for combined holdout predictions) \*\*

##

## MSE: 0.25

## RMSE: 0.5

## LogLoss: 0.6931472

## Mean Per-Class Error: 0.5

## AUC: 0.5

## Gini: 0

##

## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:

## No Yes Error Rate

## No 0 160 1.000000 =160/160

## Yes 0 419 0.000000 =0/419

## Totals 0 579 0.276339 =160/579

##

## Maximum Metrics: Maximum metrics at their respective thresholds

## metric threshold value idx

## 1 max f1 0.500000 0.839679 0

## 2 max f2 0.500000 0.929047 0

## 3 max f0point5 0.500000 0.765996 0

## 4 max accuracy 0.500000 0.723661 0

## 5 max precision 0.500000 0.723661 0

## 6 max recall 0.500000 1.000000 0

## 7 max specificity 0.500000 0.000000 0

## 8 max absolute\_mcc 0.500000 0.000000 0

## 9 max min\_per\_class\_accuracy 0.500000 0.000000 0

## 10 max mean\_per\_class\_accuracy 0.500000 0.500000 0

##

## Gains/Lift Table: Extract with `h2o.gainsLift(, )` or `h2o.gainsLift(, valid=, xval=)`

## Cross-Validation Metrics Summary:

## mean sd cv\_1\_valid cv\_2\_valid

## accuracy 0.720723 0.032527234 0.77294683 0.72820514

## auc 0.5 0.0 0.5 0.5

## err 0.27927703 0.032527234 0.22705314 0.2717949

## err\_count 53.333332 3.756476 47.0 53.0

## f0point5 0.7629575 0.029264713 0.8097166 0.77006507

## f1 0.83686095 0.022139339 0.8719346 0.84273

## f2 0.9273414 0.010952134 0.94451004 0.93053734

## lift\_top\_group 1.0 0.0 1.0 1.0

## logloss 0.6931472 4.8956235E-17 0.6931472 0.6931472

## max\_per\_class\_error 1.0 0.0 1.0 1.0

## mcc 0.0 NaN NaN NaN

## mean\_per\_class\_accuracy 0.5 0.0 0.5 0.5

## mean\_per\_class\_error 0.5 0.0 0.5 0.5

## mse 0.25 0.0 0.25 0.25

## precision 0.720723 0.032527234 0.77294683 0.72820514

## r2 -0.2677759 0.08917216 -0.42450133 -0.2631212

## recall 1.0 0.0 1.0 1.0

## rmse 0.5 0.0 0.5 0.5

## specificity 0.0 0.0 0.0 0.0

## cv\_3\_valid

## accuracy 0.66101694

## auc 0.5

## err 0.33898306

## err\_count 60.0

## f0point5 0.7090909

## f1 0.79591835

## f2 0.90697676

## lift\_top\_group 1.0

## logloss 0.6931472

## max\_per\_class\_error 1.0

## mcc NaN

## mean\_per\_class\_accuracy 0.5

## mean\_per\_class\_error 0.5

## mse 0.25

## precision 0.66101694

## r2 -0.115705125

## recall 1.0

## rmse 0.5

## specificity 0.0

And use it just as before:

h2o.performance(h2o\_xgb, test)

## H2OBinomialMetrics: xgboost

##

## MSE: 0.25

## RMSE: 0.5

## LogLoss: 0.6931472

## Mean Per-Class Error: 0.5

## AUC: 0.5

## Gini: 0

##

## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:

## No Yes Error Rate

## No 0 52 1.000000 =52/52

## Yes 0 146 0.000000 =0/146

## Totals 0 198 0.262626 =52/198

##

## Maximum Metrics: Maximum metrics at their respective thresholds

## metric threshold value idx

## 1 max f1 0.500000 0.848837 0

## 2 max f2 0.500000 0.933504 0

## 3 max f0point5 0.500000 0.778252 0

## 4 max accuracy 0.500000 0.737374 0

## 5 max precision 0.500000 0.737374 0

## 6 max recall 0.500000 1.000000 0

## 7 max specificity 0.500000 0.000000 0

## 8 max absolute\_mcc 0.500000 0.000000 0

## 9 max min\_per\_class\_accuracy 0.500000 0.000000 0

## 10 max mean\_per\_class\_accuracy 0.500000 0.500000 0

##

## Gains/Lift Table: Extract with `h2o.gainsLift(, )` or `h2o.gainsLift(, valid=, xval=)`

**Video**

**Slides**

sessionInfo()

## R version 3.5.1 (2018-07-02)

## Platform: x86\_64-apple-darwin15.6.0 (64-bit)

## Running under: macOS 10.14.1

##

## Matrix products: default

## BLAS: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRblas.0.dylib

## LAPACK: /Library/Frameworks/R.framework/Versions/3.5/Resources/lib/libRlapack.dylib

##

## locale:

## [1] en\_US.UTF-8/en\_US.UTF-8/en\_US.UTF-8/C/en\_US.UTF-8/en\_US.UTF-8

##

## attached base packages:

## [1] stats graphics grDevices utils datasets methods base

##

## other attached packages:

## [1] h2o\_3.20.0.8 bindrcpp\_0.2.2 xgboost\_0.71.2 caret\_6.0-80

## [5] lattice\_0.20-38 ISLR\_1.2 forcats\_0.3.0 stringr\_1.3.1

## [9] dplyr\_0.7.7 purrr\_0.2.5 readr\_1.1.1 tidyr\_0.8.2

## [13] tibble\_1.4.2 ggplot2\_3.1.0 tidyverse\_1.2.1

##

## loaded via a namespace (and not attached):

## [1] nlme\_3.1-137 bitops\_1.0-6 lubridate\_1.7.4

## [4] dimRed\_0.1.0 httr\_1.3.1 rprojroot\_1.3-2

## [7] tools\_3.5.1 backports\_1.1.2 utf8\_1.1.4

## [10] R6\_2.3.0 rpart\_4.1-13 lazyeval\_0.2.1

## [13] colorspace\_1.3-2 nnet\_7.3-12 withr\_2.1.2

## [16] gbm\_2.1.4 gridExtra\_2.3 tidyselect\_0.2.5

## [19] compiler\_3.5.1 cli\_1.0.1 rvest\_0.3.2

## [22] xml2\_1.2.0 bookdown\_0.7 scales\_1.0.0

## [25] sfsmisc\_1.1-2 DEoptimR\_1.0-8 robustbase\_0.93-3

## [28] digest\_0.6.18 rmarkdown\_1.10 pkgconfig\_2.0.2

## [31] htmltools\_0.3.6 rlang\_0.3.0.1 readxl\_1.1.0

## [34] ddalpha\_1.3.4 rstudioapi\_0.8 bindr\_0.1.1

## [37] jsonlite\_1.5 ModelMetrics\_1.2.2 RCurl\_1.95-4.11

## [40] magrittr\_1.5 Matrix\_1.2-15 fansi\_0.4.0

## [43] Rcpp\_0.12.19 munsell\_0.5.0 abind\_1.4-5

## [46] stringi\_1.2.4 yaml\_2.2.0 MASS\_7.3-51.1

## [49] plyr\_1.8.4 recipes\_0.1.3 grid\_3.5.1

## [52] pls\_2.7-0 crayon\_1.3.4 haven\_1.1.2

## [55] splines\_3.5.1 hms\_0.4.2 knitr\_1.20

## [58] pillar\_1.3.0 reshape2\_1.4.3 codetools\_0.2-15

## [61] stats4\_3.5.1 CVST\_0.2-2 magic\_1.5-9

## [64] glue\_1.3.0 evaluate\_0.12 blogdown\_0.9

## [67] data.table\_1.11.8 modelr\_0.1.2 foreach\_1.4.4

## [70] cellranger\_1.1.0 gtable\_0.2.0 kernlab\_0.9-27

## [73] assertthat\_0.2.0 DRR\_0.0.3 xfun\_0.4

## [76] gower\_0.1.2 prodlim\_2018.04.18 broom\_0.5.0

## [79] e1071\_1.7-0 class\_7.3-14 survival\_2.43-1

## [82] geometry\_0.3-6 timeDate\_3043.102 RcppRoll\_0.3.0

## [85] iterators\_1.0.10 lava\_1.6.3 ipred\_0.9-8