Gradient Boosting

Jan-Philipp Kolb

24 Mai, 2019

Gradient Boosting Machines (GBMs)

- The following slides are based on UC Business Analytics R Programming Guide on **gbm regression**
- GBMs are extremely popular, successful across many domains and one of the leading methods for winning **Kaggle competitions**.
- GBMs build an ensemble of flat and weak successive trees with each tree learning and improving on the previous.
- When combined, these many weak successive trees produce a powerful "committee" that are often hard to beat with other algorithms.

```
library(rsample)      # data splitting
library(gbm)          # basic implementation
library(xgboost)      # a faster implementation of gbm
library(caret)        # aggregator package - machine learning
library(pdp)          # model visualization
library(ggplot2)      # model visualization
library(lime)         # model visualization
```

The dataset

```
set.seed(123)
ames_split <- initial_split(AmesHousing::make_ames(), prop = .7)
ames_train <- training(ames_split)
ames_test  <- testing(ames_split)
```

Advantages

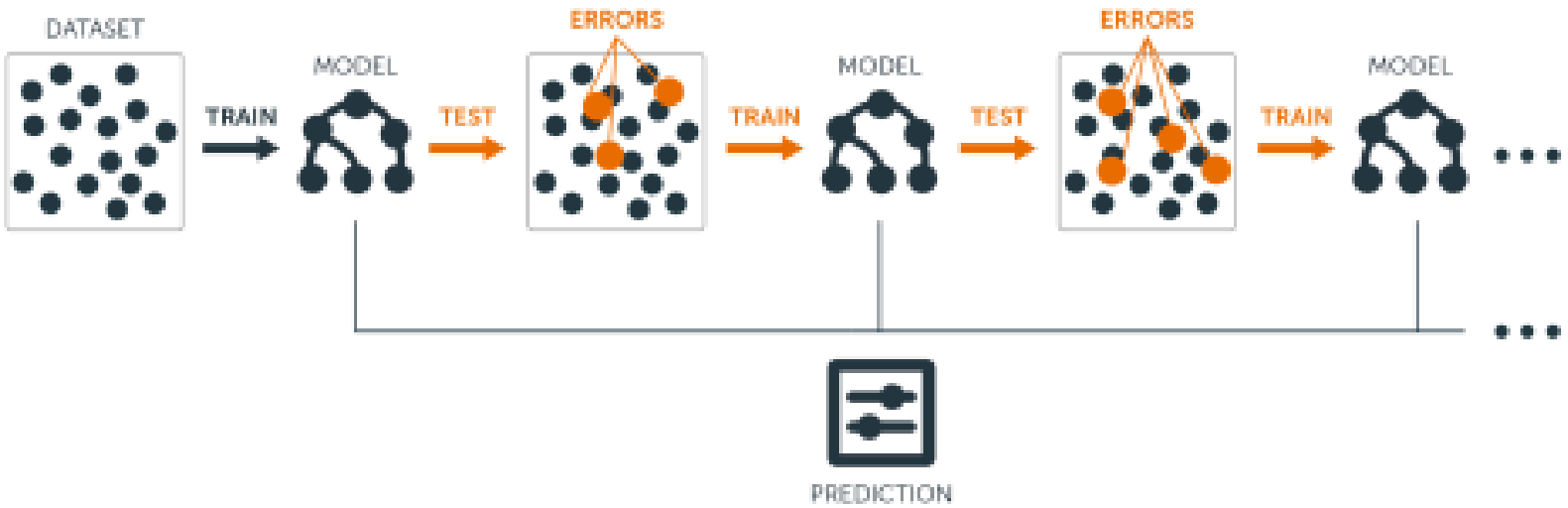
- Often provides predictive accuracy that cannot be beat.
- Lots of flexibility - can optimize on different loss functions and provides several hyperparameter tuning options that make the function fit very flexible.
- No data pre-processing required - often works great with categorical and numerical values as is.
- Handles missing data - imputation not required.

Disadvantages of GBMs

- GBMs will continue improving to minimize all errors. This can overemphasize outliers and cause overfitting. Must use cross-validation to neutralize.
- Computationally expensive - GBMs often require many trees ($>$1000) which can be time and memory exhaustive.
- The high flexibility results in many parameters that interact and influence heavily the behavior of the approach (number of iterations, tree depth, regularization parameters, etc.). This requires a large grid search during tuning.
- Less interpretable although this is easily addressed with various tools (variable importance, partial dependence plots, LIME, etc.).

The idea of GBMs

- Many machine learning models are founded on a single predictive model (i.e. linear regression, penalized models, naive Bayes, svm).
- Other approaches (bagging, random forests) are built on the idea of building an ensemble of models where each individual model predicts the outcome and the ensemble simply averages the predicted values.
- The idea of boosting is to add models to the ensemble sequentially.
- At each particular iteration, a new weak, base-learner model is trained with respect to the error of the whole ensemble learnt so far.



Important concepts

Base-learning models

- 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, boosted algorithms often use decision trees as the base-learner.

Training weak models

- A weak model is one whose error rate is only slightly better than random guessing.
- The idea behind boosting is that each sequential model builds a simple weak model to slightly improve the remaining errors.
- Shallow trees represent a weak learner.
- Trees with only 1-6 splits are used.
- Combining many weak models (versus strong ones) has a few benefits:
- Speed: Constructing weak models is computationally cheap.
- Accuracy improvement: Weak models allow the algorithm to learn slowly; making minor adjustments in new areas where it does not perform well. In general, statistical approaches that learn slowly tend to perform well.
- Avoids overfitting: Due to making only small incremental improvements with each model in the ensemble, this allows us to stop the learning process as soon as overfitting has been detected (typically by using cross-validation).

Sequential training with respect to errors

- Boosted trees are grown sequentially;
- each tree is grown using information from previously grown trees.
- The basic algorithm for boosted regression trees can be generalized to the following where x represents our features and y represents our response:

1.) Fit a decission tree: $F_1(x) = y$

2.) the next decission tree is fixed 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.) The next decission tree is fixed to the residuals of $F_2 : h_2(x) = y - F_2(x)$

5.) Add the new tree to the algorithm: $F_3(x) = F_2(x) + h_1(x)$

Continue this process until some mechanism (i.e. cross validation) tells us to stop.

Basic algorithm for boosted regression trees

The basic algorithm for boosted regression trees can be generalized to the following where the final model is simply a stagewise additive model of b individual regression trees:

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

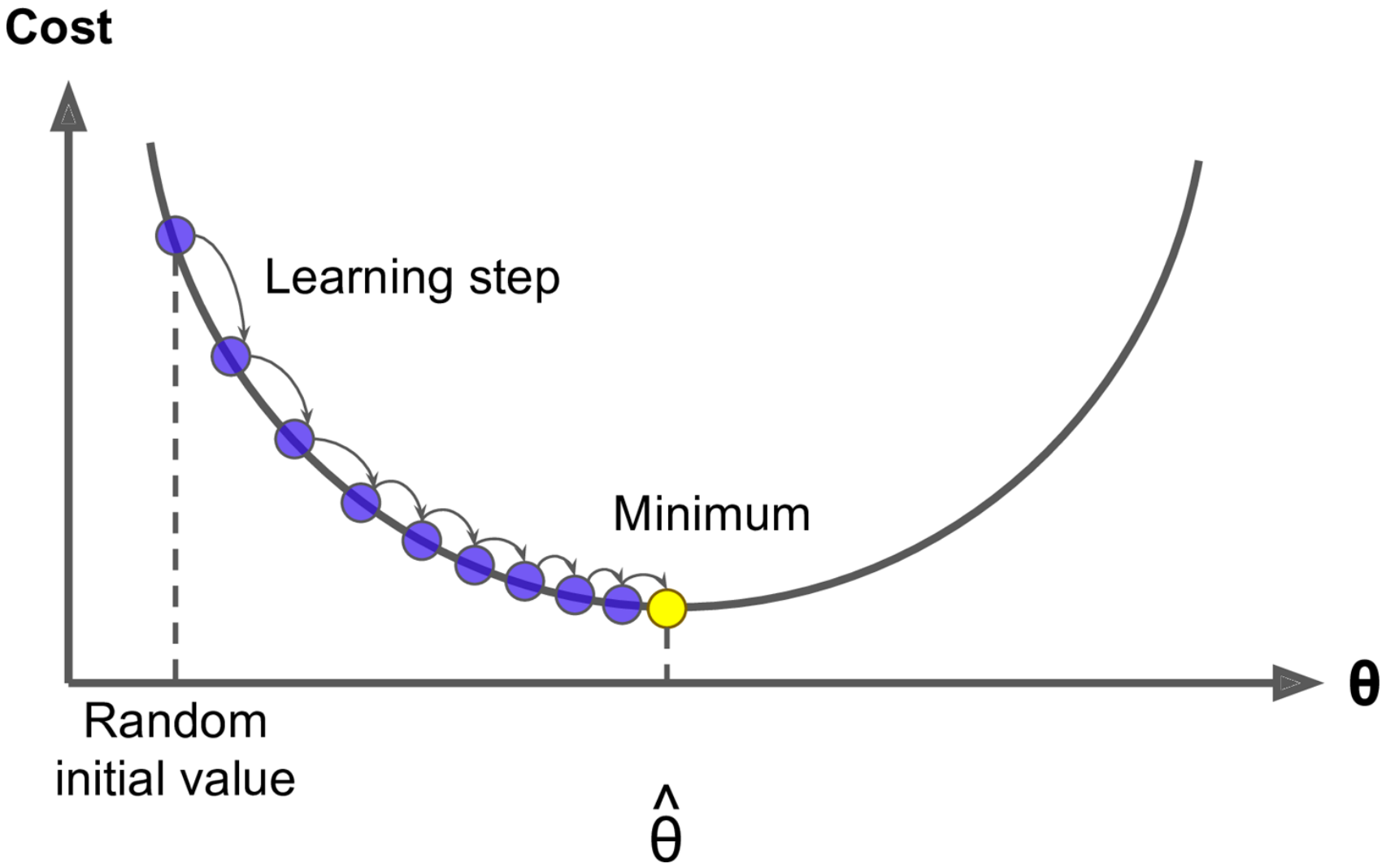
Gradient descent

- Many algorithms, including decision trees, focus on minimizing the residuals and, therefore, emphasize the MSE loss function.
- This algorithm outlines the approach of sequentially fitting regression trees to minimize the errors.
- This specific approach is how gradient boosting minimizes the mean squared error (MSE) loss function.
- Often we wish to focus on other loss functions such as mean absolute error (MAE) or to be able to apply the method to a classification problem with a loss function such as deviance.
- The name gradient boosting machines come from the fact that this procedure can be generalized to loss functions other than MSE.

A gradient descent algorithm

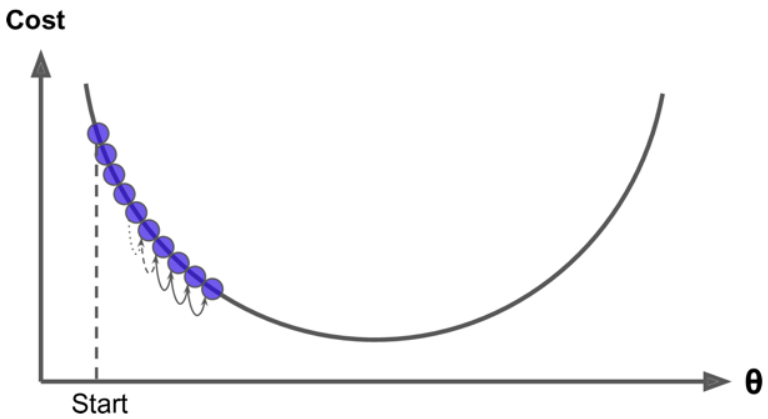
- Gradient boosting is considered a gradient descent algorithm.
- Gradient descent is a very generic optimization algorithm capable of finding optimal solutions to a wide range of problems.
- The general idea of gradient descent is to tweak parameters iteratively in order to minimize a cost function.
- Suppose you are a downhill skier racing your friend.
- A good strategy to beat your friend to the bottom is to take the path with the steepest slope.
- This is exactly what gradient descent does - it measures the local gradient of the loss (cost) function for a given set of parameters (Φ) and takes steps in the direction of the descending gradient.
- Once the gradient is zero, we have reached the minimum.

Gradient descent (Geron, 2017).

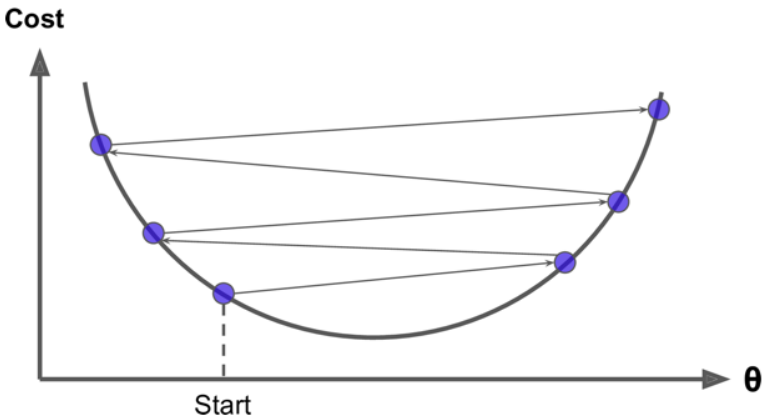


Gradient descent

- Gradient descent can be performed on any loss function that is differentiable.
- This allows GBMs to optimize different loss functions as desired
- An important parameter in gradient descent is the size of the steps which is determined by the learning rate.
- If the learning rate is too small, then the algorithm will take many iterations to find the minimum.
- But if the learning rate is too high, you might jump cross the minimum and end up further away than when you started.



a) too small

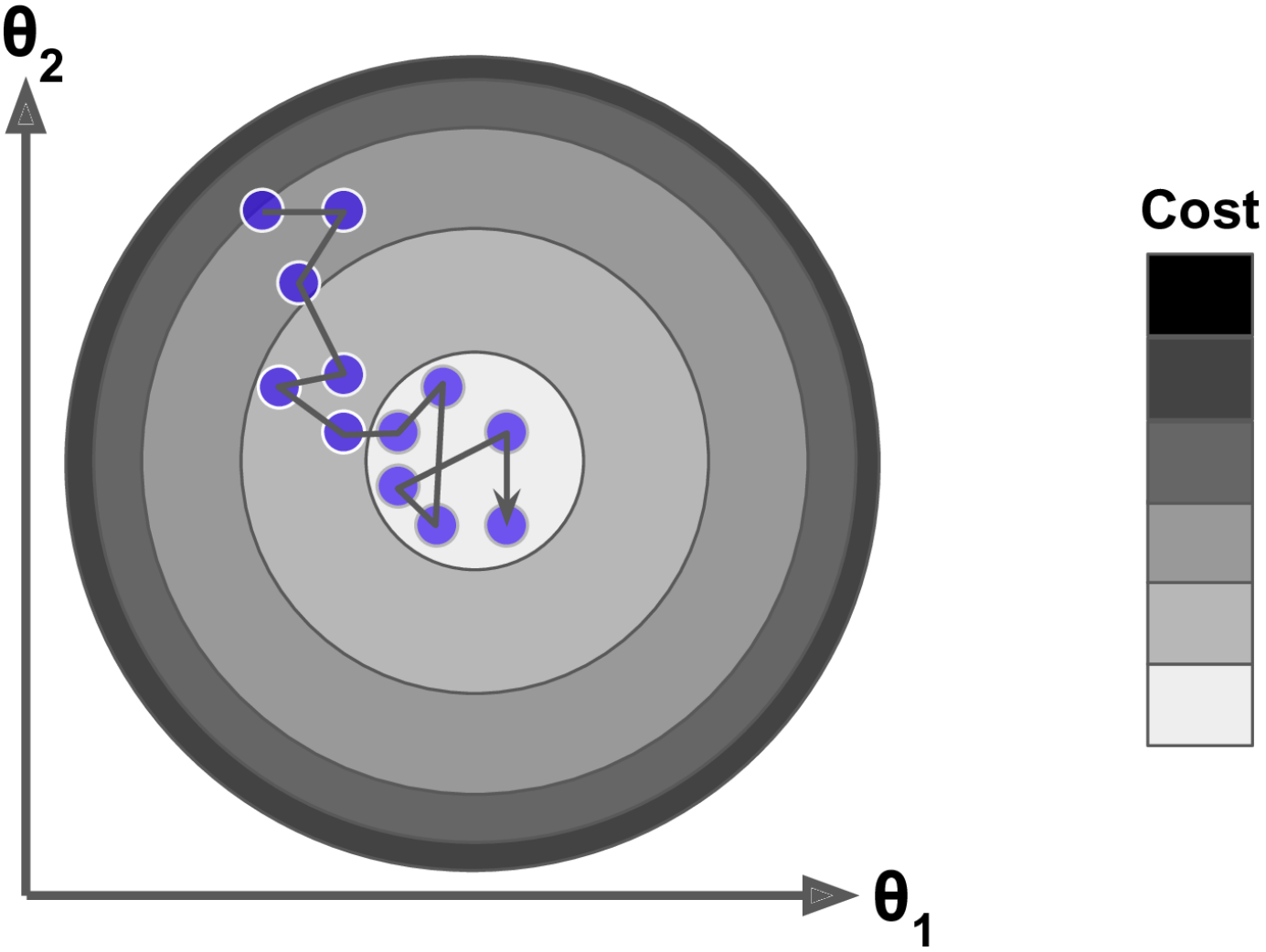


a) too big

Shape of cost functions

- Not all cost functions are convex (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 by sampling a fraction of the training observations (typically without replacement) and growing the next tree using that subsample.
- This makes the algorithm faster but the stochastic nature of random sampling also adds some random nature in descending the loss function gradient.
- Although this randomness does not allow the algorithm to find the absolute global minimum, it can actually help the algorithm jump out of local minima and off plateaus and get near the global minimum.

Stochastic gradient descent



Tuning GBM

- GBMs are highly flexible - many tuning parameters
- It is time consuming to find the optimal combination of hyperparameters

Number of trees

- GBMs often require many trees; however, unlike random forests GBMs can overfit so the goal is to find the optimal number of trees that minimize the loss function of interest with cross validation.

Tuning parameters

Depth of trees

- The number d of splits in each tree, which controls the complexity of the boosted ensemble.
- Often $d = 1$ works well, in which case each tree is a stump consisting of a single split. More commonly, d is greater than 1 but it is unlikely $d > 10$ will be required.

Learning rate

- The number d of splits in each tree, which controls the complexity of the boosted ensemble.
- Often $d = 1$ works well, in which case each tree is a stump consisting of a single split. -
- Normally, d is greater than 1 but it is unlikely $d > 10$ will be required.

Tuning parameters (II)

Subsampling

- Controls if a fraction of the available training observations is used.
- Using less than 100% of the training observations means you are implementing **stochastic gradient descent**.
- This can help to minimize overfitting and keep from getting stuck in a local minimum or plateau of the loss function gradient.

Package implementation

The most popular implementations of GBM in R:

GBM
The original R implementation of GBMs

XGBOOST
A fast and efficient gradient boosting framework (C++ backend).

H2O
A powerful java-based interface that provides parallel distributed algorithms and efficient productionalization.

The R-package gbm

The gbm R package is an implementation of extensions to Freund and Schapire's AdaBoost algorithm and Friedman's gradient boosting machine. This is the original R implementation of GBM.

Basic implementation

- two primary training functions: `gbm::gbm` and `gbm::gbm.fit`.
- `gbm::gbm` uses the formula interface to specify your model
- `gbm::gbm.fit` requires the separated `x` and `y` matrices (more efficient with many variables).

- The default settings in `gbm` includes a learning rate (shrinkage) of 0.001.
- This is a very small learning rate and typically requires a large number of trees to find the minimum MSE.
- `gbm` uses a default number of trees of 100, which is rarely sufficient. - - The default depth of each tree (interaction.depth) is 1, which means we are ensembling a bunch of stumps. Lastly, I also include cv.folds to perform a 5 fold cross validation. The model took about 90 seconds to run and the results show that our MSE loss function is minimized with 10,000 trees.

Train GBM model

```
set.seed(123)
gbm.fit <- gbm(formula = Sale_Price ~ .,distribution = "gaussian
  data = ames_train,n.trees = 10000,interaction.depth = 1,
  shrinkage = 0.001,cv.folds = 5,
  n.cores = NULL, # will use all cores by default
  verbose = FALSE)
# print results

print(gbm.fit)
```

The output object...

... is a list containing several modelling and results information. We can access this information with regular indexing; I recommend you take some time to dig around in the object to get comfortable with its components. Here, we see that the minimum CV RMSE is 29133 (this means on average our model is about $29,133 off from the actual sales price) but the plot also illustrates that the CV error is still decreasing at 10,000 trees.

Get MSE and compute RMSE

```
sqrt(min(gbm.fit$cv.error))
# plot loss function as a result of n trees added to the ensembl
gbm.perf(gbm.fit, method = "cv")
```

Tuning GBMs

- The learning rate is increased to take larger steps down the gradient descent,
- The number of trees is reduced (since we are reducing the learning rate), and increase the depth of each tree from using a single split to 3 splits.

```
set.seed(123)
gbm.fit2 <- gbm(formula = Sale_Price ~ .,
  distribution = "gaussian",data = ames_train,
  n.trees = 5000,interaction.depth = 3,shrinkage = 0.1,
  cv.folds = 5,n.cores = NULL,verbose = FALSE)

# find index for n trees with minimum CV error
min_MSE <- which.min(gbm.fit2$cv.error)

# get MSE and compute RMSE
sqrt(gbm.fit2$cv.error[min_MSE])
# plot loss function as a result of n trees added to the ensembl
gbm.perf(gbm.fit2, method = "cv")
```

Grid search

- the minimum number of observations allowed in the trees terminal nodes (`n.minobsinnode`) is varied

```
hyper_grid <- expand.grid(
  shrinkage = c(.01, .1, .3),
  interaction.depth = c(1, 3, 5),
  n.minobsinnode = c(5, 10, 15),
  bag.fraction = c(.65, .8, 1),
  optimal_trees = 0,# a place to dump results
  min_RMSE = 0
)

# total number of combinations
nrow(hyper_grid)
```

- We loop through each hyperparameter combination (5,000 trees).
- To speed up the tuning process, instead of performing 5-fold CV I train on 75% of the training observations and evaluate performance on the remaining 25%.
- When using `train.fraction` it will take the first XX% of the data so its important to randomize your rows in case their is any logic behind the ordering of the data (i.e. ordered by neighborhood).

- First, our top model has better performance than our previously fitted model above, with the RMSE nearly $3,000 lower. Second, looking at the top 10 models we see that:
- none of the top models used a learning rate of 0.3; small incremental steps down the gradient descent appears to work best,
- none of the top models used stumps (`interaction.depth = 1`); there are likely stome important interactions that the deeper trees are able to capture,
- adding a stochastic component with bag.fraction < 1 seems to help; there may be some local minimas in our loss function gradient,

Randomize data and loop over hyperparameter grid

```
# randomize data
random_index <- sample(1:nrow(ames_train), nrow(ames_train))
random_ames_train <- ames_train[random_index, ]
# grid search
for(i in 1:nrow(hyper_grid)) {
  set.seed(123)
  # train model
  gbm.tune <- gbm(
    formula = Sale_Price ~ .,
    distribution = "gaussian",
    data = random_ames_train,
    n.trees = 5000,
    interaction.depth = hyper_grid$interaction.depth[i],
    shrinkage = hyper_grid$shrinkage[i],
    n.minobsinnode = hyper_grid$n.minobsinnode[i],
    bag.fraction = hyper_grid$bag.fraction[i],
```

Refine the search - adjust the grid

```
# modify hyperparameter grid
hyper_grid <- expand.grid(
  shrinkage = c(.01, .05, .1),
  interaction.depth = c(3, 5, 7),
  n.minobsinnode = c(5, 7, 10),
  bag.fraction = c(.65, .8, 1),
  optimal_trees = 0,                    # a place to dump results
  min_RMSE = 0                          # a place to dump results
)

# total number of combinations
nrow(hyper_grid)
```

The final model

```
set.seed(123)

# train GBM model
gbm.fit.final <- gbm(
  formula = Sale_Price ~ .,
  distribution = "gaussian",
  data = ames_train,
  n.trees = 483,
  interaction.depth = 5,
  shrinkage = 0.1,
  n.minobsinnode = 5,
  bag.fraction = .65,
  train.fraction = 1,
  n.cores = NULL, # will use all cores by default
  verbose = FALSE
  )
```

Visualizing - Variable importance

- cBars allows you to adjust the number of variables to show

```
par(mar = c(5, 8, 1, 1))
summary(gbm.fit.final,cBars = 10,
  # also can use permutation.test.gbm
  method = relative.influence,las = 2)
```

The vip package

```
devtools::install_github("koalaverse/vip")

vip::vip(gbm.fit.final)
```

Partial dependence plots

- PDPs plot the change in the average predicted value as specified feature(s) vary over their marginal distribution.
- The PDP plot below displays the average change in predicted sales price as we vary `Gr_Liv_Area` while holding all other variables constant.
- We then average the sale price across all the observations.
- This PDP illustrates how the predicted sales price increases as the square footage of the ground floor in a house increases.

Partial dependence plot - Gr_Liv_Area

```
gbm.fit.final %>%
  partial(pred.var = "Gr_Liv_Area", n.trees = gbm.fit.final$n.tr
  autoplot(rug = TRUE, train = ames_train) +
  scale_y_continuous(labels = scales::dollar)
```

ICE curves . . .

- . . . are an extension of PDP plots but the change in the predicted response variable is plotted as we vary each predictor variable.
- When the curves have a wide range of intercepts and are consequently "stacked" on each other, heterogeneity in the response variable values due to marginal changes in the predictor variable of interest can be difficult to discern.
- The centered ICE can help draw these inferences out and can highlight any strong heterogeneity in our results.
- The results show that most observations follow a common trend as `Gr_Liv_Area` increases;
- the centered ICE plot highlights a few observations that deviate from the common trend.

```
ice1 <- gbm.fit.final %>%
  partial(
    pred.var = "Gr_Liv_Area",
    n.trees = gbm.fit.final$n.trees,
    grid.resolution = 100,
    ice = TRUE
    ) %>%
  autoplot(rug = TRUE, train = ames_train, alpha = .1) +
  ggtitle("Non-centered") +
  scale_y_continuous(labels = scales::dollar)

ice2 <- gbm.fit.final %>%
  partial(
    pred.var = "Gr_Liv_Area",
    n.trees = gbm.fit.final$n.trees,
    grid.resolution = 100,
    ice = TRUE
    ) %>%
  autoplot(rug = TRUE, train = ames_train, alpha = .1, center =
```

LIME

LIME is a newer procedure for understanding why a prediction resulted in a given value for a single observation. You can read more about LIME here. To use the lime package on a gbm model we need to define model type and prediction methods.

```
model_type.gbm <- function(x, ...) {
  return("regression")
}

predict_model.gbm <- function(x, newdata, ...) {
  pred <- predict(x, newdata, n.trees = x$n.trees)
  return(as.data.frame(pred))
}
```

We can now apply to our two observations. The results show the predicted value (Case 1: 118K Dollar, Case 2: 161K Dollar), local model fit (both are relatively poor), and the most influential variables driving the predicted value for each observation.

```
# get a few observations to perform local interpretation on
local_obs <- ames_test[1:2, ]

# apply LIME
explainer <- lime(ames_train, gbm.fit.final)
explanation <- explain(local_obs, explainer, n_features = 5)
plot_features(explanation)
```

Predicting

Once you have decided on a final model you will likely want to use the model to predict on new observations. Like most models, we simply use the predict function; however, we also need to supply the number of trees to use (see ?predict.gbm for details). We see that our RMSE for our test set is very close to the RMSE we obtained on our best gbm model.

```
# predict values for test data
pred <- predict(gbm.fit.final, n.trees = gbm.fit.final$n.trees,

# results
caret::RMSE(pred, ames_test$Sale_Price)
```

XGBOOST

The xgboost R package provides an R API to "Extreme Gradient Boosting", which is an efficient implementation of gradient boosting framework (apprx 10x faster than gbm). The xgboost/demo repository provides a wealth of information. You can also find a fairly comprehensive parameter tuning guide here. The xgboost package has been quite popular and successful on Kaggle for data mining competitions.

Features include:

- Provides built-in k-fold cross-validation -Stochastic GBM with column and row sampling (per split and per tree) for better generalization.
- Includes efficient linear model solver and tree learning algorithms.
- Parallel computation on a single machine.
- Supports various objective functions, including regression, classification and ranking.
- The package is made to be extensible, so that users are also allowed to define their own objectives easily.
- Apache 2.0 License.

Basic implementation

XGBoost only works with matrices that contain all numeric variables; consequently, we need to one hot encode our data. There are different ways to do this in R (i.e. Matrix::sparse.model.matrix, caret::dummyVars) but here we will use the vtreat package. vtreat is a robust package for data prep and helps to eliminate problems caused by missing values, novel categorical levels that appear in future data sets that were not in the training data, etc. However, vtreat is not very intuitive. I will not explain the functionalities but you can find more information here, here, and here.

The following applies vtreat to one-hot encode the training and testing data sets.

```
# variable names
features <- setdiff(names(ames_train), "Sale_Price")

# Create the treatment plan from the training data
treatplan <- vtreat::designTreatmentsZ(ames_train, features, ver

# Get the "clean" variable names from the scoreFrame
new_vars <- treatplan %>%
  magrittr::use_series(scoreFrame) %>%
  dplyr::filter(code %in% c("clean", "lev")) %>%
  magrittr::use_series(varName)

# Prepare the training data
features_train <- vtreat::prepare(treatplan, ames_train, varRest
response_train <- ames_train$Sale_Price

# Prepare the test data
```

xgboost provides different training functions (i.e. xgb.train which is just a wrapper for xgboost). However, to train an XGBoost we typically want to use xgb.cv, which incorporates cross-validation. The following trains a basic 5-fold cross validated XGBoost model with 1,000 trees. There are many parameters available in xgb.cv but the ones you have become more familiar with in this tutorial include the following default values:

- learning rate (eta): 0.3
- tree depth (max_depth): 6
- minimum node size (min_child_weight): 1
- percent of training data to sample for each tree (subsample –> equivalent to gbm's bag.fraction): 100%

```
# reproducibility
set.seed(123)

xgb.fit1 <- xgb.cv(
  data = features_train,
  label = response_train,
  nrounds = 1000,
  nfold = 5
```

The xgb.fit1 object contains lots of good information. In particular we can assess the `xgb.fit1$evaluation_log` to identify the minimum RMSE and the optimal number of trees for both the training data and the cross-validated error. We can see that the training error continues to decrease to 965 trees where the RMSE nearly reaches zero; however, the cross validated error reaches a minimum RMSE of 27,572 Dollar with only 60 trees.

```
# get number of trees that minimize error
xgb.fit1$evaluation_log %>%
  dplyr::summarise(
    ntrees.train = which(train_rmse_mean == min(train_rmse_mean)
    rmse.train   = min(train_rmse_mean),
    ntrees.test  = which(test_rmse_mean == min(test_rmse_mean))[
    rmse.test   = min(test_rmse_mean),
  )
##   ntrees.train rmse.train ntrees.test rmse.test
## 1          965  0.5022836          60  27572.31

# plot error vs number trees
```

A nice feature provided by xgb.cv is early stopping. This allows us to tell the function to stop running if the cross validated error does not improve for n continuous trees. For example, the above model could be re-run with the following where we tell it stop if we see no improvement for 10 consecutive trees. This feature will help us speed up the tuning process in the next section.

```
# reproducibility
set.seed(123)

xgb.fit2 <- xgb.cv(
  data = features_train,
  label = response_train,
  nrounds = 1000,
  nfold = 5,
  objective = "reg:linear",  # for regression models
  verbose = 0,               # silent,
  early_stopping_rounds = 10 # stop if no improvement for 10 con
)
```

Tuning

To tune the XGBoost model we pass parameters as a list object to the params argument. The most common parameters include:

- eta:controls the learning rate
- max_depth: tree depth
- min_child_weight: minimum number of observations required in each terminal node
- subsample: percent of training data to sample for each tree
- colsample_bytrees: percent of columns to sample from for each tree
  For example, if we wanted to specify specific values for these parameters we would extend the above model with the following parameters.

```r
# create parameter list
  params <- list(
    eta = .1,
    max_depth = 5,
    min_child_weight = 2,
    subsample = .8,
    colsample_bytree = .9
  )

# reproducibility
set.seed(123)

# train model
xgb.fit3 <- xgb.cv(
  params = params,
  data = features_train,
  label = response_train,
  nrounds = 1000,
  nfold = 5,
  objective = "reg:linear"  # for regression models
```

To perform a large search grid, we can follow the same procedure we did with gbm. We create our hyperparameter search grid along with columns to dump our results in. Here, I create a pretty large search grid consisting of 576 different hyperparameter combinations to model.

```
# create hyperparameter grid
hyper_grid <- expand.grid(
  eta = c(.01, .05, .1, .3),
  max_depth = c(1, 3, 5, 7),
  min_child_weight = c(1, 3, 5, 7),
  subsample = c(.65, .8, 1),
  colsample_bytree = c(.8, .9, 1),
  optimal_trees = 0,               # a place to dump results
  min_RMSE = 0                     # a place to dump results
)

nrow(hyper_grid)
```

Now I apply the same for loop procedure to loop through and apply a XGBoost model for each hyperparameter combination and dump the results in the hyper_grid data frame. Important note: if you plan to run this code be prepared to run it before going out to eat or going to bed as it the full search grid took 6 hours to run!

```
# grid search
for(i in 1:nrow(hyper_grid)) {

  # create parameter list
  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]
  )

  # reproducibility
  set.seed(123)
```

After assessing the results you would likely perform a few more grid searches to hone in on the parameters that appear to influence the model the most. In fact, here is a link to a great blog post that discusses a strategic approach to tuning with xgboost. However, for brevity, we'll just assume the top model in the above search is the globally optimal model. Once you've found the optimal model, we can fit our final model with xgb.train.

```
# parameter list
params <- list(
  eta = 0.01,
  max_depth = 5,
  min_child_weight = 5,
  subsample = 0.65,
  colsample_bytree = 1
)

# train final model
xgb.fit.final <- xgboost(
  params = params,
```

Visualizing

Variable importance xgboost provides built-in variable importance plotting. First, you need to create the importance matrix with xgb.importance and then feed this matrix into xgb.plot.importance. There are 3 variable importance measure:

- Gain: the relative contribution of the corresponding feature to the model calculated by taking each feature's contribution for each tree in the model. This is synonymous with gbm's relative.influence.
- Cover: the relative number of observations related to this feature. For example, if you have 100 observations, 4 features and 3 trees, and suppose feature1 is used to decide the leaf node for 10, 5, and 2 observations in tree1, tree2 and tree3 respectively; then the metric will count cover for this feature as 10+5+2 = 17 observations. This will be calculated for all the 4 features and the cover will be 17 expressed as a percentage for all features' cover metrics.
- Frequency: the percentage representing the relative number of times a particular feature occurs in the trees of the model. In the above example, if feature1 occurred in 2 splits, 1 split and 3 splits in each of

Partial dependence plots

PDP and ICE plots work similarly to how we implemented them with gbm. The only difference is you need to incorporate the training data within the partial function.

```
pdp <- xgb.fit.final %>%
  partial(pred.var = "Gr_Liv_Area_clean", n.trees = 1576, grid.r
  autoplot(rug = TRUE, train = features_train) +
  scale_y_continuous(labels = scales::dollar) +
  ggtitle("PDP")

ice <- xgb.fit.final %>%
  partial(pred.var = "Gr_Liv_Area_clean", n.trees = 1576, grid.r
  autoplot(rug = TRUE, train = features_train, alpha = .1, cente
  scale_y_continuous(labels = scales::dollar) +
  ggtitle("ICE")

gridExtra::grid.arrange(pdp, ice, nrow = 1)
```

LIME

LIME provides built-in functionality for xgboost objects (see ?model_type). However, just keep in mind that the local observations being analyzed need to be one-hot encoded in the same manner as we prepared the training and test data. Also, when you feed the training data into the lime::lime function be sure that you coerce it from a matrix to a data frame.

```
# one-hot encode the local observations to be assessed.
local_obs_onehot <- vtreat::prepare(treatplan, local_obs, varRes

# apply LIME
explainer <- lime(data.frame(features_train), xgb.fit.final)
explanation <- explain(local_obs_onehot, explainer, n_features =
plot_features(explanation)
```

Predicting

Lastly, we use predict to predict on new observations; however, unlike gbm we do not need to provide the number of trees. Our test set RMSE is only about $600 different than that produced by our gbm model.

```
# predict values for test data
pred <- predict(xgb.fit.final, features_test)

# results
caret::RMSE(pred, response_test)
## [1] 21319.3
```

Links - Boosting

- **Gradient Boosting Machines**
- **How to Visualize Gradient Boosting Decision Trees With XGBoost in Python**