UC business Analytics R programming Guide

Koji Mizumura

2019-05-29

Table of Contents

This is a practice of [UC business analytics R programming guide](http://uc-r.github.io/).

# Predictive Analysis

## Gradient boosting machines (GBM)

Gradient boosted machines (GBMs) are an extremely popular machine learning algorithm that have proven successful across many domains and is one of the learning methods for winning Kaggle competitions. Whereas [random forest](http://uc-r.github.io/random_forests) build anc ensemble of deepe independent trees, GBMs build an ensemple of shallow and weak successive trees with each tree leading 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. This tutorial will cover the fundamentals of GBMs for regression problems.

### tl;dr

This tutotrial serves as an introduction to the GBMs. This tutorial will cover the following material:

* [Replication Requirements](#GBM_RR): What you’ll need to reproduce the analysis in this tutorial.
* [Advantages & Disadvantages](#GBM_feature): Primary strengths and weaknesses of GBMs.
* [The idea](#GBM_idea): A quick overview of how GBMs work.
* [gbm](#GBM_gbm_pkg): Training and tuning with the gbm package
* [xgboost](#GBM_xgboost_pkg): Training and tuning with the xgboost package
* [h2o](#GBM_h2o_pkg): Training and tuning with the h2o package
* [Learning more](#GBM_Learn): Where you can learn more.

### Replication requirements

This tutorial leverages the following packages. Some of these packages play a supporting role; however, we demonstrate how to implement GBMs with several different packages and discuss the pros and cons to each.

library(rsample) # data splitting   
library(gbm) # basic implementation  
library(xgboost) # a faster implementation of gbm  
library(caret) # an aggregator package for performing many machine learning models  
library(h2o) # a java-based platform  
library(pdp) # model visualization  
library(ggplot2) # model visualization  
library(lime) # model visualization

To illustrate various GBM concepts, we will use the Ames housing data that has been included in the AmesHousing package.

# create training(70%) and test (30%) sets   
# Use set.seed for reproducibility  
  
set.seed(123)  
ames\_split <- initial\_split(AmesHousing::make\_ames(), prop = .7)  
ames\_train <- training(ames\_split)  
ames\_test <- testing(ames\_split)

**Important notice**: tree-based methods tend to perform well on unprocessed data (i.e., without normalizing, centering, scaling features). In this tutorial, I focus on how to implement GBMs with various packages. Although I do not pre-process the data, realize that you **can** improve model performance by spending time processing variable attributes.

### Advantages & Disadvantages

**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.

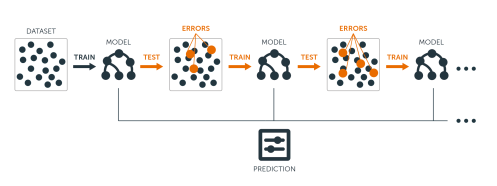
**Disadvantages** - GBMs will continue improving to minimize all errors. This can overemphasize outliers and cause overfitting. Must use cross-validation to neutralize. - Computationally expensive - GBMs often require many trees (>1000) which can be time and memory exhaustive. - The high flexibility 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

Several supervised machine learning models are founded on a single predictive model (i.e., liner regression, penalized models, naive Bayes, support vector machines). Alternatively, other approaches such as bagging and random forests are built on the idea of building an ensemble of models where each individal model predicts the outcome and then the ensemble simply averages the predictive values. The family of boosting methods is based on a different, constructive strategy of ensemble formation.

The main idea of boosting is to add new 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.

knitr::include\_graphics("C:/Protected/Data Science/UC Business Analytics/image/boosted-trees-process.png")



Let’s discuss each component of the previous sentence in closer detail because they are important.

**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 however, boosted algorithms almost always use decision trees as the base-learner. Consequently, this tutorial will discuss boosting in the context of regression trees.

**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. With regards to decision trees, shallow trees represent a weak learner. Commonly, 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 decision tree to the data:
2. We then fit the next decision tree to the residuals of the previous:
3. Add this new tree to our algorithm: ,
4. Fir the next decision tree to the residuals of :
5. Add this new tree to our algorithm:
6. Continue this process until some mmechasim (i.e., cross-validation) tells us to stop.

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

To illustrate the behaivor, assume the following and observations. The blue sine wave represents the true underlying function and the points represent observations that include some irriducible error (noise). The boosted prediction illustrates the adjusted predictions after each additional sequential tree is added to the algorithm. Initially, there are large errors which the boosted algorithm improves upon immediately but as the predictions get closer to the true underlying function you see each additional tree make small improvements in different areas across the feature space where errors remain. Towards the end of the gif, the predicted values nearly converge to the true underlying function.

**Gradient descent**

Many algorithms, including decision trees, focus on minimizing the residuals and, therefore, emphasize the MSE loss function. The algorithm discussed in the previous section 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.

However, 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.

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 a 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 sloep. This is is exactly what gradient descent does - it measures the local gradient of the loss (cost) function for a given set of parameter () and takes steps in the direction of the descending gradient. Once the gradient is zero, we have reached the minimum.

Gradient descent can be performed on any loss function that is differentiable. Consequently, this allows GBMs to optimize different loss functions as desired (see ESL, p. 360 for common loss functions). 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. On the other hand, if the learning rate is too high, you might jump cross the minimum and end up further away than when you started.

Moreover, 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 algorithms 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 helpt the algorithm jump out of local minima and off plateus get near the global minimum.

As we’ll see in the next section, there are several hyperparameter tuning options that allow us to address how we approach the gradient descent of our loss function.

#### Tuning

Part of the beauty and challenges of GBM is that they offer several tuning parameters. The beauty in this is GBMs are highly flexible. The challenge is that they can be time consuming to tune and find the optimal combination of hyperparamters. The most common hyperparameters that you will find in most GBM implementations include:

* **Number of trees**: The total number of trees to fit. 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.
* **Depth of trees**; The number of splits in each tree, which controls the complexity of the boosted ensembles. Often works well, in which case each tree is a stump consisting of a single split. More commonly, is greater than 1 but is unlikely will be required.
* **Learning rate**: Controls how quickly the algorithm proceeds down the gradient descent. Smaller values reduce the chance of overfitting but also increases the time to find the optimal fit. This is also called *shrinkage*.
* **Subsampling**: Controls whether or not you use a fraction of the available training observations. 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 gradien

Throughout this tutorial you’ll be exposed to additional hyperparameters that are specific to certain packages and can improve performance and/or the efficiency of training and tuning models.

#### Package implementation

There are many packages that implement GBMs and GBM variants. You can find a fairly comprehensive list [here](https://koalaverse.github.io/machine-learning-in-R/gradient-boosting-machines.html#gbm-software-in-r) and at the [CRAN Machine Learning Task View](https://cran.r-project.org/web/views/MachineLearning.html). However, the most popular implementations which we will cover in this plot include:

* [gbm](https://cran.r-project.org/web/packages/gbm/index.html): The original R implementation of GBMs
* [xgboost](https://cran.r-project.org/web/packages/xgboost/index.html): A fast and efficient gradient boosting framework (C++ backend)
* [h2o](https://cran.r-project.org/web/packages/gamboostLSS/index.html): A powerful java-based interface that provides parallel distributed algorithms and efficient productionalization.

#### gbm

The gbm R package is an implementation of extensions to Freund and Schapire’s [AdaBoost algorithm](http://www.site.uottawa.ca/~stan/csi5387/boost-tut-ppr.pdf) and Friedman’s [gradient boosting machine.](https://statweb.stanford.edu/~jhf/ftp/trebst.pdf) This is the original R implementation of GBM. A presentation is available [here](https://www.slideshare.net/mark_landry/gbm-package-in-r) by Mark Landry.

Features include:

* Stochastic GBM.
* Supports up to 1024 factor levels.
* Supports Classification and regression trees.
* Can incorporate many loss functions.
* Out-of-bag estimator for the optimal number of iterations is provided.
* Easy to overfit since early stopping -functionality is not automated in this package.
* If internal cross-validation is used, this can be parallelized to all cores on the machine.
* Currently undergoing a major refactoring & rewrite (and has been for some time).
* GPL-2/3 License.

#### Basic implementation

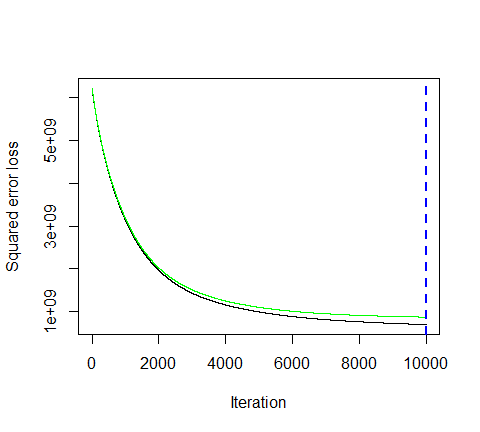
gbm has two primary training functions - gbm::gbm and gbm::gbm.fit. The primary difference is that gbm::gbm uses the formula interface to specify your model whereas gbm::gbm.fit requires the separated x and y matrices. When working with many variables it is more efficient to use the matrix rather than formula interface.

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. However, gbm uses a default number of trees of 100, which is rarely sufficient. Consequently, I crank it up to 10,000 trees. 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.

# for reproducibility  
set.seed(123)  
  
# train GBM model  
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)  
## gbm(formula = Sale\_Price ~ ., distribution = "gaussian", data = ames\_train,   
## n.trees = 10000, interaction.depth = 1, shrinkage = 0.001,   
## cv.folds = 5, verbose = FALSE, n.cores = NULL)  
## A gradient boosted model with gaussian loss function.  
## 10000 iterations were performed.  
## The best cross-validation iteration was 9998.  
## There were 80 predictors of which 47 had non-zero influence.  
# gbm(formula = Sale\_Price ~ ., distribution = "gaussian", data = ames\_train,   
## n.trees = 10000, interaction.depth = 1, shrinkage = 0.001,   
## cv.folds = 5, verbose = FALSE, n.cores = NULL)  
## A gradient boosted model with gaussian loss function.  
## 10000 iterations were performed.  
## The best cross-validation iteration was 10000.  
## There were 80 predictors of which 45 had non-zero influence.

The output object is a list containing several modeling 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))  
## [1] 29551.99  
## [1] 29133.33  
  
# plot loss function as a result of n trees added to the ensemble  
gbm.perf(gbm.fit, method = "cv")



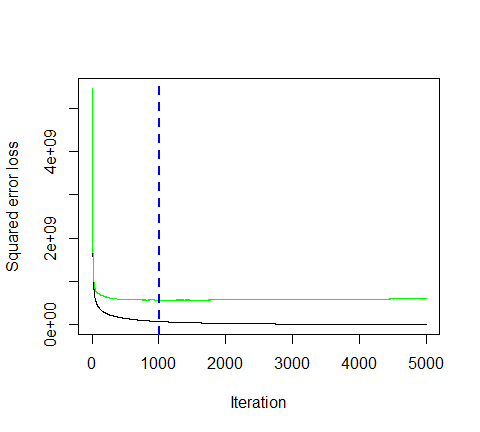
## [1] 9998

In this case, the small learning rate is resulting in very small incremental improvements which means many trees are required. In fact, for the default learning rate and tree depth settings it takes 39,906 trees for the CV error to minimize (~ 5 minutes of run time)!

#### Tuning

However, rarely do the degfault settings suffice. We could tune parameters one at a time to see how the results change. for example, here, I increase the learning rate to take large steps down the gradient descent, reduce the number of trees (since we are reducing the learning rate), and increase the depth of each tree from using a single split to 3 splits. This model takes about 90 seconds to run and achieves a significantly lower RMSE than our initial model with only 1,260 trees.

# for reproducibility  
set.seed(123)  
  
# train GBM model  
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, # will use all cores by default  
 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])  
## [1] 23852.61  
## [1] 23112.1  
  
# plot loss function as a result of n trees added to the ensemble  
gbm.perf(gbm.fit2, method = "cv")



## [1] 1003

However, a better option than manually tweaking hyperparameters one at a time is to perform a grid search which iterates over every combination of hyperparameter values and allows us to assess which combination tends to perform well. To perform a manual grid search, first we want to construct our grid of hyperparameter combinations. We’re going to search across 81 models with varying learning rates and tree depth. I also vary the minimum number of observations allowed in the trees terminal nodes (n.minobsinnode) and introduce stochastic gradient descent by allowing bag.fraction < 1.

# create hyperparameter grid  
  
hyper\_grid <- expand.grid(  
 shrinkage = c(0.01, .1, .3),  
 interaction.depth = c(1,3,5),  
 n.minosinnode = c(5,10,15),  
 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)  
## [1] 81

We loop through each hyperparameter combination and apply 5,000 trees. However, 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 remaning 25%. **Important note**: when using train.fraction it will take the first Xx % of the data so its important to randomize your rows in case this is any logic behind the ordering of the data (i.e., ordered by neighborhood)

After about 30 minutes of training time our grid search ends and we see a few important results pop out. 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 ; 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,
* none of the top models used n.minobsinnode = 15; the smaller nodes may allow us to capture pockets of unique feature-price point instances,
* in a few instances we appear to use nearly all 5,000 trees; maybe we should increase this parameter in our next search?

# 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)){  
 # reproducibility  
 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$n.minosinnode[i],  
 bag.fraction = hyper\_grid$bag.fraction[i],  
 train.fraction = .75,  
 n.cores = NULL, # will use all cores by default,  
 verbose = FALSE  
 )  
   
 # add min training error and trees to grid  
 hyper\_grid$optimal\_trees[i] <- which.min(gbm.tune$valid.error)  
 hyper\_grid$min\_RMSE[i] <- sqrt(min(gbm.tune$valid.error))  
}  
  
hyper\_grid %>%   
 dplyr::arrange(min\_RMSE) %>%  
 head(10)  
## shrinkage interaction.depth n.minosinnode bag.fraction optimal\_trees  
## 1 0.10 5 5 0.80 583  
## 2 0.01 5 5 0.80 4969  
## 3 0.01 5 5 1.00 4844  
## 4 0.01 5 10 0.80 4669  
## 5 0.01 5 10 1.00 4417  
## 6 0.10 5 15 0.80 364  
## 7 0.01 5 10 0.65 5000  
## 8 0.01 5 5 0.65 5000  
## 9 0.10 5 10 0.80 1417  
## 10 0.01 5 15 0.80 5000  
## min\_RMSE  
## 1 20916.08  
## 2 21341.28  
## 3 21402.16  
## 4 21418.73  
## 5 21496.43  
## 6 21527.81  
## 7 21569.37  
## 8 21599.52  
## 9 21652.69  
## 10 21694.30

These results help us to zoom into areas where we can refine our search. Let’s adjust our grid and zoom into closer regions of the values that appear to produce the best results in our previous grid search. This grid contains 81 combinations that we’ll search across.

# 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)  
## [1] 81  
## [1] 81

We can use the same for loop as before and perform our grid search. We get pretty similar results as before and, actually, our best model is the same as the best model above with an RMSE just above $20K.

# grid search  
  
for (i in 1:nrow(hyper\_grid)){  
   
 # reproducibility  
 set.seed(123)  
   
 # train model  
 gbm.tune <- gbm(  
 formula = Sale\_Price ~.,  
 distribution = "gaussian",  
 data = random\_ames\_train,  
 n.trees = 6000,  
 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],  
 train.fraction = 0.75,  
 n.cores = NULL, # will use all cores by default,  
 verbose = FALSE  
 )  
   
 # add min trainining error and trees to grid  
 hyper\_grid$optimal\_trees[i] <- which.min(gbm.tune$valid.error)  
 hyper\_grid$min\_RMSE[i] <- sqrt(min(gbm.tune$valid.error))  
 }  
  
hyper\_grid %>%   
 dplyr::arrange(min\_RMSE) %>%   
 head(10)  
## shrinkage interaction.depth n.minobsinnode bag.fraction optimal\_trees  
## 1 0.05 5 7 0.65 1088  
## 2 0.10 5 5 0.80 583  
## 3 0.05 7 7 0.65 1624  
## 4 0.01 7 5 1.00 5727  
## 5 0.05 5 5 0.65 5247  
## 6 0.05 7 7 1.00 1160  
## 7 0.05 7 5 1.00 1196  
## 8 0.01 7 5 0.80 5979  
## 9 0.01 7 7 0.80 5409  
## 10 0.05 7 5 0.80 2432  
## min\_RMSE  
## 1 20818.68  
## 2 20916.08  
## 3 21007.13  
## 4 21042.62  
## 5 21047.53  
## 6 21100.00  
## 7 21140.22  
## 8 21164.36  
## 9 21174.42  
## 10 21233.68  
## n.trees shrinkage interaction.depth n.minobsinnode bag.fraction optimal\_trees min\_RMSE  
## 1 6000 0.10 5 5 0.65 483 20407.76  
## 2 6000 0.01 5 7 0.65 4999 20598.62  
## 3 6000 0.01 5 5 0.65 4644 20608.75  
## 4 6000 0.05 5 7 0.80 1420 20614.77  
## 5 6000 0.01 7 7 0.65 4977 20762.26  
## 6 6000 0.10 3 10 0.80 1076 20822.23  
## 7 6000 0.01 7 10 0.80 4995 20830.03  
## 8 6000 0.01 7 5 0.80 4636 20830.18  
## 9 6000 0.10 3 7 0.80 949 20839.92  
## 10 6000 0.01 5 10 0.65 4980 20840.43

Once we have found our top model we train a model with those specific parameters. And since the model coveraged at 483 trees I train a cross validated model (to provide a more robust errro estimate) with 1000 trees. The cross-validated error of ~$22K is a better representation of the error we might expect on a new unseen data set.

# for reproducibility  
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  
 )

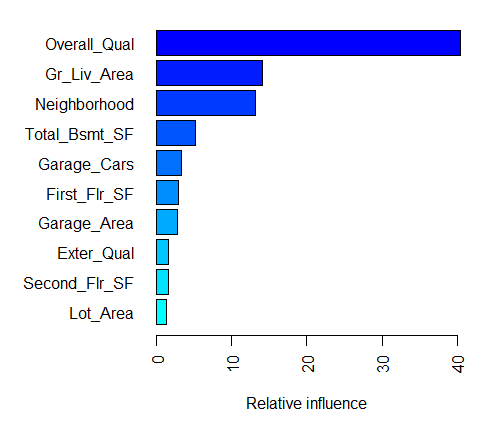
#### Visualizing

*variable importance*

After re-running our final model we likely want to understand the variables that have the largest influence on sale price. The summary method for gbm will output a data frame and a plot that shows the most influential variables. cBars allows you to adjust the number of variables to show (in order of influence). The default method for computing variable importance is with relative influence

1. method=relative.influence: At each split in each tree, gbm computes the improvement in the split-criterion (MSE for regression). gbm then averages the improvement made by each variable across all the trees that the variable is used. The variables with the largest average decrease in MSE are considered most important.
2. method = permutation.test.gbm: For each tree, the OOB sample is passed down the tree and the prediction accuracy is recorded. Then the values for each variable (one at a time) are randomly permuted and the accuracy is again computed. The decrease in accuracy as a result of this randomly “shaking up” of variable values is averaged over all the trees for each variable. The variables with the largest average decrease in accuracy are considered most important.

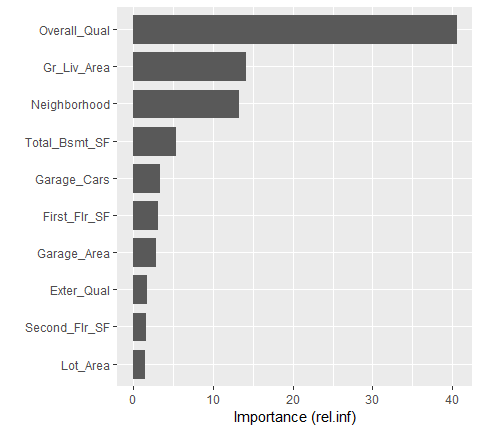
par(mar = c(5, 8, 1, 1))  
summary(  
 gbm.fit.final,   
 cBars = 10,  
 method = relative.influence, # also can use permutation.test.gbm  
 las = 2  
 )



## var rel.inf  
## Overall\_Qual Overall\_Qual 4.048861e+01  
## Gr\_Liv\_Area Gr\_Liv\_Area 1.402976e+01  
## Neighborhood Neighborhood 1.320178e+01  
## Total\_Bsmt\_SF Total\_Bsmt\_SF 5.227608e+00  
## Garage\_Cars Garage\_Cars 3.296480e+00  
## First\_Flr\_SF First\_Flr\_SF 2.927688e+00  
## Garage\_Area Garage\_Area 2.779194e+00  
## Exter\_Qual Exter\_Qual 1.618451e+00  
## Second\_Flr\_SF Second\_Flr\_SF 1.517907e+00  
## Lot\_Area Lot\_Area 1.296325e+00  
## Bsmt\_Qual Bsmt\_Qual 1.106896e+00  
## MS\_SubClass MS\_SubClass 9.788950e-01  
## Bsmt\_Unf\_SF Bsmt\_Unf\_SF 8.313946e-01  
## Year\_Remod\_Add Year\_Remod\_Add 7.381023e-01  
## BsmtFin\_Type\_1 BsmtFin\_Type\_1 6.967389e-01  
## Overall\_Cond Overall\_Cond 6.939359e-01  
## Kitchen\_Qual Kitchen\_Qual 6.649051e-01  
## Garage\_Type Garage\_Type 5.727906e-01  
## Fireplace\_Qu Fireplace\_Qu 5.281111e-01  
## Bsmt\_Exposure Bsmt\_Exposure 4.601763e-01  
## Sale\_Type Sale\_Type 4.063181e-01  
## Exterior\_1st Exterior\_1st 3.889487e-01  
## Open\_Porch\_SF Open\_Porch\_SF 3.797533e-01  
## Sale\_Condition Sale\_Condition 3.771138e-01  
## Exterior\_2nd Exterior\_2nd 3.482380e-01  
## Year\_Built Year\_Built 3.289944e-01  
## Full\_Bath Full\_Bath 3.070894e-01  
## Screen\_Porch Screen\_Porch 2.907584e-01  
## Fireplaces Fireplaces 2.906831e-01  
## Bsmt\_Full\_Bath Bsmt\_Full\_Bath 2.890666e-01  
## Wood\_Deck\_SF Wood\_Deck\_SF 2.448412e-01  
## Longitude Longitude 2.249308e-01  
## Central\_Air Central\_Air 2.232338e-01  
## Condition\_1 Condition\_1 1.867403e-01  
## Latitude Latitude 1.650208e-01  
## Lot\_Frontage Lot\_Frontage 1.488962e-01  
## Heating\_QC Heating\_QC 1.456177e-01  
## Mo\_Sold Mo\_Sold 1.406729e-01  
## TotRms\_AbvGrd TotRms\_AbvGrd 1.162828e-01  
## Functional Functional 1.118959e-01  
## Mas\_Vnr\_Area Mas\_Vnr\_Area 1.005664e-01  
## Paved\_Drive Paved\_Drive 9.346526e-02  
## Land\_Contour Land\_Contour 9.026977e-02  
## Enclosed\_Porch Enclosed\_Porch 7.512751e-02  
## Garage\_Cond Garage\_Cond 7.143649e-02  
## Lot\_Config Lot\_Config 6.355982e-02  
## Year\_Sold Year\_Sold 6.198938e-02  
## Bedroom\_AbvGr Bedroom\_AbvGr 6.146463e-02  
## Mas\_Vnr\_Type Mas\_Vnr\_Type 5.820946e-02  
## Roof\_Style Roof\_Style 5.682478e-02  
## Roof\_Matl Roof\_Matl 4.948087e-02  
## Alley Alley 4.279173e-02  
## BsmtFin\_Type\_2 BsmtFin\_Type\_2 3.482743e-02  
## Low\_Qual\_Fin\_SF Low\_Qual\_Fin\_SF 3.427532e-02  
## Garage\_Qual Garage\_Qual 3.136598e-02  
## Foundation Foundation 2.803887e-02  
## Bsmt\_Cond Bsmt\_Cond 2.768063e-02  
## Condition\_2 Condition\_2 2.751263e-02  
## Bsmt\_Half\_Bath Bsmt\_Half\_Bath 2.650291e-02  
## MS\_Zoning MS\_Zoning 2.346874e-02  
## Pool\_QC Pool\_QC 2.211997e-02  
## Half\_Bath Half\_Bath 2.148830e-02  
## Three\_season\_porch Three\_season\_porch 1.833322e-02  
## BsmtFin\_SF\_2 BsmtFin\_SF\_2 1.774457e-02  
## Garage\_Finish Garage\_Finish 1.407325e-02  
## Fence Fence 1.188101e-02  
## House\_Style House\_Style 1.105981e-02  
## Exter\_Cond Exter\_Cond 1.061516e-02  
## Misc\_Val Misc\_Val 7.996187e-03  
## Street Street 7.460435e-03  
## Land\_Slope Land\_Slope 6.847183e-03  
## Lot\_Shape Lot\_Shape 6.681489e-03  
## Electrical Electrical 5.498046e-03  
## BsmtFin\_SF\_1 BsmtFin\_SF\_1 4.149129e-03  
## Pool\_Area Pool\_Area 3.622225e-03  
## Misc\_Feature Misc\_Feature 7.232842e-04  
## Utilities Utilities 0.000000e+00  
## Bldg\_Type Bldg\_Type 0.000000e+00  
## Heating Heating 0.000000e+00  
## Kitchen\_AbvGr Kitchen\_AbvGr 0.000000e+00

An alternative approach is to use the underdevelopment vip package, which provides ggplot2 plots. vip also provides an additional measure of variable importance based on partial dependence measures and is a common variable importance plotting framework for many machine learning models.

# devtools::install\_github("koalaverse/vip")  
vip::vip(gbm.fit.final)

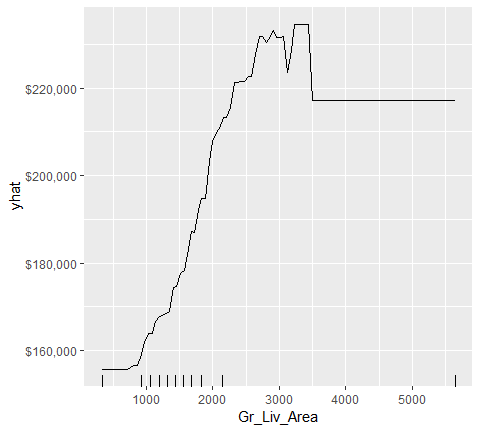


*Partial dependence plots*

After the most relevant variables have been identified the next step is to attemp to understand how the response variable changes based on these variables. For this, we can use **partial dependence plots (PDPs)** and **individual conditional expectation (ICE) curves**.

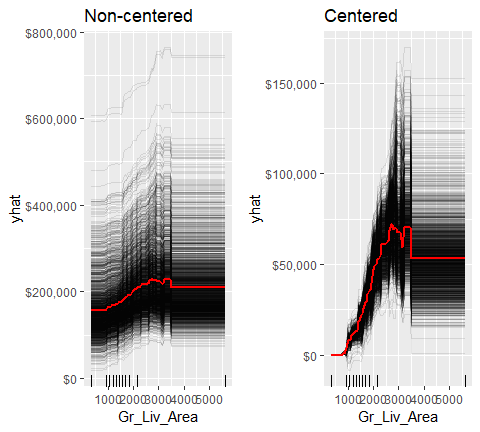
PDPs plot the change in the average predicted values as specified feature(s) vary over their marginal distribution. for example, consider the Gr\_Liv\_Area variable. The PDP plot below displays the average change in predicted sales price as we vary Gr\_Liv\_Area while holding all other variables constant. This is done by holding all variables constant for each observation in our training data set but then apply the unique values of Gr\_Liv\_Area for each observation. 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.

gbm.fit.final %>%   
 partial(pred.var = "Gr\_Liv\_Area",   
 n.trees = gbm.fit.final$n.trees,  
 grid.resolution = 100) %>%   
 autoplot(rug = TRUE, train=ames\_train)+  
 scale\_y\_continuous(labels=scales::dollar)



ICE curves are an extension of PDP plots but, rather than plot the average marginal effect on the response variable, we plot the change in the predicted response variable for each observation as we vary each predictor variable. Below shows the regular ICE curve plot (left) and the centered ICE curves (right). 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 resuts show that most observations follow a common trend as Gr\_Liv\_Area increases; however, 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 = TRUE) +  
 ggtitle("Centered") +  
 scale\_y\_continuous(labels = scales::dollar)  
  
gridExtra::grid.arrange(ice1, ice2, nrow=1)



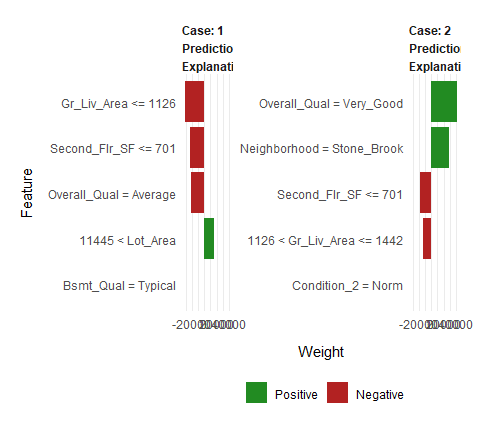
*LIME*

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](http://uc-r.github.io/lime). 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, Case 2:161K),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 interpreation 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 to our best gbm model.

# predict values for test set  
pred <- predict(gbm.fit.final, n.trees = gbm.fit.final$n.trees, ames\_test)  
  
# results  
caret::RMSE(pred, ames\_test$Sale\_Price)  
## [1] 21533.65  
## [1] 20681.88

### xgboost

The xgboost R package proviedes an R API to “Extreme Gradient Boosting”, which is an efficient implementation of gradient boosting framework (approxmately faster than gbm). The [xgboost/demo](https://github.com/dmlc/xgboost/tree/master/demo) repository provides a wealth of information. You can also find a fairly comprehensive parameter tuning guide [here](https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/). The xgboost package has been quite popular and successful on Kaggle for data mining competitions.

Features include:

* Provide 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; consequentl, 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 :

* <https://arxiv.org/abs/1611.09477>
* <https://www.r-bloggers.com/a-demonstration-of-vtreat-data-preparation/>
* <https://github.com/WinVector/vtreat>

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, verbose = FALSE)  
  
# Get the "clean" variable naems 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, varRestriction = new\_vars) %>% as.matrix()  
response\_train <- ames\_train$Sale\_Price  
  
# Prepare the test data  
features\_test <- vtreat::prepare(treatplan, ames\_test, varRestriction = new\_vars) %>% as.matrix()  
response\_test <- ames\_test$Sale\_Price  
  
# dimensions of one-hot encoded data  
dim(features\_train)  
## [1] 2051 343  
## [1] 2051 208  
dim(features\_test)  
## [1] 879 343  
## [1] 879 208

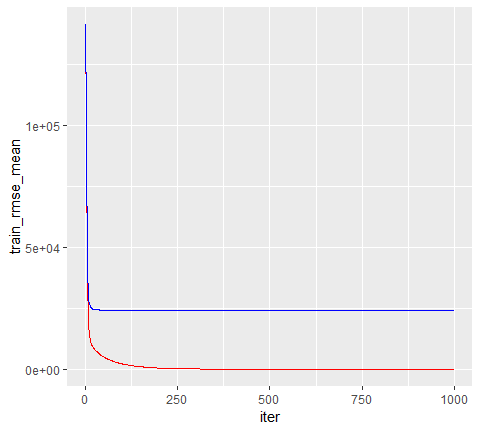
xgboost provides different training functions (i.e. xgb.train which is just a wrapper for xgboost). However, to train an XGBoost we typically want to use xgb.cv, which incorporates cross-validation. The following trains a basic 5-fold cross validated XGBoost model with 1,000 trees. There are many parameters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

* learning date(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,  
 objective = "reg:linear", # for regression models  
 verbose = 0 # silent,  
)

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 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))[1],  
 rmse.train = min(train\_rmse\_mean),  
 ntrees.test = which(test\_rmse\_mean == min(test\_rmse\_mean))[1],  
 rmse.test = min(test\_rmse\_mean)  
 )  
## ntrees.train rmse.train ntrees.test rmse.test  
## 1 896 0.0548992 123 24032.73  
## ntrees.train rmse.train ntrees.test rmse.test  
## 1 965 0.5022836 60 27572.31  
  
# plot error vs number trees  
ggplot(xgb.fit1$evaluation\_log) +  
 geom\_line(aes(iter, train\_rmse\_mean), color = "red") +  
 geom\_line(aes(iter, test\_rmse\_mean), color = "blue")



#### Tuning

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

# 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  
 verbose = 0, # silent,  
 early\_stopping\_rounds = 10 # stop if no improvement for 10 consecutive trees  
)  
# assess results  
xgb.fit3$evaluation\_log %>%  
 dplyr::summarise(  
 ntrees.train = which(train\_rmse\_mean == min(train\_rmse\_mean))[1],  
 rmse.train = min(train\_rmse\_mean),  
 ntrees.test = which(test\_rmse\_mean == min(test\_rmse\_mean))[1],  
 rmse.test = min(test\_rmse\_mean)  
 )  
## ntrees.train rmse.train ntrees.test rmse.test  
## 1 179 6042.873 169 22596.93  
## ntrees.train rmse.train ntrees.test rmse.test  
## 1 180 5891.703 170 24650.17

To perform a large search grip, 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 hyperparmeter 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),  
 optional\_trees = 0, # a place to dump results  
 min\_RMSE = 0 # a place to dump results  
)  
nrow(hyper\_grid)  
## [1] 576  
# [1] 576

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 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 is the full seach grid taking 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)  
   
 # train omdel  
 xgb.tune <- xgb.cv(  
 params = params,  
 data = features\_train,   
 label = response\_train,  
 nrounds = 5000,  
 nfold = 5,  
 objective = "reg:linear", # for regression model  
 verbose = 0, # silent  
 early\_stopping\_rounds = 10 # stop if no improvement for 10 consecutive trees  
 )  
   
 # add min training error and trees to grid  
 hyper\_grid$optimal\_trees[i] <- which.min(xgb.tune$evaluation\_log$test\_rmse\_mean)  
 hyper\_grid$min\_RMSE[i] <- min(xgb.tune$evaluation\_log$test\_rmse\_mean)  
}  
hyper\_grid %>%   
 dplyr::arrange(min\_RMSE) %>%   
 head(10)  
## eta max\_depth min\_child\_weight subsample colsample\_bytree optimal\_trees min\_RMSE  
## 1 0.01 5 5 0.65 1 1576 23548.84  
## 2 0.01 5 3 0.80 1 1626 23587.16  
## 3 0.01 5 3 0.65 1 1451 23602.96  
## 4 0.01 5 1 0.65 1 1480 23608.65  
## 5 0.05 5 3 0.65 1 305 23743.54  
## 6 0.01 5 1 0.80 1 1851 23772.90  
## 7 0.05 3 3 0.65 1 552 23783.55  
## 8 0.01 7 5 0.65 1 1248 23792.65  
## 9 0.01 3 3 0.80 1 1923 23794.78  
## 10 0.01 7 1 0.65 1 1070 23800.80

After assessing the results you would like to 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](https://www.analyticsvidhya.com/blog/2016/03/complete-guide-parameter-tuning-xgboost-with-codes-python/) to a greate blo post that discusses a strategic approach to tuning with xgboost. However, for brevity, we will just assume the top model in the above search is the global optimal model. Once you have 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,  
 data = features\_train,  
 label = response\_train,  
 nrounds = 1576,  
 objective = "reg:linear",  
 verbose = 0  
)

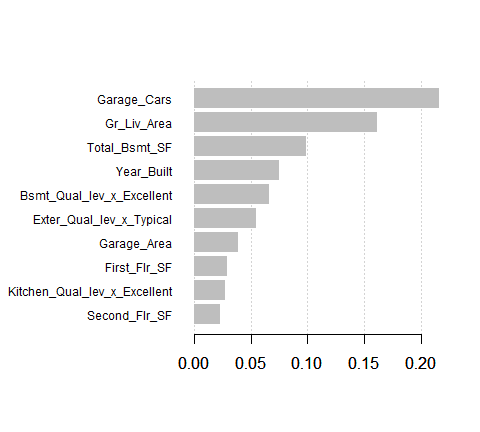
#### Visualizing

**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 measures:

* 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 tree1, tree2 and tree3; then the weightage for feature1 will be 2+1+3 = 6. The frequency for feature1 is calculated as its percentage weight over weights of all features.

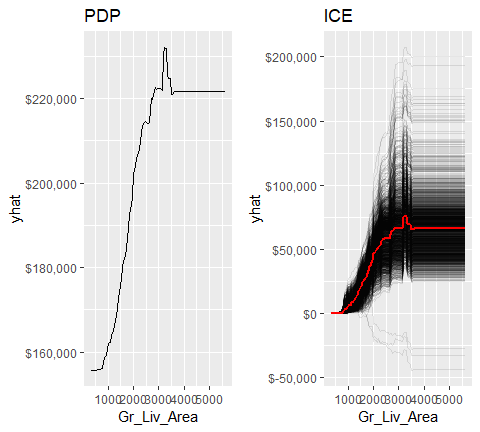
# create importance matrix  
importance\_matrix <- xgb.importance(  
 model = xgb.fit.final)  
# variable importance plot  
xgb.plot.importance(importance\_matrix, top\_n=10, measure = "Gain")



**Partial dependence plots**

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

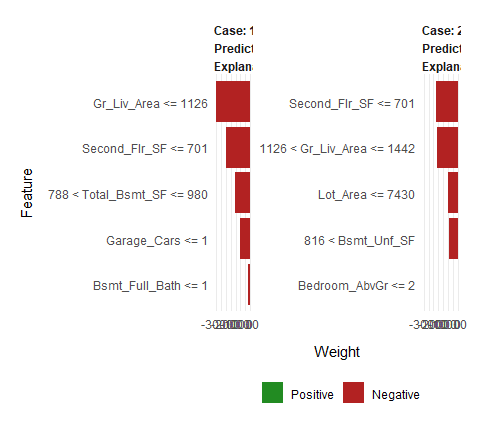
pdp <- xgb.fit.final %>%   
 partial(pred.var = "Gr\_Liv\_Area",   
 n.trees = 1576,   
 grid.resolution = 100,   
 train = features\_train) %>%   
 autoplot(rug = TRUE, train = features\_train)+  
 scale\_y\_continuous(labels = scales::dollar)+  
 ggtitle("PDP")  
  
ice <- xgb.fit.final %>%   
 partial(pred.var = "Gr\_Liv\_Area",  
 n.trees = 1576,   
 grid.resolution = 100,  
 train = features\_train,  
 ice = TRUE) %>%   
 autoplot(rug = TRUE,   
 train = features\_train,  
 alpha = .1,  
 center = TRUE)+  
 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 ne one-hot encoded in the same manner, as we prepared the raining 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,   
 varRestriction = new\_vars)  
# install.packages("vtreat")  
# apply LIME  
explainer <- lime(data.frame(features\_train), xgb.fit.final)  
explanation <- explain(local\_obs\_onehot, explainer, n\_features = 5)  
plot\_features(explanation)



**Predicting**

Lastly, we use predict to predict on new observations. owever, 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] 21898

### h2o

The h2o package is a powerful and efficient java-based interface that allows for local and cluster-based deployment. It comes with fairly comprehensive [online resource](http://docs.h2o.ai/h2o/latest-stable/h2o-docs/index.html) that includes methodology and code documentation along with tutorial.

Features include:

* Distributed and parallelized computation on either a single node or a multi-node cluster.
* Automatic early stopping based on convergence of user-specified metrics to user-specified relative tolerance.
* Stochastic GBM with column and row sampling (per split and per tree) for better generalization.
* Support for exponential families (Poisson, Gamma, Tweedie) and loss functions in addition to binomial (Bernoulli), Gaussian and multinomial distributions, such as Quantile regression (including Laplace).
* Grid search for hyperparameter optimization and model selection.
* Data-distributed, which means the entire dataset does not need to fit into memory on a single node, hence scales to any size training set.
* Uses histogram approximations of continuous variables for speedup.
* Uses dynamic binning - bin limits are reset at each tree level based on the split bins’ min and max values discovered during the last pass.
* Uses squared error to determine optimal splits.
* Distributed implementation details outlined in a blog post by Cliff Click.
* Unlimited factor levels.
* Multiclass trees (one for each class) built in parallel with each other.
* Apache 2.0 Licensed.
* Model export in plain Java code for deployment in production environments.

#### Basic implementation

Lets go ahead and start up h2o:

h2o.no\_progress()  
h2o.init(max\_mem\_size = "5g")  
##   
## H2O is not running yet, starting it now...  
##   
## Note: In case of errors look at the following log files:  
## C:\Users\KOJIKM~1.MIZ\AppData\Local\Temp\RtmpYnv4Bd/h2o\_KojiKM\_Mizumura\_started\_from\_r.out  
## C:\Users\KOJIKM~1.MIZ\AppData\Local\Temp\RtmpYnv4Bd/h2o\_KojiKM\_Mizumura\_started\_from\_r.err  
##   
##   
## Starting H2O JVM and connecting: . Connection successful!  
##   
## R is connected to the H2O cluster:   
## H2O cluster uptime: 3 seconds 742 milliseconds   
## H2O cluster timezone: Asia/Tokyo   
## H2O data parsing timezone: UTC   
## H2O cluster version: 3.22.1.1   
## H2O cluster version age: 4 months and 10 days !!!   
## H2O cluster name: H2O\_started\_from\_R\_KojiKM.Mizumura\_hsp331   
## H2O cluster total nodes: 1   
## H2O cluster total memory: 5.00 GB   
## H2O cluster total cores: 4   
## H2O cluster allowed cores: 4   
## H2O cluster healthy: TRUE   
## H2O Connection ip: localhost   
## H2O Connection port: 54321   
## H2O Connection proxy: NA   
## H2O Internal Security: FALSE   
## H2O API Extensions: Algos, AutoML, Core V3, Core V4   
## R Version: R version 3.6.0 (2019-04-26)

h2o.gbm allows us to perform a GBM with H20. However, prior to running our initial model, we need to convert our training datato an h2o object. By default, h2o.gbm applies a GBM model with the following parameters:

* number of trees (ntrees): 50
* learning rate (learn\_rate): 0.1
* tree depth(max\_depth): 5
* minimum observations in a terminal node(min\_rows):10
* no sampling of observations or columns

# create feature names  
y <- "Sale\_Price"  
x <- setdiff(names(ames\_train), y)  
  
# turn training set into h2o object  
train.h2o <- as.h2o(ames\_train)  
  
# training basic GBM model with defaults  
h2o.fit1 <- h2o.gbm(  
 x = x,  
 y = y,  
 training\_frame = train.h2o,  
 nfolds = 5  
)  
  
# assess the model results  
h2o.fit1  
## Model Details:  
## ==============  
##   
## H2ORegressionModel: gbm  
## Model ID: GBM\_model\_R\_1557373144350\_1   
## Model Summary:   
## number\_of\_trees number\_of\_internal\_trees model\_size\_in\_bytes min\_depth  
## 1 50 50 18109 5  
## max\_depth mean\_depth min\_leaves max\_leaves mean\_leaves  
## 1 5 5.00000 12 32 24.12000  
##   
##   
## H2ORegressionMetrics: gbm  
## \*\* Reported on training data. \*\*  
##   
## MSE: 155858084  
## RMSE: 12484.31  
## MAE: 8822.822  
## RMSLE: 0.07910112  
## Mean Residual Deviance : 155858084  
##   
##   
##   
## H2ORegressionMetrics: gbm  
## \*\* Reported on cross-validation data. \*\*  
## \*\* 5-fold cross-validation on training data (Metrics computed for combined holdout predictions) \*\*  
##   
## MSE: 631238872  
## RMSE: 25124.47  
## MAE: 15220.24  
## RMSLE: 0.1332259  
## Mean Residual Deviance : 631238872  
##   
##   
## Cross-Validation Metrics Summary:   
## mean sd cv\_1\_valid cv\_2\_valid  
## mae 15197.057 734.7227 15101.938 14173.995  
## mean\_residual\_deviance 6.2750202E8 9.0961888E7 5.65776E8 5.8381907E8  
## mse 6.2750202E8 9.0961888E7 5.65776E8 5.8381907E8  
## r2 0.89940906 0.011050546 0.9047268 0.89797634  
## residual\_deviance 6.2750202E8 9.0961888E7 5.65776E8 5.8381907E8  
## rmse 24922.357 1785.7914 23786.047 24162.348  
## rmsle 0.13232477 0.01019797 0.118831456 0.1514387  
## cv\_3\_valid cv\_4\_valid cv\_5\_valid  
## mae 16049.667 16667.8 13991.884  
## mean\_residual\_deviance 6.7806176E8 8.459264E8 4.63926848E8  
## mse 6.7806176E8 8.459264E8 4.63926848E8  
## r2 0.89385164 0.87617725 0.9243131  
## residual\_deviance 6.7806176E8 8.459264E8 4.63926848E8  
## rmse 26039.62 29084.814 21538.96  
## rmsle 0.14580649 0.13078508 0.11476211

Similar to XGBoost, we can incorporate automated stopping so that we can crank up the number of trees but terminate training once model improvement decreases or stops. There is also an option to terminate training after so much time has passed (see max\_runtime\_secs). In this example, I train a default model with 5,000 trees but stop training after 10 consecutive trees have no improvement on the cross-validated error. In this case, training stops after 3828 trees and has a cross validated RMSE of 24,684.

# training basic GBM model with defaults  
h2o.fit2 <- h2o.gbm(  
 x = x,  
 y = y,  
 training\_frame = train.h2o,  
 nfolds = 5,  
 ntrees = 5000,  
 stopping\_rounds = 10,  
 stopping\_tolerance = 0,  
 seed = 123  
)  
  
# model stopped after xx trees  
h2o.fit2@parameters$ntrees  
## [1] 2482  
## [1] 3828  
  
# cross validated RMSE  
h2o.rmse(h2o.fit2, xval = TRUE)  
## [1] 24626.19  
## [1] 24684.09

#### Tuning

H2o provides many parameters that can be adjusted. It is well worth your time to check out the available documentation at [H2O.ai](http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/gbm.html#gbm-tuning-guide). For this tutorial, we will focus on the more common hyperparametes that I typically apply. This includes:

* tree complexity:
  + ntrees: number of trees to train
  + max\_depth: depth of each tree
  + min\_rows: Fewest observations allowed in a terminal node
* Learning rate:
  + learn\_rate rate to descend the loss function gradient
  + learn\_rate\_annealing: allow you to have a high initial learn\_rate, then gradually reduce as trees are added (speeds up training)
* Adding stochastic nature:
  + sample rate: row sample rate per tree
  + col\_sample\_rate: columns sample rate per tree (synonymous with xgboost’s colsample\_bytree).

Note that there are parameters that control how categorical and continuous variables are encoded, binned and split. The defaults tend to perform quite well, but I have been able to gain small improvements in certain circmustances by adjusting these. I will not cover them but the are work reviewing.

To perform grid search tuning with H2O we have two options: perform a *full* or *random discrete grid search*.

**Full grid search**

A **full cartesian grid search** examines every combination of hyperparameter settings that we specify in a tuning grid. This has been the type of tuning we have been performing with our manual for loops with gbm and xgboost. However, to speed up training with H2O I’ll use a validation set rather than perform k-fold cross validation. The following creates a hyperparameter grid consisting of 486 hyperparameter combinations. We apply h2o.grid to perform a grid search while also incorporating stopping parameters to reduce training time. Total grid search time was about 90 minutes.

A few characteristics pop out when we assess the results - models with trees deeper than one split with a low learning rate, no annealing, and stochastic observation sampling tend to perform best.

# create training & validation sets  
split <- h2o.splitFrame(train.h2o,   
 ratios = 0.75)  
train <- split[[1]]  
valid <- split[[2]]  
  
# create hyperparameter grid  
hyper\_grid <- list(  
 max\_depth = c(1, 3, 5),  
 min\_rows = c(1, 5, 10),  
 learn\_rate = c(0.01, 0.05, 0.1),  
 learn\_rate\_annealing = c(.99, 1),  
 sample\_rate = c(.5, .75, 1),  
 col\_sample\_rate = c(.8, .9, 1)  
)  
  
# perform grid search  
grid <- h2o.grid(  
 algorithm = "gbm",  
 grid\_id = "gbm\_grid1",  
 x = x,  
 y = y,  
 training\_frame = train,  
 validation\_frame = valid,  
 hyper\_params = hyper\_grid,  
 ntrees = 5000,  
 stopping\_rounds = 10,  
 stopping\_tolerance = 0,  
 seed = 123  
)  
  
# collect the results and sort by our model performance metric of choice  
  
grid\_perf <- h2o.getGrid(  
 grid\_id = "gbm\_grid1",  
 sort\_by = "mse",  
 decreasing = FALSE  
)  
  
grid\_perf  
## H2O Grid Details  
## ================  
##   
## Grid ID: gbm\_grid1   
## Used hyper parameters:   
## - col\_sample\_rate   
## - learn\_rate   
## - learn\_rate\_annealing   
## - max\_depth   
## - min\_rows   
## - sample\_rate   
## Number of models: 486   
## Number of failed models: 0   
##   
## Hyper-Parameter Search Summary: ordered by increasing mse  
## col\_sample\_rate learn\_rate learn\_rate\_annealing max\_depth min\_rows sample\_rate model\_ids mse  
## 1 1.0 0.01 1.0 3 10.0 0.75 gbm\_grid1\_model\_299 3.6209830674536294E8  
## 2 0.8 0.01 1.0 3 10.0 0.75 gbm\_grid1\_model\_297 3.6380633209494674E8  
## 3 0.8 0.01 1.0 3 1.0 0.5 gbm\_grid1\_model\_27 3.6672773986842275E8  
## 4 0.8 0.01 1.0 5 1.0 0.5 gbm\_grid1\_model\_45 3.683498830618852E8  
## 5 0.9 0.01 1.0 3 10.0 0.75 gbm\_grid1\_model\_298 3.686060225554216E8  
##   
## ---  
## col\_sample\_rate learn\_rate learn\_rate\_annealing max\_depth min\_rows sample\_rate model\_ids mse  
## 481 0.9 0.01 0.99 1 10.0 1.0 gbm\_grid1\_model\_433 2.824716768094968E9  
## 482 0.9 0.01 0.99 1 1.0 1.0 gbm\_grid1\_model\_325 2.824716768094968E9  
## 483 0.9 0.01 0.99 1 5.0 1.0 gbm\_grid1\_model\_379 2.824716768094968E9  
## 484 1.0 0.01 0.99 1 5.0 1.0 gbm\_grid1\_model\_380 2.8252384874380198E9  
## 485 1.0 0.01 0.99 1 1.0 1.0 gbm\_grid1\_model\_326 2.8252384874380198E9  
## 486 1.0 0.01 0.99 1 10.0 1.0 gbm\_grid1\_model\_434 2.8252384874380198E9

We can check out more details of the best performing model. The top model achieves a validation RMSE of 19,029.

# Grab the model\_id for the top model, chosen by 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, valid = TRUE)  
## H2ORegressionMetrics: gbm  
## \*\* Reported on validation data. \*\*  
##   
## MSE: 362098307  
## RMSE: 19028.88  
## MAE: 12427.99  
## RMSLE: 0.1403692  
## Mean Residual Deviance : 362098307

#### Random discrete grid search

Because of the combinatiorial explosion, each additional hyperparameter that gets added to our grid search has a hue effect on the time to complete. Consequently, h2o provides an additional grid search path called “RandomDiscrete”, which will jump from one random combination to another and stop once a certain level of improvement has been made, certain amo unt of time has been exceeded, or certain amount of models have been ran (or a combination of these have been met). Although using a random discrete search path will likely not find the optimal model, it typically does a good job of finding a very good model.

The following performs a random discrete using the same hyperparameter grid we used above. However, in this example we add a search criteria (which is preferred when using a random search) that stops the grid search if none of the last 10 models have managed to have a 0.5% improvement in MSE compared to the best model before that. If we continue to find improvements then I cut the grid search off after 3600 seconds (60 minutes). In this example, our search went for the entire 60 minutes and evaluated 291 of the 486 potential models.

# random grid search criteria  
search\_criteria <- list(  
 strategy = "RandomDiscrete",  
 stopping\_metric = "mse",  
 stopping\_tolerance = 0.005,  
 stopping\_rounds = 10,  
 max\_runtimes\_secs = 60\*60  
)  
  
# perform grid search  
grid <- h2o.grid(  
 algorithm = "gbm",  
 grid\_id = "gbm\_grid2",  
 x = x,  
 y = y,  
 training\_frame = train,  
 validation\_frame = valid,  
 hyper\_params = hyper\_grid,  
 search\_criteria = search\_criteria, # add this  
 ntrees = 5000,  
 stopping\_rounds = 10,  
 stopping\_tolerance = 0,  
 seed = 123  
)  
  
# collect the result and sort by our model performance metric of choice  
  
grid\_perf <- h2o.getGrid(  
 grid\_id = "gbm\_grid2",  
 sort\_by = "mse",  
 decreasing = FALSE  
)  
  
grid\_perf  
## H2O Grid Details  
## ================  
##   
## Grid ID: gbm\_grid2   
## Used hyper parameters:   
## - col\_sample\_rate   
## - learn\_rate   
## - learn\_rate\_annealing   
## - max\_depth   
## - min\_rows   
## - sample\_rate   
## Number of models: 291   
## Number of failed models: 0   
##   
## Hyper-Parameter Search Summary: ordered by increasing mse  
## col\_sample\_rate learn\_rate learn\_rate\_annealing max\_depth min\_rows sample\_rate model\_ids mse  
## 1 0.8 0.05 1.0 3 10.0 1.0 gbm\_grid2\_model\_74 5.150720254988258E8  
## 2 0.9 0.01 1.0 3 5.0 0.5 gbm\_grid2\_model\_146 5.1889115659740096E8  
## 3 0.9 0.05 1.0 3 5.0 0.5 gbm\_grid2\_model\_114 5.2062049083883923E8  
## 4 0.8 0.05 1.0 3 5.0 0.75 gbm\_grid2\_model\_37 5.2124226584496534E8  
## 5 0.9 0.05 1.0 3 10.0 1.0 gbm\_grid2\_model\_157 5.212796449846914E8  
##   
## ---  
## col\_sample\_rate learn\_rate learn\_rate\_annealing max\_depth min\_rows sample\_rate model\_ids mse  
## 286 0.9 0.01 0.99 1 10.0 1.0 gbm\_grid2\_model\_179 3.323851889022955E9  
## 287 1.0 0.01 0.99 1 10.0 1.0 gbm\_grid2\_model\_260 3.3243159009633546E9  
## 288 0.9 0.01 0.99 1 5.0 0.5 gbm\_grid2\_model\_199 3.3243216930611935E9  
## 289 0.8 0.01 0.99 1 10.0 0.5 gbm\_grid2\_model\_80 3.3244630344508557E9  
## 290 0.8 0.01 0.99 1 1.0 0.5 gbm\_grid2\_model\_71 3.3244630344508557E9  
## 291 0.8 0.01 0.99 1 5.0 0.5 gbm\_grid2\_model\_227 3.3244630344508557E9

In this example, the best model obtained a cross-validated RMSE of 22,695. Not quite as good as the full grid search; however, often the results come much closer.

# Grab the model id for the top model, chosen by 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, valid = TRUE)  
## H2ORegressionMetrics: gbm  
## \*\* Reported on validation data. \*\*  
##   
## MSE: 515072025  
## RMSE: 22695.2  
## MAE: 13841.13  
## RMSLE: 0.1427291  
## Mean Residual Deviance : 515072025

Once we’ve found our preferred model, we’ll go ahead and retrain a new model with the full training data. I’ll use the best model from the full grid search and perform a 5-fold CV to get a robust estimate of the expected error.

# train final model  
  
h2o.final <- h2o.gbm(  
 x = x,  
 y = y,  
 training\_frame = train.h2o,  
 nfolds = 5,  
 ntrees = 10000,  
 learn\_rate = 0.01,  
 learn\_rate\_annealing = 1,  
 max\_depth = 3,  
 min\_rows = 10,  
 sample\_rate = 0.75,  
 col\_sample\_rate = 1,  
 stopping\_rounds = 10,  
 stopping\_tolerance = 0,  
 seed = 123  
)  
  
# model stopped after xx trees  
h2o.final@parameters$ntrees  
## [1] 9385  
  
# cross validated RMSE  
h2o.rmse(h2o.final, xval = TRUE)  
## [1] 23218.45

**Visualizing** **Variable importance**

h2o provides a built function that plots variable importance. It only has one measure of variable importance, relative importance, which measures the average impact each variable has across all the trees on the loss function. The variable with the largest is most importance and the impact of all other variables are provided relative to the most important variable. The vip package also works with h2o objects to plot variable importance.

h2o.varimp\_plot(h2o.final,  
 num\_of\_features = 10)  
  
# vip package usage - to be checked  
vip::vip(h2o.final)

**Partial dependence plots(PDP)**

We can also create similar PDP and ICE plots as before. We only need to incorporate a specialty function that converts the supplied data to an h2o object and then formats the predicted output as a data frame. We feed this into the partial function and the rest is standard.

pfun <- function(object, newdata) {  
 as.data.frame(predict(object, newdata = as.h2o(newdata)))[[1L]]  
}  
  
pdp <- h2o.final %>%  
 partial(  
 pred.var = "Gr\_Liv\_Area",   
 pred.fun = pfun,  
 grid.resolution = 20,   
 train = ames\_train  
 ) %>%  
 autoplot(rug = TRUE, train = ames\_train, alpha = .1) +  
 scale\_y\_continuous(labels = scales::dollar) +  
 ggtitle("PDP")  
  
ice <- h2o.final %>%  
 partial(  
 pred.var = "Gr\_Liv\_Area",   
 pred.fun = pfun,  
 grid.resolution = 20,   
 train = ames\_train,  
 ice = TRUE  
 ) %>%  
 autoplot(rug = TRUE, train = ames\_train, alpha = .1, center = TRUE) +  
 scale\_y\_continuous(labels = scales::dollar) +  
 ggtitle("ICE")  
  
gridExtra::grid.arrange(pdp, ice, nrow = 1)

h2o does not provide built-in ICE plots but it does provide a PDP plot that plots not only the mean marginal impact (as in a normal PDP) but also one standard error to show the variability.

h2o.partialPlot(h2o.final, data = train.h2o, cols = "Overall\_Qual")

Unfortunately, h2o’s function plots the categorical levels in alphabetical order whereas pdp will plot them in their specified level order making inference more intuitive.

pdp <- h2o.final %>%  
 partial(  
 pred.var = "Overall\_Qual",   
 pred.fun = pfun,  
 grid.resolution = 20,   
 train = as.data.frame(ames\_train)  
 ) %>%  
 autoplot(rug = TRUE, train = ames\_train, alpha = .1) +  
 scale\_y\_continuous(labels = scales::dollar) +  
 ggtitle("PDP")  
  
ice <- h2o.final %>%  
 partial(  
 pred.var = "Overall\_Qual",   
 pred.fun = pfun,  
 grid.resolution = 20,   
 train = as.data.frame(ames\_train),  
 ice = TRUE  
 ) %>%  
 autoplot(rug = TRUE, train = ames\_train, alpha = .1, center = TRUE) +  
 scale\_y\_continuous(labels = scales::dollar) +  
 ggtitle("ICE")  
  
gridExtra::grid.arrange(pdp, ice, nrow = 1)

**LIME**

LIME also provides built-in functionality for h2o objects (see ?model\_type).

# apply LIME  
explainer <- lime(ames\_train, h2.final)  
explanation <- explain(local\_obs, explainer, n\_features = 5)  
plot\_features(explanation)

#### Predicting

Lastly, we use h2o.predict or predict to predict on new observations and we can also evaluate the performance of our model on our test set easily with h2o.performance. Results are quite similar to both gbm and xgboost.

# convert test set to h2o object  
test.h2o <- as.h2o(ames\_test)  
  
# evaluate performance on new data  
h2o.performance(model = h2o.final, newdata = test.h2o)  
## H2ORegressionMetrics: gbm  
##   
## MSE: 407532539  
## RMSE: 20187.44  
## MAE: 12683.01  
## RMSLE: 0.100829  
## Mean Residual Deviance : 407532539  
  
# predict with h2o.predict  
h2o.predict(h2o.final, newdata = test.h2o)  
## predict  
## 1 130114.9  
## 2 162136.7  
## 3 263438.5  
## 4 484853.0  
## 5 219152.9  
## 6 208616.2  
##   
## [879 rows x 1 column]  
  
# predict values with predict  
predict(h2o.final, test.h2o)  
## predict  
## 1 130114.9  
## 2 162136.7  
## 3 263438.5  
## 4 484853.0  
## 5 219152.9  
## 6 208616.2  
##   
## [879 rows x 1 column]

### Learning more

GBMs are one of the most powerful ensemble algorithms that are often first-in-class with predictive accuracy. Although they are less intuitive and more computationally demanding than many other machine learning algorithms, they are essential to have in your toolbox. To learn more I would start with the following resources:

Traditional book resources:

An Introduction to Statistical Learning Applied Predictive Modeling Computer Age Statistical Inference The Elements of Statistical Learning Alternative online resources:

* [Trevor Hastie - Gradient Boosting & Random Forests at H2O World 2014](https://koalaverse.github.io/machine-learning-in-R/%20//www.youtube.com/watch?v=wPqtzj5VZus&index=16&list=PLNtMya54qvOFQhSZ4IKKXRbMkyL%20Mn0caa) (YouTube)
* [Trevor Hastie - Data Science of GBM (2013)](http://www.slideshare.net/0xdata/gbm-27891077) (slides)
* [Mark Landry - Gradient Boosting Method and Random Forest at H2O World 2015](https://www.youtube.com/watch?v=9wn1f-30_ZY) (YouTube)
* [Peter Prettenhofer - Gradient Boosted Regression Trees in scikit-learn at PyData London 2014](https://www.youtube.com/watch?v=IXZKgIsZRm0) (YouTube)
* [Alexey Natekin1 and Alois Knoll - Gradient boosting machines, a tutorial (blog post)](http://journal.frontiersin.org/article/10.3389/fnbot.2013.00021/full)

## Linear & Quadratic Discriminant Analysis

In the previous tutorial, you learned that logistic regression is a classification algorithm traditionally limited to only two-class classification problems (i.e., *default* = *Yes* or *No*). However, if you have more than two classes then linear (and its cousin Quadratic) Discriminant Analysis (LDA & QDA) is an often-preferred classification technique. Discriminant analysis models the distribution of the predictors X separately in each of the response classes (i.e., default = *“Yes”*, default = *“No”*), and then uses [Bayes’ theorem](https://en.wikipedia.org/wiki/Bayes'_theorem) to flip these around into estimates for the probability of the response category given the value of X.

### tl;dr

This tutorial covers an introduction to LDA & QDA and covers:

1. [Replication requirements](#DA_RR): What you’ll need to reproduce the analysis in this tutorial
2. [Why use discriminant analysis](#DA_Why): Understand why and when to use discriminant analysis and the basics behind how it works
3. [Preparing our data](#LDA_Data): Prepare our data for modeling
4. [Linear discriminant analysis](#DA_LDA): Modeling and 5. classifying the categorical response Y with a linear combination of predictor variables X.
5. [Quadratic discriminant analysis](#DA_QDA): Modeling and classifying the categorical response Y with a non-linear combination of predictor variables X.
6. [Prediction Performance](#DA_Pred): How well does the model fit the data? Which predictors are most important? Are the predictions accurate?
7. [A comparison](#DA_Comp): An example comparing logistic regression & Discriminant Analysis
8. [Additional resources](#DA_Resource): Additional resources to help you learn more

### Replication requirements

This tutorial primarily leverages the Default data provided by the ISLR package. This is a simulated data set containing information on ten thousand customers such as whether the customer defaulted, is a student, the average balance carried by the customer and the income of the customer. We will also use a few packages that provide data manipulation, visualization, pipeline modeling functions, and model outout tidying functions.

library(tidyverse) # data manipulation and visualization  
library(MASS) # provides LDA & QDA model functions  
  
# data set from ISLR package  
library(ISLR)  
  
# Load data   
(default <- as\_tibble(ISLR::Default))  
## # A tibble: 10,000 x 4  
## default student balance income  
## <fct> <fct> <dbl> <dbl>  
## 1 No No 730. 44362.  
## 2 No Yes 817. 12106.  
## 3 No No 1074. 31767.  
## 4 No No 529. 35704.  
## 5 No No 786. 38463.  
## 6 No Yes 920. 7492.  
## 7 No No 826. 24905.  
## 8 No Yes 809. 17600.  
## 9 No No 1161. 37469.  
## 10 No No 0 29275.  
## # ... with 9,990 more rows  
## # A tibble: 10,000 × 4  
## default student balance income  
## <fctr> <fctr> <dbl> <dbl>  
## 1 No No 729.5265 44361.625  
## 2 No Yes 817.1804 12106.135  
## 3 No No 1073.5492 31767.139  
## 4 No No 529.2506 35704.494  
## 5 No No 785.6559 38463.496  
## 6 No Yes 919.5885 7491.559  
## 7 No No 825.5133 24905.227  
## 8 No Yes 808.6675 17600.451  
## 9 No No 1161.0579 37468.529  
## 10 No No 0.0000 29275.268  
## # ... with 9,990 more rows

### Why use discriminant analysis

In the previous tutorial, we saw that a logistic regression model does a fairly good job classifying customers that default. So why do we need another classification method beyond logistic regression? There are several reasons:

* When the classes of the reponse variable Y (i.e. *default = “Yes”*, *default = “No”*) are well-separated, the parameter estimates for the logistic regression model are surprisingly unstable. LDA & QDA do not suffer from this problem.
* If *n* is small and the distribution of the predictors *X* is approximately normal in each of the classes, the LDA & QDA models are again more stable than the logistic reression model.
* LDA & QDA are often preferred over logistic regression when we have more than two non-ordinal response classes (i.e., *stroke*, *drug overdose* and *epileptic seizure*).

However, it is important to note that LDA & QDA have assumptions that are often more restictive than logistic regression:

* Both LDA and QDA assume the the predictor variables *X* are drawn from a multivariate Gaussian (aka *normal*) distribution.
* LDA assumes equality of covariances among the predictor variables X across each all levels of *Y*. This assumption is relaxed with **the QDA model**.
* LDA and QDA require the number of predictor variables (p) to be less then the sample size (n). Furthermore, its important to keep in mind that performance will severely decline as p approaches n. A simple rule of thumb is to use LDA & QDA on data sets where .

Also, when considering between LDA & QDA its important to know that **LDA** is a much **less flexible** classifier than QDA, and so has substantially *lower variance*. This can potentially lead to improved prediction performance. But there is a trade-off: if LDA’s assumption that the the predictor variable share a common variance across each Y response class is badly off, then LDA can suffer from *high bias*. Roughly speaking, LDA tends to be a better bet than QDA if there are relatively few training observations and so reducing variance is crucial. In contrast, QDA is recommended if the training set is very *large*, so that the variance of the classifier is not a major concern, or if the assumption of a common covariance matrix is clearly untenable.

### Preparing our data

As we’ve done in the previous tutorials, we’ll split our data into a training (60%) and testing (40%) data sets so we can assess how well our model performs on an out-of-sample data set.

set.seed(123)  
sample <- sample(c(TRUE, FALSE), nrow(default), replace = T, prob = c(0.6,0.4))  
train <- default[sample, ]  
test <- default[!sample, ]

### Linear discriminant analysis

LDA computes “discriminat scores” for each observation to classify what response variable class it is in (i.e., default or not default). These scores are obtained by finding linear combination of the independent variables. For a single predictor variable the LDA classifier is estimated as

where:

* is the estimated discriminant score that the observation will fall in the kth class within the response variable (i.e. default or not default) based on the value of the predictor variable x
* is the average of all the training observations from the kth class
* is the weighted average of the sample variances for each of the K classes
* is the prior probability that an observation belongs to the kth class (not to be confused with the mathematical constant ≈3.14159.

This clssifier assigns an observation to the kth class of for which discriminant score is largest. For example, lets assume there are two clases (A and B) for the response variable . Based on the predictor variable, LDA is going to compute the probability distribution of being classified as class or . The linear decision boundary between the probability distribution is represented by the dashed line. Discriminat scores to the left of the dashed line will be classified as and scores to the right will be classified as .

When dealing with more than one predictor variable, the LDA classifier assumes that the observations in the th class are drawn from a multivariate Gaussian distribution , where is class-specific mean vector, and is a covariance matrix that is common to all classes. Incorporate this into the LDA classifier results in:

,where an observation will be assigned to class where the discriminant score is largest.

#### Estimate & Understand Model

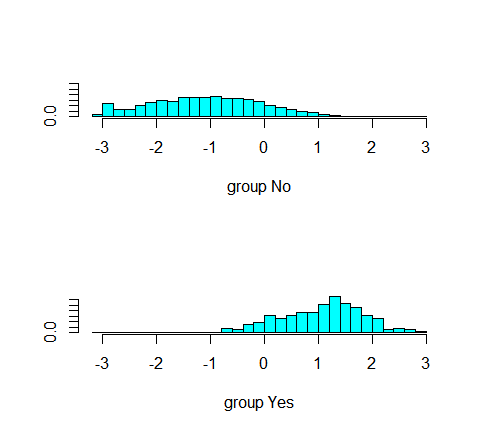
In R, we fit a LDA model using the lda function, which is part of the MASS library. Notice that the syntax for the lda is identical to that of lm (as seen in the linear regression tutorial), and to that of glm (as seen in the logistic regression tutorial) except for the absence of the family option.

(lda.m1 <- lda(default ~ balance + student, data = train))  
## Call:  
## lda(default ~ balance + student, data = train)  
##   
## Prior probabilities of groups:  
## No Yes   
## 0.9677526 0.0322474   
##   
## Group means:  
## balance studentYes  
## No 804.968 0.2956254  
## Yes 1776.971 0.3948718  
##   
## Coefficients of linear discriminants:  
## LD1  
## balance 0.002232368  
## studentYes -0.227922283

The LDA output indicates that our prior probabilities are and : in other words, 96.8% of the training observations are customers who did not default and 3% represent those that defaulted. It also provides the group means; these are the average of each predictor within each class, and are used by LDA as estimates of . These suggest that customers that tend to default have, on average, a credit card balance of $1,777 and are more likely to be students then non-defaulters (29% of non-defaulters are students whereas 39% of defaulters are). However, as we learned from the last tutorial this is largely because students tend to have higher balances then non-students.

The *coefficients of linear disciminants* provide he linear combination of balance and student=Yes that are used to form the LDA decision rule. In other words, these are the multipliers of the elements of X = x in Eq 1 & 2. If 0.0022 × balance − 0.228 × student is large, then the LDA classifier will predict that the customer will default, and if it is small, then the LDA classifier will predict the customer will not default. We can use plot to produce plots of the linear discriminants, obtained by computing 0.0022 × balance − 0.228 × student for each of the training observations. As you can see, when the probability increases that the customer will not default and when the probability increases that the customer will default.

plot(lda.m1)



#### Make predictions

We can use predict for LDA much like we did with logistic regression. I will illustrate the output that predict provides based on this simple data set.

train  
## # A tibble: 6,047 x 4  
## default student balance income  
## <fct> <fct> <dbl> <dbl>  
## 1 No No 730. 44362.  
## 2 No No 1074. 31767.  
## 3 No Yes 920. 7492.  
## 4 No No 826. 24905.  
## 5 No No 1161. 37469.  
## 6 No No 0 29275.  
## 7 No Yes 1221. 13269.  
## 8 No No 607. 44995.  
## 9 No No 1113. 23810.  
## 10 No No 0 50265.  
## # ... with 6,037 more rows  
  
df <- tibble(balance = rep(c(1000, 2000), 2),   
 student = c("No", "No", "Yes", "Yes"))  
df <- df %>%   
 mutate\_at(vars(student), as.factor)  
df  
## # A tibble: 4 x 2  
## balance student  
## <dbl> <fct>   
## 1 1000 No   
## 2 2000 No   
## 3 1000 Yes   
## 4 2000 Yes

Below we see that predict returns a list with three elements. The first element, class contains LDA’s predictions about the customer defaulting. Here we see that the second observation (non-student with balance of $2,000) is the only one kthat is predicted to default. The second element, posterior is a matrix that contains the posterior probability that the corresponding observations will or will not default. Here we see that the only observation to have a posterior probability of defaulting greater than 50% is observation 2, which is why the LDA model predicted this observation will default.

However, we also see that observation 4 has a 42% probability of defaulting. Right now, the model is predicting that this observation will not default because this probability is less than 50%; however, we will see shortly how we can make adjustments to our posterior probability thresholds. Finally, x contains the linear discriminant values, described earlier.

(df.pred <- predict(lda.m1, df, type="response"))  
## $class  
## [1] No Yes No No   
## Levels: No Yes  
##   
## $posterior  
## No Yes  
## 1 0.9903138 0.009686200  
## 2 0.4585630 0.541436990  
## 3 0.9940401 0.005959891  
## 4 0.5801226 0.419877403  
##   
## $x  
## LD1  
## 1 0.4335197  
## 2 2.6658881  
## 3 0.2055974  
## 4 2.4379658

As previously mentioned, the default setting is to use a 50% threshold for the posterior probabilities. We can recreate the predictions contained in the class element above:

# number of non-defaulters  
sum(df.pred$posterior[,1]>.5)  
## [1] 3  
  
# number of defaulters  
sum(df.pred$posterior[,2]>.5)  
## [1] 1

If we wanted to use a posterior probability threshold other than 50% in order to make predictions, then we could easily do so. For instance, suppose that a credit card company is extremely risk-averse and want to assume that a customer with 40% or greater probability is a high-risk customer. We can easily assess the number of high-risk customers.

# number of high-risk customers with 40$ probability of defaulting  
df.pred$posterior  
## No Yes  
## 1 0.9903138 0.009686200  
## 2 0.4585630 0.541436990  
## 3 0.9940401 0.005959891  
## 4 0.5801226 0.419877403  
df.pred$posterior[,2] %>% sum(. > .4)  
## [1] 2.97696  
sum(df.pred$posterior[, 2] > .4)  
## [1] 2

### Quadratic discriminant analysis]

As previously mentioned, LDA asssumes that the observations within each class are drawn from a multivatiate Gaussian distribution and the covariance of the predictor variables are common across all levels of the response variable .

Quadratic discriminant analysis (QDA) provides an alternative approach. Like LDA the QDA classifier assumes that the observations from each class of are drawn from a Gaussian distribution.

However, unlike LDA, QDA assumes that each class has its own covariance matrix. In other words, the predictor variables are not assumed to have common variance across each of the levels in . Mathemetically, it assumes that an observation from the th class is of the form , where is a covariance matrix for the th class. Under this assumption, the classifier assings an observation to the class for which:

is largest. Why is this important? Consider the image below. In trying to classify the observation into the three (color coded) classes, LDA provides linear decision boundaries that are based on the assumption that the observations vary consistently across all classes.

However, when looking at the data, it becomes apparent that the variability of the observations within each class differ. Consequently, QDA (right plot) is able to capture the differing covariances and provide more accurate non-linear classification decision boundaries.

knitr::include\_graphics("C:/Protected/Data Science/UC Business Analytics/image/QDA.png")

#### Estimate & Understand model

Similar to lda, we can use the MASS library to fit a QDa model. Here we use the qda function. The output is very similar to the lda output. It contains the prior probabilities and the group means. But it does not contain the coefficients of the linear disciminants, because the QDA classifier involves a quadratic, rather than a linear, function of the predictors.

(qda.m1 <- MASS::qda(default ~ balance + student, data = train))  
## Call:  
## qda(default ~ balance + student, data = train)  
##   
## Prior probabilities of groups:  
## No Yes   
## 0.9677526 0.0322474   
##   
## Group means:  
## balance studentYes  
## No 804.968 0.2956254  
## Yes 1776.971 0.3948718

#### Make predictions

The predict function works in exactly the same fashion as for LDA except it does not return the linear disciminant values. In comparing this simple prediction example to that seen in the LDA section we see minor changes in the posterior probabilities. Most notably, the posterior probability that observation 4 will default increased by nearly 8% points.

predict(qda.m1, df, prob = TRUE)  
## $class  
## [1] No Yes No No   
## Levels: No Yes  
##   
## $posterior  
## No Yes  
## 1 0.9957697 0.004230299  
## 2 0.4381383 0.561861660  
## 3 0.9980862 0.001913799  
## 4 0.5148050 0.485194962  
  
## $class  
## [1] No Yes No No   
## Levels: No Yes  
##   
## $posterior  
## No Yes  
## 1 0.9957697 0.004230299  
## 2 0.4381383 0.561861660  
## 3 0.9980862 0.001913799  
## 4 0.5148050 0.485194962

### Prediction Performance

Now that we understand the basics of evaluating our model and making predictions. Let’s assess how well our two models (lda.m1, qda.m1) perform on our test data set. First, we need to apply our models to the test data.

test.predicted.lda <- predict(lda.m1, newdata = test)  
test.predicted.qda <- predict(qda.m1, newdata = test)

Now, we can evaluate how well our model predicts by assessing the different classification rates discussed in the logistic regression tutorial. First, we will look at the confusion matrix in a percentage form.

The below results show that the models perform in a very similar manner. 96% of the predicted observations are true negatives and about 1% are true positives. Both models have a type II error of less than 3% in which the model predicts the customer will not default but they actually did. And both models have a type I error of less than 1% in which the models predict the customer will default but they never did.

lda.cm <- table(test$default, test.predicted.lda$class)  
qda.cm <- table(test$default, test.predicted.qda$class)  
  
list(LDA\_model = lda.cm %>% prop.table() %>% round(3),  
 QDA\_model = qda.cm %>% prop.table() %>% round(3))  
## $LDA\_model  
##   
## No Yes  
## No 0.964 0.002  
## Yes 0.028 0.007  
##   
## $QDA\_model  
##   
## No Yes  
## No 0.963 0.002  
## Yes 0.026 0.009  
  
## $LDA\_model  
##   
## No Yes  
## No 0.964 0.002  
## Yes 0.028 0.007  
##   
## $QDA\_model  
##   
## No Yes  
## No 0.963 0.002  
## Yes 0.026 0.009

Furtheremore, we can estimate the overall error rates. Here we see that the QDA model reduces the error rate by just a hair.

test %>%   
 mutate(lda.pred = (test.predicted.lda$class),  
 qda.pred = (test.predicted.qda$class)) %>%   
 summarise(  
 lda.error = mean(default != lda.pred),  
 qda.error = mean(default != qda.pred)  
 )  
## # A tibble: 1 x 2  
## lda.error qda.error  
## <dbl> <dbl>  
## 1 0.0291 0.0278  
## # A tibble: 1 × 2  
## lda.error qda.error  
## <dbl> <dbl>  
## 1 0.02909183 0.02782697

However, as we discussed in the last tutorial, the overall error may be less important then understanding the *precision* of our model. If we look at the raw number of our *confusion matrix* we can compute the precision:

* LDA model: %
* QDA model: %

So our QDA model has a slightly higer precision than the LDA model; however, both of them are lower than the logistic regression model precision of 29%.

list(LDA\_model = lda.cm,  
 QDA\_model = qda.cm)  
## $LDA\_model  
##   
## No Yes  
## No 3809 6  
## Yes 109 29  
##   
## $QDA\_model  
##   
## No Yes  
## No 3808 7  
## Yes 103 35

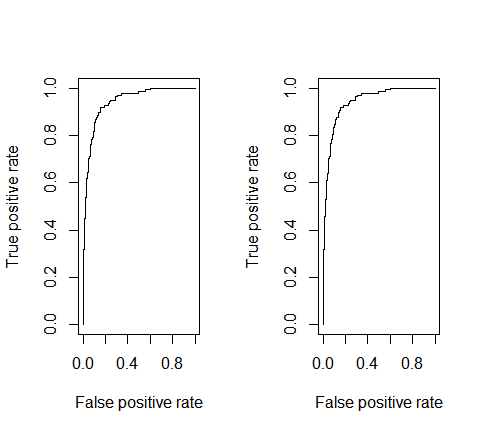
If we are concerned with increasing the precision of our model we can tune our model by adjusting the posterior probability threshold. For instance, we might label any customer with a posterior probability of default above 20% as high-risk.

Now the precision of our QDA model improves to %. However, the overall error rate has increased to 4%. But a credit card company may consider this slight increase in the total error rate to be a small price to pay for more accurate identification of individuals who do indeed default. It is important to keep in mind these kinds of trade-offs, which are common with classification models - tuning models can improve certain classification rates while worsening others.

# create adjusted predictions  
lda.pred.adj <- ifelse(test.predicted.lda$posterior[,2]>.2,"Yes","No")  
qda.pred.adj <- ifelse(test.predicted.qda$posterior[,2]>.2, "Yes", "No")  
  
# create new confusion matrices  
list(LDA\_model = table(test$default, lda.pred.adj),  
 QDA\_model = table(test$default, qda.pred.adj))  
## $LDA\_model  
## lda.pred.adj  
## No Yes  
## No 3731 84  
## Yes 69 69  
##   
## $QDA\_model  
## qda.pred.adj  
## No Yes  
## No 3699 116  
## Yes 55 83  
## $LDA\_model  
## lda.pred.adj  
## No Yes  
## No 3731 84  
## Yes 69 69  
##   
## $QDA\_model  
## qda.pred.adj  
## No Yes  
## No 3699 116  
## Yes 55 83

We can also assess the ROC curve for our models as we did in the logistic regression tutorial and compute the AUC.

#ROC curve  
library(ROCR)  
  
par(mfrow=c(1,2))  
  
prediction(test.predicted.lda$posterior[,2], test$default) %>%   
 performance(measure = "tpr", x.measure = "fpr") %>%   
 plot()  
  
prediction(test.predicted.qda$posterior[,2], test$default) %>%   
 performance(measure = "tpr", x.measure = "fpr") %>%   
 plot()



# model 1 AUC  
prediction(test.predicted.lda$posterior[,2], test$default) %>%  
 performance(measure = "auc") %>%  
 .@y.values  
## [[1]]  
## [1] 0.9420727  
## [[1]]  
## [1] 0.9420727  
  
# model 2 AUC  
prediction(test.predicted.qda$posterior[,2], test$default) %>%  
 performance(measure = "auc") %>%  
 .@y.values  
## [[1]]  
## [1] 0.9420746  
## [[1]]  
## [1] 0.9420746

### Comparison of logistic regression & discriminant analysis

The logistic regression and LDA methods are closely connected and differ primarily in their fitting procedure. Consequently, the two often produce similar results. However, LDA assumes that the observations are drawn from a \_Gaussian distribution\_\_ with a common covariance matrix across each class of , and so can provide some improvements over logistic regression when this assumption approximately holds.

Conversely, logistic regression can outperform LDA if these Gaussian assumptions are not met. Both LDA and logistic regression produce linear decision boundaries so when the true decision boundaries are linear, then the LDA and logistic regression approaches will tend to perform well. QDA, on the other hand, provides a non-linear quadratic decision boundary. Thus, when the decision boundary is moderately non-linear, QDA may give better results (we will see other non-linear classifiers in later tutorials).

What is important to keep in mind is that non one method will dominate the others in every situation. And, often we want to compare multiple approaches to see how they compare. To illustrate, we will examine stock market (Smarket) data provided by the ISLR package.

This data sset consists of percentage returs for the S&P stock index over 1,250 days, from the beginning of 2001 until the end of 2005. For each date, percentage returns for each of the five previous trading days, *Lag1* through *Lag5* are provided. In addition, *Volume* (the number of shares traded on the previous day, in billions), *Today* (the percentage return on the date in question), and *Direction*(whether the market was Up or down on this date) are provided.

dim(ISLR::Smarket)  
## [1] 1250 9  
head(ISLR::Smarket)  
## Year Lag1 Lag2 Lag3 Lag4 Lag5 Volume Today Direction  
## 1 2001 0.381 -0.192 -2.624 -1.055 5.010 1.1913 0.959 Up  
## 2 2001 0.959 0.381 -0.192 -2.624 -1.055 1.2965 1.032 Up  
## 3 2001 1.032 0.959 0.381 -0.192 -2.624 1.4112 -0.623 Down  
## 4 2001 -0.623 1.032 0.959 0.381 -0.192 1.2760 0.614 Up  
## 5 2001 0.614 -0.623 1.032 0.959 0.381 1.2057 0.213 Up  
## 6 2001 0.213 0.614 -0.623 1.032 0.959 1.3491 1.392 Up

Let’s model this data with logistic regression, LDA and QDA to assess well each model does in predicting the direction of the stock market based on previous data returns. We will use 2001-2004 data to train our models and then test these models on 2005 data.

train <- subset(ISLR::Smarket, Year < 2005)  
test <- subset(ISLR::Smarket, Year == 2005)

#### Logistic regression

Here we fit a logistic regression model to the training data. Looking at the summary our modle does not look too convincing considering no coefficients are statistically significant and our residual deviance has barely been reduced.

glm.fit <- glm(Direction ~ Lag1 + Lag2 + Lag3 + Lag4 + Lag5 + Volume,  
 data = train,  
 family = binomial)  
  
glm.fit %>% summary()  
##   
## Call:  
## glm(formula = Direction ~ Lag1 + Lag2 + Lag3 + Lag4 + Lag5 +   
## Volume, family = binomial, data = train)  
##   
## Deviance Residuals:   
## Min 1Q Median 3Q Max   
## -1.302 -1.190 1.079 1.160 1.350   
##   
## Coefficients:  
## Estimate Std. Error z value Pr(>|z|)  
## (Intercept) 0.191213 0.333690 0.573 0.567  
## Lag1 -0.054178 0.051785 -1.046 0.295  
## Lag2 -0.045805 0.051797 -0.884 0.377  
## Lag3 0.007200 0.051644 0.139 0.889  
## Lag4 0.006441 0.051706 0.125 0.901  
## Lag5 -0.004223 0.051138 -0.083 0.934  
## Volume -0.116257 0.239618 -0.485 0.628  
##   
## (Dispersion parameter for binomial family taken to be 1)  
##   
## Null deviance: 1383.3 on 997 degrees of freedom  
## Residual deviance: 1381.1 on 991 degrees of freedom  
## AIC: 1395.1  
##   
## Number of Fisher Scoring iterations: 3  
glm.fit %>% broom::tidy()  
## # A tibble: 7 x 5  
## term estimate std.error statistic p.value  
## <chr> <dbl> <dbl> <dbl> <dbl>  
## 1 (Intercept) 0.191 0.334 0.573 0.567  
## 2 Lag1 -0.0542 0.0518 -1.05 0.295  
## 3 Lag2 -0.0458 0.0518 -0.884 0.377  
## 4 Lag3 0.00720 0.0516 0.139 0.889  
## 5 Lag4 0.00644 0.0517 0.125 0.901  
## 6 Lag5 -0.00422 0.0511 -0.0826 0.934  
## 7 Volume -0.116 0.240 -0.485 0.628

Now we comupte the predictions for 2005 and compare them to the actual movements of the market over that time period with a confusion matrix. The results are rather disappointing: the test error rate is 52%, which is worse than random guessing! Furthermore, our precision is only 31%. However, this should not be surprising considering the lack of statistical significance with our predictors.

# predictions  
glm.probs <- predict(glm.fit, test, type="response")  
  
# confusion matrix  
table(test$Direction, ifelse(glm.probs >0.5, "Up", "Down"))  
##   
## Down Up  
## Down 77 34  
## Up 97 44  
  
# accracy rate  
mean(ifelse(glm.probs > 0.5, "Up", "Down")==test$Direction)  
## [1] 0.4801587  
  
# error rate  
mean(ifelse(glm.probs > 0.5, "Up", "Down")!=test$Direction)  
## [1] 0.5198413

Remeber that using predictors that have no relationship with the response tends to cause a deterioration in the test error rate(since such predictors cause an increase in variance without a corresponding decrease in bias), and so removing such predictors may in turn yield an improvement. The variables that appear to have the highest importance rating are *Lag1* and *Lag2*.

caret::varImp(glm.fit)  
## Overall  
## Lag1 1.04620896  
## Lag2 0.88432632  
## Lag3 0.13941784  
## Lag4 0.12456821  
## Lag5 0.08257344  
## Volume 0.48517564

Lets re-fit with just these two variables and reassess performance. We don’t see much improvement within our model summary.

glm.fit <- glm(Direction ~ Lag1 + Lag2,   
 data = train,  
 family = binomial)  
  
summary(glm.fit)  
##   
## Call:  
## glm(formula = Direction ~ Lag1 + Lag2, family = binomial, data = train)  
##   
## Deviance Residuals:   
## Min 1Q Median 3Q Max   
## -1.345 -1.188 1.074 1.164 1.326   
##   
## Coefficients:  
## Estimate Std. Error z value Pr(>|z|)  
## (Intercept) 0.03222 0.06338 0.508 0.611  
## Lag1 -0.05562 0.05171 -1.076 0.282  
## Lag2 -0.04449 0.05166 -0.861 0.389  
##   
## (Dispersion parameter for binomial family taken to be 1)  
##   
## Null deviance: 1383.3 on 997 degrees of freedom  
## Residual deviance: 1381.4 on 995 degrees of freedom  
## AIC: 1387.4  
##   
## Number of Fisher Scoring iterations: 3

However, our prediction classification rates have improved slightly. Our error rate has decreased to 44% (accuracy = 56%) and our precision has increased to 75%. However, its worth noting that the market moved up 56% of the time in 2005 and moved down 44% of the time. Thus, the logistic regression approach is no better than a naive approach!

# predictions  
glm.probs <- predict(glm.fit, test, type = "response")  
  
# confusion matrix  
table(test$Direction, ifelse(glm.probs > 0.5, "Up", "Down"))  
##   
## Down Up  
## Down 35 76  
## Up 35 106  
##   
## Down Up  
## Down 35 76  
## Up 35 106  
  
# accuracy rate  
mean(ifelse(glm.probs > 0.5, "Up", "Down") == test$Direction)  
## [1] 0.5595238  
## [1] 0.5595238  
  
# error rate  
mean(ifelse(glm.probs > 0.5, "Up", "Down") != test$Direction)  
## [1] 0.4404762  
## [1] 0.4404762

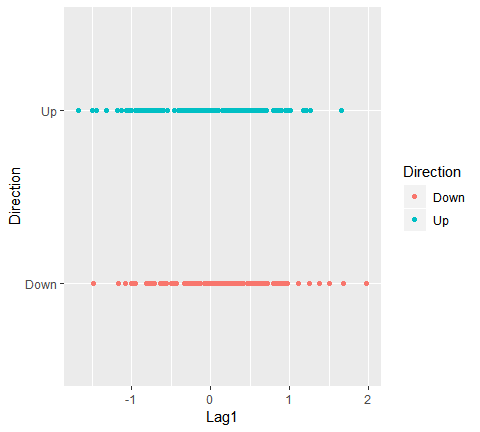
#### Linear disciminant analysis (LDA)

Now we will perform LDA on the stock market data. Our summary shows that our prior probabilities of market movements are 49% (down) and 51% (up). The group means indicate that there is a tendency for the previous 2 days’ returns to be negative on days when the market increases, and a tendency for the previous days’ returns to be positive on days when the market declines.

lda.fit <- lda(Direction ~ Lag1 + Lag2, data = train)  
lda.fit  
## Call:  
## lda(Direction ~ Lag1 + Lag2, data = train)  
##   
## Prior probabilities of groups:  
## Down Up   
## 0.491984 0.508016   
##   
## Group means:  
## Lag1 Lag2  
## Down 0.04279022 0.03389409  
## Up -0.03954635 -0.03132544  
##   
## Coefficients of linear discriminants:  
## LD1  
## Lag1 -0.6420190  
## Lag2 -0.5135293

When we predict with our LDA model and assess the confusion matrix we see that our prediction rates mirror those produced by logistic regression. The overall error and the **precision** of our LDA and logistic regression models are the same.

# predictions  
test.predicted.lda <- predict(lda.fit, newdata = test)  
  
# confusion matrix  
table(test$Direction, test.predicted.lda$class)  
##   
## Down Up  
## Down 35 76  
## Up 35 106  
  
# accuracy rate  
mean(test.predicted.lda$class == test$Direction)  
## [1] 0.5595238  
## [1] 0.5595238  
  
# error rate  
mean(test.predicted.lda$class != test$Direction)  
## [1] 0.4404762  
## [1] 0.4404762  
  
# visualize the results  
test %>%   
 ggplot(aes(Lag1, Direction))+  
 geom\_point(aes(col=Direction))+  
 geom\_smooth(method = "lda", formula = y~x)



#### Quadratic disciminant analysis

Lastly, we will predict with a QDA model to see if we can improve our performance.

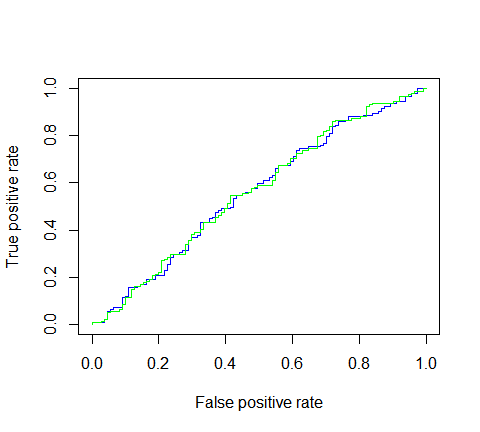
(qda.fit <- qda(Direction ~ Lag1 + Lag2, data = train))  
## Call:  
## qda(Direction ~ Lag1 + Lag2, data = train)  
##   
## Prior probabilities of groups:  
## Down Up   
## 0.491984 0.508016   
##   
## Group means:  
## Lag1 Lag2  
## Down 0.04279022 0.03389409  
## Up -0.03954635 -0.03132544

Surprisingly, the QDA predictions are acrurate almost 60% of the time! Furthermore, the precision of the model is 86%. This level of accuracy is quite impressive for stock market data, which is known to be quite hard to model accurarately. This uggests that the quadratic form assumed by QDA may capture the true relationship more accurately than the linear forms assumed by the LDA and logstic regression.

# predictions  
test.predicted.qda <- predict(qda.fit, newdata = test)  
  
# confusion matrix  
table(test$Direction, test.predicted.qda$class)  
##   
## Down Up  
## Down 30 81  
## Up 20 121  
##   
## Down Up  
## Down 30 81  
## Up 20 121  
  
# accuracy rate  
mean(test.predicted.qda$class == test$Direction)  
## [1] 0.5992063  
## [1] 0.5992063  
  
# error rate  
mean(test.predicted.qda$class != test$Direction)  
## [1] 0.4007937  
## [1] 0.4007937

We can see how our models differ with a ROC curve. Although you can’t tell, the logistic regressio and LDA ROC curve sit directly on top of one another. However, we can see how the QDA (green) differs slightly.

# ROC curve  
library(ROCR)  
  
p1 <- prediction(glm.probs, test$Direction) %>%   
 performance(measure = "tpr", x.measure = "fpr")  
  
p2 <- prediction(test.predicted.lda$posterior[,2], test$Direction) %>%   
 performance(measure = "tpr", x.measure = "fpr")  
  
p3 <- prediction(test.predicted.qda$posterior[,2], test$Direction) %>%   
 performance(measure = "tpr", x.measure = "fpr")  
  
plot(p1, col = "red")  
plot(p2, add = TRUE, col = "blue")  
plot(p3, add = TRUE, col = "green")



The difference is subtle. You can see where we experince increases in the true positive predictions (where the green line go above the red and blue lines). An although our precision increases, overall AUC is not that much higher.

# logistic regression AUC  
prediction(  
 glm.probs, test$Direction) %>%   
 performance(measure = "auc") %>%   
 .@y.values  
## [[1]]  
## [1] 0.5584308  
  
# LDA AUC  
prediction(test.predicted.lda$posterior[,2], test$Direction) %>%  
 performance(measure = "auc") %>%  
 .@y.values  
## [[1]]  
## [1] 0.5584308  
## [[1]]  
## [1] 0.5584308  
  
# QDA AUC  
prediction(test.predicted.qda$posterior[,2], test$Direction) %>%  
 performance(measure = "auc") %>%  
 .@y.values  
## [[1]]  
## [1] 0.5620088  
## [[1]]  
## [1] 0.5620088

Although we get some improvements with the QDA model we probabily want to continue tuning our models or assess other techniques to improve our classification performance before hedging any bets. But this ullustrates the usefulness of assessing multiple classification models.

### Additional resources {#}

This will get you up and running with LDA and QDA. Keep in mind that there is a lot more you can dig into so the following resources will help you learn more:

* An Introduction to Statistical Learning
* Applied Predictive Modeling
* Elements of Statistical Learning

## Support vector machine

The advent of computers brough on rapid advances in the field of statistical classification, one of which is the *Support Vector Machine*, or SVM. The goal of an SVM is to take groups of observations and construct boundaries to predict which group future observations belong to based on their measurements. The different groups that must be separated will be called “classes”. SVMs can handle any number of classes, as well as observations of any dimension. SVMs can take almost any shape (including linear, radial, and polynomial, among others), and are generally flexible enough to be used in almost any classification endeavor that the user chooses to undertake.

### tl;dr

1. [Replication Requirements](#SVM_RR): What you’ll need to reproduce the analysis in this tutorial
2. [Maximal Margin Classifier](#SVM_MM_Classifier): Constructing a classification line for completely separable data
3. [Support Vector Classifiers](#SVM_Classifier): Constructing a classification line for data that is not separable
4. [Support Vector Machines](#SVM_Overview): Constructing a classification boundary, whether linear or nonlinear, for data that may or may not be separable
5. [SVMs for Multiple Classes](#SVM_Multi_Class): SVM techniques for more than 2 classes of observations

### Replication Requirements

In this tutorial, we will leverage the tidyverse package to perform data manipulation, the kernlab and e1071 packages to perform calculkating and produce visualization related to SVMs, and the ISLR package to load a real world data set and demonstrate the functionality of SVM.

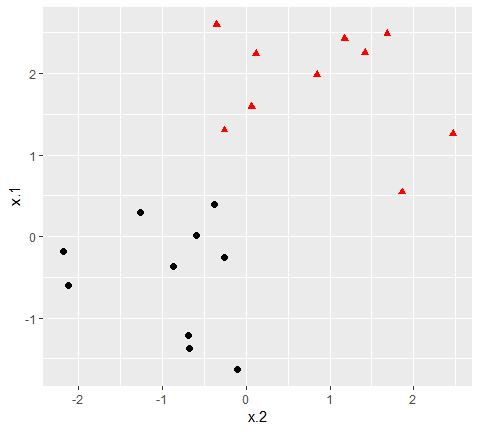
# set pseudorandom number generator  
set.seed(10)  
  
# Attach Packages  
library(tidyverse) # data manipulation and visualization  
library(kernlab) # SVM methodology  
library(e1071) # SVM methodology  
library(ISLR) # contains example data set "Khan"  
library(RColorBrewer) # customized coloring of plots

The data sets used in the tutorial (with the exception of Khan) will be generated using built-in R commands. The Support Vector Machine methodology is sound for any number of dimensions, but becomes difficult to visualize for more than 2. As previously mentioned, SVMs are robust for any number of classes, but we will stick to no more than 3 for the duration of this tutorial.

### Maximum Margin Classifier

If the classes are separable by a linear boundary, we can use a *Maximal Margin Classifier* to find the classification boundary. To visualize an example of separated data, we generate 40 random observations and assign them to two classes. Upon visual inspection, we can see that infinitely many lines exist that split the two classes.

# Construct sample data set - completely separated  
x <- matrix(rnorm(20\*2), ncol = 2)  
y <- c(rep(-1,10), rep(1,10))  
x[y==1,] <- x[y==1,] + 3/2  
dat <- data.frame(x=x, y=as.factor(y))  
  
# Plot data  
ggplot(data = dat, aes(x = x.2, y = x.1, color = y, shape = y)) +   
 geom\_point(size = 2) +  
 scale\_color\_manual(values=c("#000000", "#FF0000")) +  
 theme(legend.position = "none")

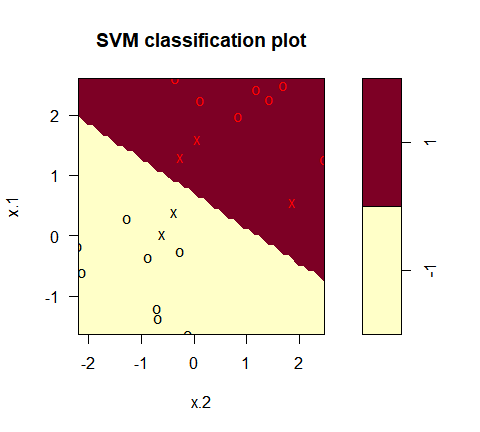


The goal of the maximum margin classifier is to identify the linear boundary that maximizes the total distance between the line and the closeset point in each class. We can use the svm() function in the e1071 to find this boundary.

For the aesttheic of the svm() function, there are four types of kernel availabile:

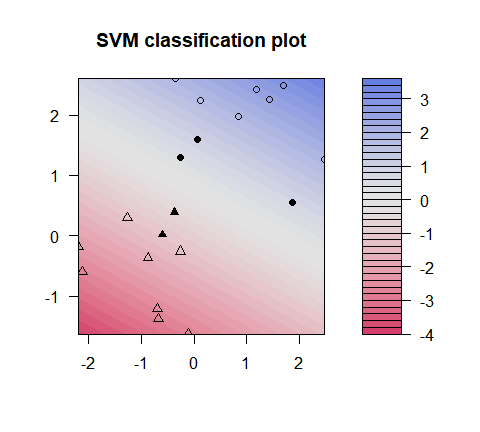
1. linear:
2. polynomial:
3. radial basis:
4. sigmoid:

# Fit Support Vector Machine model to data set  
svmfit <- svm(y ~ .,   
 data = dat,  
 kernel = "linear",   
 scale = FALSE)  
  
# Plot Results  
plot(svmfit, dat)



In the plot, points that are represented by an “X” are the **support vectors**, or the points that directly affect the classification line. The points marked with an “o” are the other points, which don’t affect the calculation of the line. This principle will lay the foundation for support vector machines. The same plot can be generated using the kernlab package, with the following results:

# fit model and produce plot  
kernfit <- ksvm(x, y, type = "C-svc", kernel = 'vanilladot')  
## Setting default kernel parameters  
plot(kernfit, data = x)



But how do we decide how costly these missclasifications actually are? Instead of specifying a cost up front, we can use the tune() function from e1071 to test various costs and identify which value produces the best fitting model.

# find optimal cost of misclassification  
tune.out <- tune(svm, y~., data = dat, kernel = "linear",  
 ranges = list(cost = c(0.001, 0.01, 0.1, 1, 5, 10, 100)))  
# extract the best model  
(bestmod <- tune.out$best.model)  
##   
## Call:  
## best.tune(method = svm, train.x = y ~ ., data = dat, ranges = list(cost = c(0.001,   
## 0.01, 0.1, 1, 5, 10, 100)), kernel = "linear")  
##   
##   
## Parameters:  
## SVM-Type: C-classification   
## SVM-Kernel: linear   
## cost: 1   
## gamma: 0.5   
##   
## Number of Support Vectors: 5  
##   
## Call:  
## best.tune(method = svm, train.x = y ~ ., data = dat, ranges = list(cost = c(0.001,   
## 0.01, 0.1, 1, 5, 10, 100)), kernel = "linear")  
##   
##   
## Parameters:  
## SVM-Type: C-classification   
## SVM-Kernel: linear   
## cost: 0.1   
## gamma: 0.5   
##   
## Number of Support Vectors: 16

For our data set, the optimal cost (from amongst the choices we provided), is calculated to be 0.1, which does not penalize the model for miscclasified observations. Once this model has been identified, we can construct a table of predicted classes against true classes using the predict() command as follows.

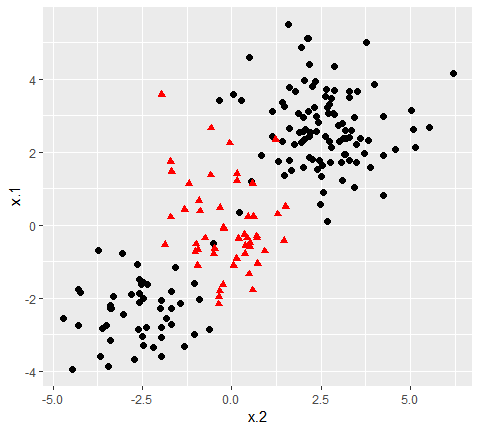
# Create a table of misclassified observations  
ypred <- predict(bestmod, dat)  
(misclass <- table(predict = ypred, truth = dat$y))  
## truth  
## predict -1 1  
## -1 10 0  
## 1 0 10  
  
## truth  
## predict -1 1  
## -1 9 3  
## 1 1 7

Using this support vector classifier, 80% of the observations were correctly classified, which matches what we see in the plot. If we wanted to test our classifier more rigorously, we could split our data into training and testing sets and then see how our SVC performed with the observations not used to construct the model. We will use this training-testing method later in this tutorial to validate our SVMs.

### Support Vector Machines

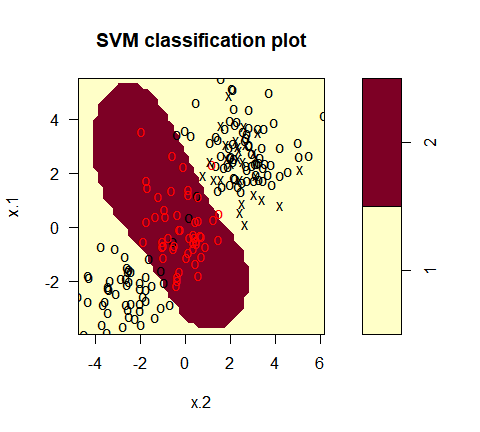
Support Vector Classifiers are a subset of the group of classification structures known as Support Vector Machines. Support Vector Machines can construct classification boundaries that are nonlinear in shape. The options for classification structures using the svm() command from the e1071 package are linear, polynomial, radial, and sigmoid. To demonstrate a nonlinear classification boundary, we will construct a new data set.

# construct larger random data set  
x <- matrix(rnorm(200\*2), ncol = 2)  
x[1:100,] <- x[1:100,] + 2.5  
x[101:150,] <- x[101:150,] - 2.5  
y <- c(rep(1,150), rep(2,50))  
dat <- data.frame(x=x,y=as.factor(y))  
  
# Plot data  
ggplot(data = dat, aes(x = x.2, y = x.1, color = y, shape = y)) +   
 geom\_point(size = 2) +  
 scale\_color\_manual(values=c("#000000", "#FF0000")) +  
 theme(legend.position = "none")



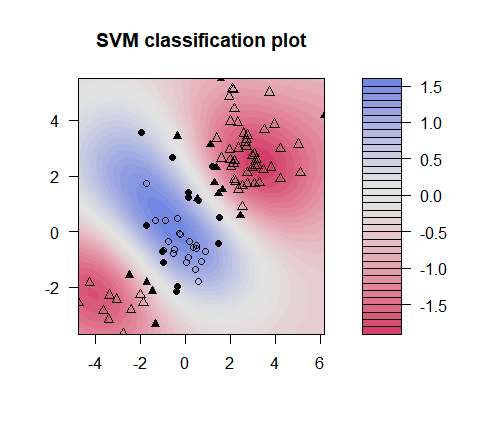
Notice that the data is not linearly separable, and furthermore, isn’t all clustered together in a single group. There are two sections of class 1 observations with a cluster of class 2 observations in between. To demonstrate the power of SVMs, we’ll take 100 random observations from the set and use them to construct our boundary. We set kernel = "radial" based on the shape of our data and plot the results.

# set pseudorandom number generator  
set.seed(123)  
# sample training data and fit model  
train <- base::sample(200,100, replace = FALSE)  
svmfit <- svm(y~., data = dat[train,], kernel = "radial", gamma = 1, cost = 1)  
# plot classifier  
plot(svmfit, dat)



The same procedure can be run using the kernlab package, which has far more kernel options than the corresponding function in e1071. In addition to the four choices in e1071, this package allows use of a hyperbolic tangent, Laplacian, Bessel, Spline, String, or ANOVA RBF kernel. To fit this data, we set the cost to be the same as it was before, 1.

# Fit radial-based SVM in kernlab  
kernfit <- ksvm(x[train,],y[train], type = "C-svc", kernel = 'rbfdot', C = 1, scaled = c())  
# Plot training data  
plot(kernfit, data = x[train,])



We see that, at least visually, the SVM does a reasonable job of separating the two classes. To fit the model, we used cost = 1, but as mentioned previously, it isn’t usually obvious which cost will produce the optimal classification boundary. We can use the tune() command to try several different values of cost as well as several different values of , a scaling parameter used to fit nonlinear boundaries.

# tune model to find optimal cost, gamma values  
tune.out <- tune(svm, y~., data = dat[train,], kernel = "radial",  
 ranges = list(cost = c(0.1,1,10,100,1000),  
 gamma = c(0.5,1,2,3,4)))  
# show best model  
tune.out$best.model  
##   
## Call:  
## best.tune(method = svm, train.x = y ~ ., data = dat[train, ],   
## ranges = list(cost = c(0.1, 1, 10, 100, 1000), gamma = c(0.5,   
## 1, 2, 3, 4)), kernel = "radial")  
##   
##   
## Parameters:  
## SVM-Type: C-classification   
## SVM-Kernel: radial   
## cost: 1   
## gamma: 0.5   
##   
## Number of Support Vectors: 27  
##   
## Call:  
## best.tune(method = svm, train.x = y ~ ., data = dat[train, ],   
## ranges = list(cost = c(0.1, 1, 10, 100, 1000), gamma = c(0.5,   
## 1, 2, 3, 4)), kernel = "radial")  
##   
##   
## Parameters:  
## SVM-Type: C-classification   
## SVM-Kernel: radial   
## cost: 1   
## gamma: 0.5   
##   
## Number of Support Vectors: 34

The model that reduces the error the most in the training data uses a cost of 1 and value of 0.5. We can now see how well the SVM performs by predicting the class of the 100 testing observations:

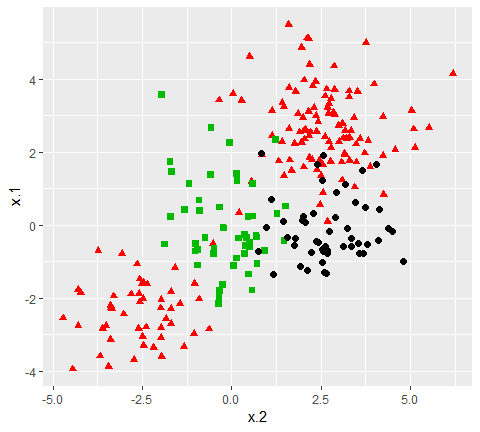
# validate model performance  
(valid <- table(true = dat[-train,"y"], pred = predict(tune.out$best.model,newx = dat[-train,])))  
## pred  
## true 1 2  
## 1 54 29  
## 2 12 5  
## pred  
## true 1 2  
## 1 58 19  
## 2 16 7

Our best-fitting model produces 65% accuracy in identifying classes. For such a complicated shape of observations, this performed reasonably well. We can challenge this method further by adding additional classes of observations.

### SVMs for Multiple Classes

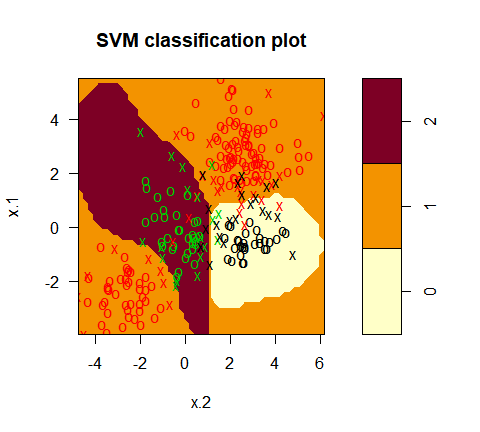
The procedure does not change for data sets that involve more than two classes of observations. We construct our data set the same way as we have previously, only now specifying three classes instead of two:

# construct data set  
x <- rbind(x, matrix(rnorm(50\*2), ncol = 2))  
y <- c(y, rep(0,50))  
x[y==0,2] <- x[y==0,2] + 2.5  
dat <- data.frame(x=x, y=as.factor(y))  
# plot data set  
ggplot(data = dat, aes(x = x.2, y = x.1, color = y, shape = y)) +   
 geom\_point(size = 2) +  
 scale\_color\_manual(values=c("#000000","#FF0000","#00BA00")) +  
 theme(legend.position = "none")



The commands don’t change for the e1071 package. We specify a cost and tuning parameter and fit a support vector machine. The results and interpretation are similar to two-class classification.

# fit model  
svmfit <- svm(y~., data = dat, kernel = "radial", cost = 10, gamma = 1)  
# plot results  
plot(svmfit, dat)



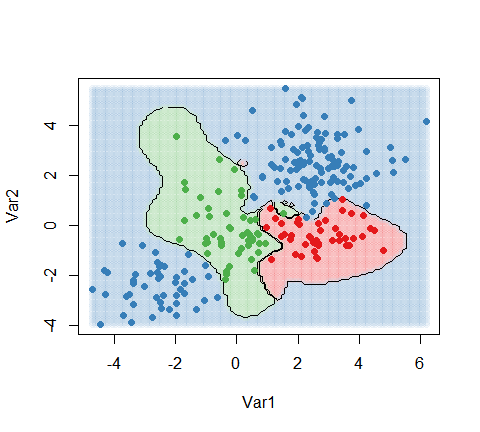
We can check to see how well our model fit the data by using the predict() command, as follows:

#construct table  
ypred <- predict(svmfit, dat)  
(misclass <- table(predict = ypred, truth = dat$y))  
## truth  
## predict 0 1 2  
## 0 39 3 3  
## 1 7 144 1  
## 2 4 3 46  
## truth  
## predict 0 1 2  
## 0 38 2 4  
## 1 8 143 4  
## 2 4 5 42

As shown in the resulting table, 89% of our training observations were correctly classified. However, since we didn’t break our data into training and testing sets, we didn’t truly validate our results.

The kernlab package, on the other hand, can fit more than 2 classes, but cannot plot the results. To visualize the results of the ksvm function, we take the steps listed below to create a grid of points, predict the value of each point, and plot the results:

# fit and plot  
kernfit <- ksvm(as.matrix(dat[,2:1]),dat$y, type = "C-svc", kernel = 'rbfdot',   
 C = 100, scaled = c())  
  
# Create a fine grid of the feature space  
x.1 <- seq(from = min(dat$x.1), to = max(dat$x.1), length = 100)  
x.2 <- seq(from = min(dat$x.2), to = max(dat$x.2), length = 100)  
x.grid <- expand.grid(x.2, x.1)  
  
# Get class predictions over grid  
pred <- predict(kernfit, newdata = x.grid)  
  
# Plot the results  
cols <- brewer.pal(3, "Set1")  
plot(x.grid, pch = 19, col = adjustcolor(cols[pred], alpha.f = 0.05))  
  
classes <- matrix(pred, nrow = 100, ncol = 100)  
contour(x = x.2, y = x.1, z = classes, levels = 1:3, labels = "", add = TRUE)  
  
points(dat[, 2:1], pch = 19, col = cols[predict(kernfit)])



#### Aplications

The Khan data set contains data on 83 tissue samples with 2308 gene expression measurements on each sample. These were split into 63 training observations and 20 testing observations, and there are four distinct classes in the set. It would be impossible to visualize such data, so we choose the simplest classifier (linear) to construct our model. We will use the svm command from e1071 to conduct our analysis.

# fit model  
dat <- data.frame(x = Khan$xtrain, y=as.factor(Khan$ytrain))  
(out <- svm(y~., data = dat, kernel = "linear", cost=10))  
##   
## Call:  
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 10)  
##   
##   
## Parameters:  
## SVM-Type: C-classification   
## SVM-Kernel: linear   
## cost: 10   
## gamma: 0.0004332756   
##   
## Number of Support Vectors: 58  
##   
## Call:  
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 10)  
##   
##   
## Parameters:  
## SVM-Type: C-classification   
## SVM-Kernel: linear   
## cost: 10   
## gamma: 0.0004332756   
##   
## Number of Support Vectors: 58

First of all, we can check how well our model did at classifying the training observations. This is usually high, but again, doesn’t validate the model. If the model doesn’t do a very good job of classifying the training set, it could be a red flag. In our case, all 63 training observations were correctly classified.

# check model performance on training set  
table(out$fitted, dat$y)  
##   
## 1 2 3 4  
## 1 8 0 0 0  
## 2 0 23 0 0  
## 3 0 0 12 0  
## 4 0 0 0 20  
##   
## 1 2 3 4  
## 1 8 0 0 0  
## 2 0 23 0 0  
## 3 0 0 12 0  
## 4 0 0 0 20

To perform validation, we can check how the model performs on the testing set:

# validate model performance  
dat.te <- data.frame(x=Khan$xtest, y=as.factor(Khan$ytest))  
pred.te <- predict(out, newdata=dat.te)  
table(pred.te, dat.te$y)  
##   
## pred.te 1 2 3 4  
## 1 3 0 0 0  
## 2 0 6 2 0  
## 3 0 0 4 0  
## 4 0 0 0 5  
##   
## pred.te 1 2 3 4  
## 1 3 0 0 0  
## 2 0 6 2 0  
## 3 0 0 4 0  
## 4 0 0 0 5

The model correctly identifies 18 of the 20 testing observations. SVMs and the boundaries they impose are more difficult to interpret at higher dimensions, but these results seem to suggest that our model is a good classifier for the gene data.

## Resampling methods

Resampling methods are an indispensable tool in modern statistics. They involve repeatedly drawing samples from a training set and refitting a model of interest on each sample in order to obtain additional information about the fitted model. For example, in order to estimate the variability of a linear regression fit, we can repeatedly draw different samples from the training data, fit a linear regression to each new sample, and then examine the extent to which the resulting fits differ. Such an approach may allow us to obtain information that would not be available from fitting the model only once using the oriiginal training sample.

### tl;dr

This tutorial services asa an itorudction to sampling methods and covers:

1. [Replication requirements](#RS_RR): What you’ll need to reproduce the analysis in this tutorial.
2. [Why resampling](#RS_Overview): Understand why resampling is important.
3. [Leave-one-out cross-validation](#RS_LOCV): Provide greater reliability of an estimate test error.
4. [k-fold cross validation](#RS_KFCV): A faster alternative to leave-one-out cross validation.
5. [Bootstrapping](#RS_BSPNG): Quantify uncertainty around a particular statistic.
6. [Additional resources](#RS_Resources): Additional resources to help you learn more.

### Replication requirements

This tutorial primarily leverages the Auto data provided by the ISLR package. This is a data set that contains gas mileage, horsepower, and other information for 392 vehicles. We’ll also use tidyverse for some basic data manipulation and visualization. Most importantly, we’ll use the boot package to illustrate resampling methods.

# Packages  
library(tidyverse) # data manipulation and visualization  
library(boot) # resampling and bootstrapping  
  
# Load data   
(auto <- as\_tibble(ISLR::Auto))  
## # A tibble: 392 x 9  
## mpg cylinders displacement horsepower weight acceleration year origin  
## <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 18 8 307 130 3504 12 70 1  
## 2 15 8 350 165 3693 11.5 70 1  
## 3 18 8 318 150 3436 11 70 1  
## 4 16 8 304 150 3433 12 70 1  
## 5 17 8 302 140 3449 10.5 70 1  
## 6 15 8 429 198 4341 10 70 1  
## 7 14 8 454 220 4354 9 70 1  
## 8 14 8 440 215 4312 8.5 70 1  
## 9 14 8 455 225 4425 10 70 1  
## 10 15 8 390 190 3850 8.5 70 1  
## # ... with 382 more rows, and 1 more variable: name <fct>

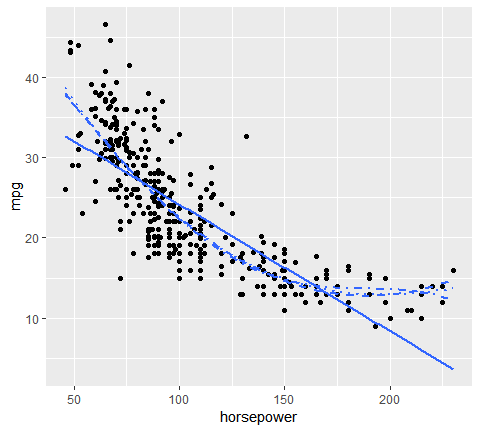
### Resampling basics

Thus far, in our tutorial we have been using the validation or hold\_out approach to estimate the predicted error of our predictive models. This involves randomly dividing the available set of observations into two parts, a *training set* and a *testing set* (aka *validation set*). Our statistical model is fit on the training set, and the fitted model is used to predict the responses for the observations in the validation set. The resulting validation set error rate (typically assessed using MSE in the case of a quantitative response) provides an estimate of the test error rate.

The validation set approach is conceptually simple and is easy to implement. But it has two potential drawbacks:

First, the estimate of the test error rate can be highly variable, depending on precisely which observations are included in the training set and which observations are included in the validation set. I will illustrate on our auto data set. Here we see that there is a relationship between mpg and horsepower and it doesn’t seem linear but we’re not sure which polynomial degree creates the best fit.

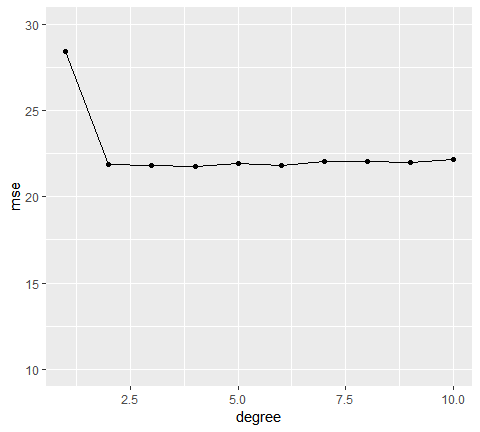
ggplot(auto, aes(horsepower, mpg)) +  
 geom\_point() +  
 geom\_smooth(method = "lm", se = FALSE) +  
 geom\_smooth(method = "lm", formula = y ~ poly(x, 2), se = FALSE, linetype = 2) +  
 geom\_smooth(method = "lm", formula = y ~ poly(x, 3), se = FALSE, linetype = 3) +  
 geom\_smooth(method = "lm", formula = y ~ poly(x, 4), se = FALSE, linetype = 4)



Let’s go ahead and do the traditional validation set approach to split our data into a training and testing set. Then we will fit 10 different models ranging from a linear model to a 10th degree polynomial model.

The results show us there is a steep decline in our test error (MSE) rate when we go from a linear model to a quadratic model; however, the MSE flatlines beyond that point suggesting that adding more polynomial degrees likely does not improve the model performance.

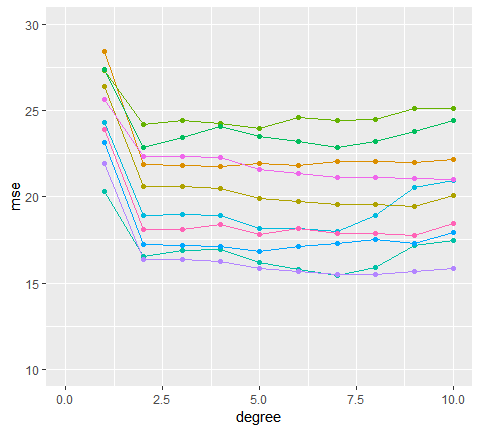
set.seed(1)  
sample <- sample(c(TRUE, FALSE), nrow(auto), replace=T,  
 prob = c(0.6, 0.4))  
train <- auto[sample,]  
test <- auto[!sample, ]  
  
# loop for first ten polynomial  
mse.df <- tibble(degree=1:10, mse=NA)  
  
for (i in 1:10){  
 lm.fit <- lm(mpg ~ poly(horsepower, i), data = train)  
 mse.df[i,2] <- mean((test$mpg - predict(lm.fit, test))^2)  
}  
  
ggplot(mse.df, aes(degree, mse))+  
 geom\_line()+  
 geom\_point()+  
 ylim(c(10,30))



However, our MSE is dependent on our training and test samples. If we repeat the process of randomly splitting the sample set into two parts, we will get a somewhat different estimate for the test MSE each time. I illustrate below, which displays ten different validation set MSE curves from the auto data set, produced using ten different random splits of the observations into training and validation sets. All ten curves indicate that the model with quadratic term has a dramatically smaller validation set MSE than the model with only a linear term.

Furthermore, all ten curves indicate that there is not much benefit in including cubic or higher-order polynomial terms in the model. But it is worth noting that each of the ten curves result in a **different test MSE estimate** for each of the ten regression models considered. And there is no consensus among the curves as to which model results in the smallest validation set MSE.

mse.df.2 <- tibble(  
 sample = vector("integer", 100),  
 degree = vector("integer", 100),  
 mse = vector("double",100)  
)  
  
counter <- i  
  
for (i in 1:10){  
 set.seed(i)  
 sample <- sample(c(TRUE, FALSE), nrow(auto), replace=TRUE, prob=c(0.6,0.4))  
 train <- auto[sample, ]  
 test <- auto[!sample,]  
   
 # modeling  
 for (j in 1:10){  
 lm.fit <- lm(mpg ~ poly(horsepower, j), data = train)  
   
 # add degree & mse values  
 mse.df.2[counter,2] <- j  
 mse.df.2[counter, 3] <- mean((test$mpg - predict(lm.fit, test))^2)  
   
 # add sample identifier  
 mse.df.2[counter,1] <- i  
 counter <- counter +1  
 }  
 next  
}  
  
mse.df.2 %>%   
 ggplot(aes(degree, mse, color=factor(sample)))+  
 geom\_line(show.legend = FALSE)+  
 geom\_point(show.legend = FALSE)+  
 ylim(c(10,30))



### Leave-one-out cross-validation

Leav-one-out cross-validation (LOOCV) is closely related to the validation set approach as it involves splitting the set of observations into two parts. However, instead of creating two subsets of comparable size (i.e., 60% training, 40% validation), a single observation () is used for the validation set, and the remaining observations (…) make up the training set.

The statistical learning method is fit on the training observations, and a prediction is made for the excluded observation. since the validation observation was not used in the fitting process, the estimate error provides and approximately unbiased estimate for the test error. But even though is unbiased for the test error, it is a poor estimate because it is highly variable, since it is based on a single observation.

However, we can repeat the procedure by selecting a difference row for the validation data, training the statistical learning procedure on the other observations and computing rror . We can repeate this approach n times, where each time we holdout a different, single observation to validate on. This produces a total of n squared errors, . The LOOCV estimate for the test MSE is the average of these test error estimates.

To peform this procedure in R, we first need to understand an important nuance. In the logistic regression tutorial, we used the glm function to perform logistic regression by passing in the family = "binomial" argument. But if we use glm to fit a model without passing in the family argument, then it performs linear regression, just like the lm function. So, for instance:

glm.fit <- glm(mpg ~ horsepower, data = auto)  
coef(glm.fit)  
## (Intercept) horsepower   
## 39.9358610 -0.157844

is the same as

lm.fit <- lm(mpg ~ horsepower, data = auto)  
coef(lm.fit)  
## (Intercept) horsepower   
## 39.9358610 -0.1578447

Why is this important? Because we can perform LOOCV for any generalized linear model using glm and the cv.glm function from the [boot](http://cran.r-project.org/web/packages/boot/index.html) package. boot provides extensive facilities for bootstrapping and related resampling methods. You can bootstrap a single statistic (e.g. a median), a vector (e.g., regression weights), or as you’ll see in this tutorial perform cross-validation. To perform LOOCV for a given generalized linear model we simply:

1. fit our model across the entire data set with glm
2. feed the entire data set and our fitted model into cv.glm

# step 1: fit model  
glm.fit <- glm(mpg ~ horsepower,   
 data = auto)  
  
# step 2 perform LOOCV across entire data set  
loocv.err <- cv.glm(auto, glm.fit)  
loocv.err %>%   
 str()  
## List of 4  
## $ call : language cv.glm(data = auto, glmfit = glm.fit)  
## $ K : num 392  
## $ delta: num [1:2] 24.2 24.2  
## $ seed : int [1:626] 10403 392 -1703707781 1994959178 434562476 -1277611857 -1105401243 1020654108 526650482 -1538305299 ...  
## List of 4  
## $ call : language cv.glm(data = auto, glmfit = glm.fit)  
## $ K : num 392  
## $ delta: num [1:2] 24.2 24.2  
## $ seed : int [1:626] 403 392 -1703707781 1994959178 434562476 -1277611857 -1105401243 1020654108 526650482 -1538305299 ...

cv.glm provide a list with 4 outputs:

1. call: the oroginal function call
2. K: the number of *folds* used. In our case, it is 392 because the LOOCV looped through and pulled out each observation at least once to use a test observation.
3. delta: the CV estimate of prediction error. The first number, which is the primary number we care about, is the output from Eq.1 listed above.
4. seed: the values of the random seed used for the function call

The result we primarily care about is the corss-validation estimate of test error (Eq.1). our cross-validation estimate for the test error is approximately 24.23. This estimate is far less biased estimate of the test error compared to our single test MSE produced by a training - testing validation approach.

loocv.err$delta[1]  
## [1] 24.23151

We can repeat this procedure to estiamte an unbiased MSE across multiple model fits. For example, to assess multiple polynomial fits (as we did above) to identify the one that represents the best fit we can integrate this procedure into a function.

Here we develop a function that computes the LOOCV MSE based on specified polynomial degree. We then feed this function (via map\_dbl) values 1-5 to compute the first through fifth polynomials.

# create function that computes LOOCV MSE on specified polynomial degree  
loocv\_error <- function(x){  
 glm.fit <- glm(mpg ~ poly(horsepower, x), data = auto)  
 cv.glm(auto, glm.fit)$delta[1]  
}  
  
# compute LOOCV MSE for polynomial degrees 1-5  
library(purrr)  
1:5 %>%   
 map\_dbl(loocv\_error) %>%   
 tibble()  
## # A tibble: 5 x 1  
## .  
## <dbl>  
## 1 24.2  
## 2 19.2  
## 3 19.3  
## 4 19.4  
## 5 19.0

Our results illustrate a sharp drop in the estimated test MSE between the linear and quadratic fits, but then no clear improvement from using higher-order polynomials. Thus, our unbiased MSEs suggest that using a 2nd polynomial (quadratic fit) is likely the optimal model balancing interpretation and low test errors.

This LOOCV approach can be used with any kind of predictive modeling. For example, we could use it with logistic regression or linear disciminant analysis. Unfortunately, this can be very time consuming approach if is large, you are trying to loop through many models (1-10 polynomials), and if each individual model is slow to fit. For example, if we wanted to perform this approach on the ggplot2::diamonds data set for a linear regression model, which contains 53,940 observations, the comutation time is nearly 30 minutes.

# DO NOT RUN THIS CODE - YOU WILL BE WAITING A LONG TIME!!  
# system.time({  
# diamonds.fit <- glm(price ~ carat + cut + color + clarity, data = diamonds)  
# cv.glm(diamonds, diamonds.fit)  
# })  
# user system elapsed   
# 1739.041 285.496 2035.062

### K-Fold Cross Validation

An alternative to LOOCV is the k-fold cross validation approach. This resamling method involves randomly dividing the data into k groups (aka folds) of approximately equal size.

The first fold is treated as a validation set, and the statistical method is fit on the remaining data. The mean squared error, is then computed on the observations in the held-out fold. This procedure is repreated k times; each time, a different group of observations is treated as the validation set.

This process results in k estimates of the test error, . Thus, the k fold CV is computed by averaging these values:

It is not hard to see that LOOCV is a special case of *k*-fold approach which *k* is set to equal *n*. However, using the *k*-fold approach, one typically use or . This can substantially reduce the computation burden of LOOCV.

Furthermore, there has been sufficient empirical evidence that demonstrates using 5-10 folds yild surprisingly accurate test error rate estimates (see chapter 5 of ISLR for model details).

We can implement the *k*-fold approach just as we did with the LOOCV approach. The only difference is incorporating the K=10 argument that we include in the cv.glm function.

Below illustrates our *k*-fold MSE values for the different polynomial models on our auto data. When compared to the LOOCV outputs we see that the results are nearly identical.

# create functions that computes k-fold MSE based on specified polynomial degree  
  
kfcv\_error <- function(x){  
 glm.fit <- glm(mpg ~ poly(horsepower, x), data = auto)  
 cv.glm(auto, glm.fit, K=10)$delta[1]  
}  
  
# compute k-fold MSE for polynomial degree 1-5  
1:5 %>% map\_dbl(kfcv\_error)  
## [1] 24.24098 19.16217 19.17898 19.43967 19.11506  
  
# compare to LOOCV MSE values  
1:5 %>% map\_dbl(loocv\_error)  
## [1] 24.23151 19.24821 19.33498 19.42443 19.03321

We can also illustrate the computational advantage of the k-fold approach. As we saw, using LOOCV on the diamonds data set took nearly 30 minutes whereas using the k-fold approach only takes about 4 seconds.

system.time({  
 diamonds.fit <- glm(price ~ carat + cut + color + clarity, data = diamonds)  
 cv.glm(diamonds, diamonds.fit, K = 10)  
})  
## user system elapsed   
## 2.61 0.19 2.85  
## user system elapsed   
## 3.760 0.564 4.347

We can apply this same approach to classification problems as well. For example, in the previous tutorial we compared the performance of a logistic regression, linear discriminant analysis (LDA), and quadratic discriminant analysis (QDA) on some stock market data using the traditional training vs. testing (60%/40%) data splitting approach. We could’ve performed the same assessment using cross validation. In the classification setting, the LOOCV error rate takes the form

where . The k-fold CV error rate and validation set error rates are defined analogously.

Consequently, for the logistic regression, we use cv.glm to perform a fold cross validation. The end result is an estimated CV error of .5 (Note: since the response variable is binary we incorporate a new cost function to compute the estimated error in Eq.3).

stock <- ISLR::Smarket  
  
# The cost function here correlates to that in Equ.3   
glm.fit <- glm(Direction ~ Lag1 + Lag2, family = binomial, data = stock)  
  
# The cost function here correlates to that in Eq.3  
cost <- function(r, pi = 0) {mean(abs(r - pi) > 0.5)}  
  
# compute the k-fold estimated error with our cost function  
cv.glm  
## function (data, glmfit, cost = function(y, yhat) mean((y - yhat)^2),   
## K = n)   
## {  
## call <- match.call()  
## if (!exists(".Random.seed", envir = .GlobalEnv, inherits = FALSE))   
## runif(1)  
## seed <- get(".Random.seed", envir = .GlobalEnv, inherits = FALSE)  
## n <- nrow(data)  
## if ((K > n) || (K <= 1))   
## stop("'K' outside allowable range")  
## K.o <- K  
## K <- round(K)  
## kvals <- unique(round(n/(1L:floor(n/2))))  
## temp <- abs(kvals - K)  
## if (!any(temp == 0))   
## K <- kvals[temp == min(temp)][1L]  
## if (K != K.o)   
## warning(gettextf("'K' has been set to %f", K), domain = NA)  
## f <- ceiling(n/K)  
## s <- sample0(rep(1L:K, f), n)  
## n.s <- table(s)  
## glm.y <- glmfit$y  
## cost.0 <- cost(glm.y, fitted(glmfit))  
## ms <- max(s)  
## CV <- 0  
## Call <- glmfit$call  
## for (i in seq\_len(ms)) {  
## j.out <- seq\_len(n)[(s == i)]  
## j.in <- seq\_len(n)[(s != i)]  
## Call$data <- data[j.in, , drop = FALSE]  
## d.glm <- eval.parent(Call)  
## p.alpha <- n.s[i]/n  
## cost.i <- cost(glm.y[j.out], predict(d.glm, data[j.out,   
## , drop = FALSE], type = "response"))  
## CV <- CV + p.alpha \* cost.i  
## cost.0 <- cost.0 - p.alpha \* cost(glm.y, predict(d.glm,   
## data, type = "response"))  
## }  
## list(call = call, K = K, delta = as.numeric(c(CV, CV + cost.0)),   
## seed = seed)  
## }  
## <bytecode: 0x0000000017bce1c0>  
## <environment: namespace:boot>  
cv.glm(stock, glm.fit, cost, K = 10)$delta[1]  
## [1] 0.4912

To performm cross validation with our LDA and QDA models we use a slightly different approach. Both the lda and qda functions have built-in cross validation arguments. Thus, setting CV = TRUE within these functions will result in a LOOCV execution and the class and posterior probabilities are a product of this cross validation.

library(MASS)  
  
# fit discirimant analysis modles with cv=TRUE for LOOCV  
lda.fit <- lda(Direction ~ Lag1+Lag2, CV=TRUE, data=stock)  
qda.fit <- qda(Direction ~ Lag1+Lag2, CV=TRUE, data=stock)  
  
# compute estimated test error based on CV  
mean(lda.fit$class != stock$Direction)  
## [1] 0.4816  
## [1] 0.4816  
mean(qda.fit$class != stock$Direction)  
## [1] 0.4872  
## [1] 0.4872

Thus, the results are similar to what we saw in the previous tutorial, none of these models do an exceptional (or even decent) job! However, we see that the LOOCV estimated error for the QDA model (.487) is fairly higher than what we saw in the train-test approach(.40). This suggests our previous QDA model with the train-test validation approach may have been a bit optimistically biased.

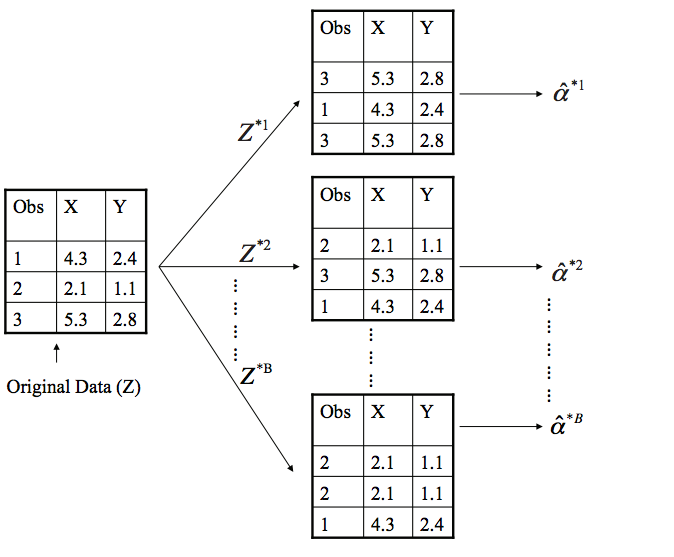
### Bootstrap

*Bootstrapping* is widely applicable and extremely powerful statistical tool that can be used to quantify the uncertainty associated with a given estimator or statistical leanring method.

As a simple example, bootstrapping can be used to estimate the standard errors of the coefficients from a linear regression fit. In the case of linear regression, this is not particularly useful, since we saw in the linear regression tutorial, that R provides such standard errors automatically. Howefver, the power of the boostrap lies in the fact that it can be easily applied to a wide range of statistical learning methods, including some for which a measure of variability is otherwise difficult to obtain and is not automatically output by statistical software..

In essence bootstrapping repeatedly draws independent samples from our data set to create bootstrap data sets. This sample is performed with replacement, which means that the same observation can be sampled more than once. The figure below from the ISLR1 book depicts the bootsrap approach on a small data set (n = 3).

knitr::include\_graphics("image/bootstrap.png")



Each bootstrap data set () contains n observations, sampled with replacement from the original data set. Each bootstrap is used to compute the estimated statistic we are interested in (). We can then use all the bootstrapped data sets to compute the standard error of desired statistic as

Thus, serves as an estimate of the standard error of estimated from the original data set. Let’s look at how we can implement this in R on a couple of simple examples:

#### Example 1: Estimating the accuracy of a single statistic

Performing a bootstrap analysis in R entails two steps:

1. Create a function that computes the statistic of interest
2. Use the boot function from the boot package to perform the bootstrapping

In this example, we will use the ISLR::Portfolio data set. This data set contains the return for two investment assets (X and Y) Here, our goal is going to be minimizing the risk of investing a fixed sum of money in each asset. Mathematically, we can achieve this by minimizing the variance of our investment using the statistic

Thus, we need to create a function that will compute this test statistic:

statistic <- function(data, index) {  
 x <- data$X[index]  
 y <- data$Y[index]  
 (var(y) - cov(x, y)) / (var(x) + var(y) - 2\* cov(x, y))  
}

Now we compute for a specified subset of our portfolio data:

portfolio <- ISLR::Portfolio  
  
# compute our statistic for all 100 observations  
statistic(portfolio, 1:100)  
## [1] 0.5758321

Next, we can use sample to randomly select 100 observations from the range 1 to 100, with replacement. This is equivalent to constructing a new bootstrap data set and recomputing based on the new data set.

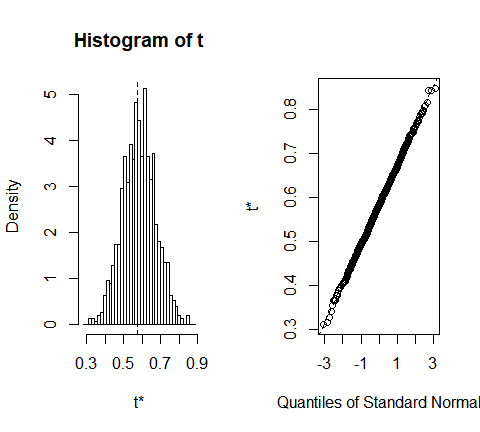
statistic(portfolio, sample(100,100, replace=TRUE))  
## [1] 0.7614692

If you re-ran this function several times, you will see that you are getting a different output each time. What we want to do is run this many times, record our output each time, and then compute a valid standard error of all the outputs. to do this, we can boot and supply it our original data, the function that computes the test statistic, and the number of bootstrap replicates(R).

set.seed(123)  
boot(portfolio, statistic, R = 1000)  
##   
## ORDINARY NONPARAMETRIC BOOTSTRAP  
##   
##   
## Call:  
## boot(data = portfolio, statistic = statistic, R = 1000)  
##   
##   
## Bootstrap Statistics :  
## original bias std. error  
## t1\* 0.5758321 0.007544609 0.08952496  
##   
## ORDINARY NONPARAMETRIC BOOTSTRAP  
##   
##   
## Call:  
## boot(data = portfolio, statistic = statistic, R = 1000)  
##   
##   
## Bootstrap Statistics :  
## original bias std. error  
## t1\* 0.5758321 0.002396754 0.08752118

The final output shows that using the oroginal data, , and it also provides the bootstrap estimate of our standard error . Once we generate the bootstrap estimates we can also view the confidence intervanls with boot.ci and plot our results.

set.seed(123)  
result <- boot(portfolio, statistic, R = 1000)  
  
boot.ci(result, type = "basic")  
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS  
## Based on 1000 bootstrap replicates  
##   
## CALL :   
## boot.ci(boot.out = result, type = "basic")  
##   
## Intervals :   
## Level Basic   
## 95% ( 0.3918, 0.7449 )   
## Calculations and Intervals on Original Scale  
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS  
## Based on 1000 bootstrap replicates  
##   
## CALL :   
## boot.ci(boot.out = result, type = "basic")  
##   
## Intervals :   
## Level Basic   
## 95% ( 0.3958, 0.7376 )   
## Calculations and Intervals on Original Scale  
plot(result)



#### Example 2: Estimating the accuracy of a linear regression model

We can use this same concept to assess the variability of the coefficient estimates and predictions from a statistical learning method such as linear regression. For instance, here we will assess the variability of the estimates for and , the intercept and slope terms for the linear regression model that uses horsepower to predict mpg in our auto data set.

First, we create the function to compute the statistic of interest. We can apply this to our entire data set to get the baseline coefficients.

statistic <- function(data, index){  
 lm.fit <- lm(mpg ~ horsepower, data = data, subset = index)  
 coef(lm.fit)  
}

Now we can inject this into the boot function to compute the bootstrapped standard error estimate:

set.seed(123)  
boot(auto, statistic, 1000)  
##   
## ORDINARY NONPARAMETRIC BOOTSTRAP  
##   
##   
## Call:  
## boot(data = auto, statistic = statistic, R = 1000)  
##   
##   
## Bootstrap Statistics :  
## original bias std. error  
## t1\* 39.9358610 0.0156469811 0.845583773  
## t2\* -0.1578447 -0.0001803022 0.007393556  
##   
## ORDINARY NONPARAMETRIC BOOTSTRAP  
##   
##   
## Call:  
## boot(data = auto, statistic = statistic, R = 1000)  
##   
##   
## Bootstrap Statistics :  
## original bias std. error  
## t1\* 39.9358610 0.0295956008 0.863541674  
## t2\* -0.1578447 -0.0002940364 0.007598619

This indicates that the boostrap estimate for is 0.86, and that the bootstrap estimate for is 0.0076. If we compute these to the standard error provided by the summary function we see a difference.

summary(lm(mpg ~ horsepower, data = auto))  
##   
## Call:  
## lm(formula = mpg ~ horsepower, data = auto)  
##   
## Residuals:  
## Min 1Q Median 3Q Max   
## -13.5710 -3.2592 -0.3435 2.7630 16.9240   
##   
## Coefficients:  
## Estimate Std. Error t value Pr(>|t|)   
## (Intercept) 39.935861 0.717499 55.66 <2e-16 \*\*\*  
## horsepower -0.157845 0.006446 -24.49 <2e-16 \*\*\*  
## ---  
## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1  
##   
## Residual standard error: 4.906 on 390 degrees of freedom  
## Multiple R-squared: 0.6059, Adjusted R-squared: 0.6049   
## F-statistic: 599.7 on 1 and 390 DF, p-value: < 2.2e-16

This difference suggests the standard errors provided by summary may be biased. That is, certain assumptions may be violated which is causing the standard errors in the non-bootstrap approach to be different than those in the bootstrap approach.

If you remember from earlier in the tutorial we found that a quadratic fit appeared to be the most approapriate for the relationship between mpg and horsepower. Lets adjust our code to capture this fit and see if we notice a difference with our outputs.

quad.statistic <- function(data, index){  
 lm.fit <- lm(mpg ~ poly(horsepower, 2), data = data, subset = index)  
 coef(lm.fit)  
}  
  
set.seed(1)  
boot(auto, quad.statistic, 1000)  
##   
## ORDINARY NONPARAMETRIC BOOTSTRAP  
##   
##   
## Call:  
## boot(data = auto, statistic = quad.statistic, R = 1000)  
##   
##   
## Bootstrap Statistics :  
## original bias std. error  
## t1\* 23.44592 -0.003660358 0.2195369  
## t2\* -120.13774 0.002769239 3.6138046  
## t3\* 44.08953 0.101767465 4.1998076  
  
##   
## ORDINARY NONPARAMETRIC BOOTSTRAP  
##   
##   
## Call:  
## boot(data = auto, statistic = quad.statistic, R = 1000)  
##   
##   
## Bootstrap Statistics :  
## original bias std. error  
## t1\* 23.44592 0.003943212 0.2255528  
## t2\* -120.13774 0.117312678 3.7008952  
## t3\* 44.08953 0.047449584 4.3294215

Now if we compare the standard errors between the bootstrap approach and the non-bootstrap approach we see the standard errors align more closely. This better correspondence between the bootstrap estimates and the standard estimates suggests a better model fit.

Thus, bootstrapping provides an additional method for assessing the adequacy of our model’s fit.

summary(lm(mpg ~ poly(horsepower, 2), data = auto))  
##   
## Call:  
## lm(formula = mpg ~ poly(horsepower, 2), data = auto)  
##   
## Residuals:  
## Min 1Q Median 3Q Max   
## -14.7135 -2.5943 -0.0859 2.2868 15.8961   
##   
## Coefficients:  
## Estimate Std. Error t value Pr(>|t|)   
## (Intercept) 23.4459 0.2209 106.13 <2e-16 \*\*\*  
## poly(horsepower, 2)1 -120.1377 4.3739 -27.47 <2e-16 \*\*\*  
## poly(horsepower, 2)2 44.0895 4.3739 10.08 <2e-16 \*\*\*  
## ---  
## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1  
##   
## Residual standard error: 4.374 on 389 degrees of freedom  
## Multiple R-squared: 0.6876, Adjusted R-squared: 0.686   
## F-statistic: 428 on 2 and 389 DF, p-value: < 2.2e-16  
##   
## Call:  
## lm(formula = mpg ~ poly(horsepower, 2), data = auto)  
##   
## Residuals:  
## Min 1Q Median 3Q Max   
## -14.7135 -2.5943 -0.0859 2.2868 15.8961   
##   
## Coefficients:  
## Estimate Std. Error t value Pr(>|t|)   
## (Intercept) 23.4459 0.2209 106.13 <2e-16 \*\*\*  
## poly(horsepower, 2)1 -120.1377 4.3739 -27.47 <2e-16 \*\*\*  
## poly(horsepower, 2)2 44.0895 4.3739 10.08 <2e-16 \*\*\*  
## ---  
## Signif. codes: 0 '\*\*\*' 0.001 '\*\*' 0.01 '\*' 0.05 '.' 0.1 ' ' 1  
##   
## Residual standard error: 4.374 on 389 degrees of freedom  
## Multiple R-squared: 0.6876, Adjusted R-squared: 0.686   
## F-statistic: 428 on 2 and 389 DF, p-value: < 2.2e-16

### Additional Resources

This will get you started with resampling methods; however, understand that there are many approaches for resampling and even more options within R to implement these approaches. The following resources will help you learn more:

* An Introduction to Statistical Learning
* Introduction to Statistics Through Resampling Methods and R
* Applied Predictive Modeling
* Elements of Statistical Learning