Hands-on-ML with R

Koji Mizumura

October 29th,2018 - November 29th,2018

Table of Contents

# Introduction

Machine learning continues to grow in importance for many organizations across nearly all domains. Examples include:

* predicting the likelihood of a patient returning to the hospital (*readmission*) within 30 days of discharge,
* segmenting customers based on common attributes or purchasing behavior for target marketing,
* predicting coupon redemption rates for a given marketing campaign,
* predicting customer churn so an organization can perform preventative intervention,
* and many more!

In essence, these tasks all seek to learn from data. To address each scenario, we use a given set of *features* to train an algorithm and extract insights. These algorithms, or *learners*, can be classified according to the amount and type of supervision provided during training. The two main groups this book focuses on includes: ***supervised learners*** that are used to construct predictive models, and ***unsupervised learners*** that are used to build descriptive models. Which type you will need to use depends on the learning task you hope to accomplish.

## Supervised Learning

A ***predictive model*** is used for tasks that involve the prediction of a given output using other variables and their values (*features*) in the data set. Or as stated by @apm, predictive modeling is *“the process of developing a mathematical tool or model that generates an accurate prediction”* (p. 2). The learning algorithm in a predictive model attempts to discover and model the relationship among the ***target*** response (the variable being predicted) and the other features (aka predictor variables). Examples of predictive modeling include:

* using customer attributes to predict the probability of the customer churning in the next 6 weeks,
* using home attributes to predict the sales price,
* using employee attributes to predict the likelihood of attrition,
* using patient attributes and symptoms to predict the risk of readmission,
* using production attributes to predict time to market.

Each of these examples have a defined learning task. They each intend to use attributes () to predict an outcome measurement ()

Notes: Throughout this text I will use various terms interchangeably for: - : “predictor variables”, “independent variables”, “attributes”, “features”, “predictors” - : “target variable”, “dependent variable”, “response”, “outcome measurement”

The predictive modeling examples above describe what is known as \_supervised learning\_. The supervision refers to the fact that the target values provide a supervisory role, which indicates to the learner the task it needs to learn. Specifically, given a set of data, the learning algorithm attempts to optimize a function (the algorithmic steps) to find the combination of feature values that results in a predicted value that is as close to the actual target output as possible.  
  
Supervised learning problems resolve around two primary themes: regression and classification.  
  
Notes:  
In supervised learning, the training data you feed the algorithm includes the desired solutions. Consequently, the solutions can be used to help \_supervise\_ the training process to find the optimal algorithm parameters.

### Regression problems

When the objective of our supervised learning is to predict a numeric outcome, we refer to this as a ***regression problem*** (not to be confused with linear regression modeling). Regression problems revolve around predicting output that falls on a continuous numeric spectrum. In the examples above predicting home sales prices and time to market reflect a regression problem because the output is numeric and continuous. This means, given the combination of predictor values, the response value could fall anywhere along the continuous spectrum. Figure @ref(fig:regression-problem) illustrates average home sales prices as a function of two home features: year built and total square footage. Depending on the combination of these two features, the expected home sales price could fall anywhere along the plane.

Average home sales price as a function of year built and total square footage.

### classification problems

When the objective of our supervised learning is to predict a categorical response, we refer to this as a ***classification problem***. Classification problems most commonly revolve around predicting a binary or multinomial response measure such as:

* did a customer redeem a coupon (yes/no, 1/0),
* did a customer churn (yes/no, 1/0),
* did a customer click on our online ad (yes/no, 1/0),
* classifying customer reviews:
  + binary: positive vs negative
  + multinomial: extremely negative to extremely positive on a 0-5 Likert scale

However, when we apply machine learning models for classification problems, rather than predict a particular class (i.e. “yes” or “no”), we often predict the *probability* of a particular class (i.e. yes: .65, no: .35). Then the class with the highest probability becomes the predicted class. Consequently, even though we are performing a classification problem, we are still predicting a numeric output (probability). However, the essence of the problem still makes it a classification problem.

### Algorithm comparison guide

Although there are machine learning algorithms that can be applied to **regression problems** but not **classification** and vice versa, the supervised learning algorithms I cover in this book can be applied to both. These algorithms have become the most popular machine learning applications in recent years.

Although the chapters that follow will go into detail on each algorithm, the following provides a quick reference guide that compares and contrasts some of their features. Moreover, I provide recommended base learner packages that I have found to scale well with typical rectangular data analyzed by organizations.

Characteristics

Regularized GLM

Random Forest

Gradient Boosting Machine

Deep Learning

Allows n < p

Provides automatic feature selection

Handles missing values

No feature pre-processing required

Robust to outliers

Easy to tune

Computational speed

Predictive power

Preferred regression base learner

glmnet h2o.glm

ranger h2o.randomForest

xgboost h2o.gbm

keras h2o.deeplearning

Preferred classifciation base learner

glmnet h2o.glm

ranger h2o.randomForest

xgboost h2o.gbm

keras h2o.deeplearning

## Unsupervised learning

***Unsupervised learning***, in contrast to supervised learning, includes a set of statistical tools to better understand and describe your data but performs the analysis without a target variable. In essence, unsupervised learning is concerned with identifying groups in a data set. The groups may be defined by the rows (i.e., *clustering*) or the columns (i.e., *dimension reduction*); however, the motive in each case is quite different.

The goal of ***clustering*** is to segment observations into similar groups based on the observed variables. For example, to divide consumers into different homogeneous groups, a process known as market segmentation. In **dimension reduction**, we are often concerned with reducing the number of variables in a data set. For example, classical regression models break down in the presence of highly correlated features. Dimension reduction techniques provide a method to reduce the feature set to a potentially smaller set of uncorrelated variables. These variables are often used as the input variables to downstream supervised models like.

Unsupervised learning is often performed as part of an exploratory data analysis. However, the exercise tends to be more subjective, and there is no simple goal for the analysis, such as prediction of a response. Furthermore, it can be hard to assess the quality of results obtained from unsupervised learning methods. The reason for this is simple. If we fit a predictive model using a supervised learning technique (i.e. linear regression), then it is possible to check our work by seeing how well our model predicts the response *Y* on observations not used in fitting the model. However, in unsupervised learning, there is no way to check our work because we don窶冲 know the true answer窶杯he problem is unsupervised.

However, the importance of unsupervised learning should not be overlooked and techniques for unsupervised learning are used in organizations to:

* Divide consumers into different homogeneous groups so that tailored marketing strategies can be developed and deployed for each segment.
* Identify groups of online shoppers with similar browsing and purchase histories, as well as items that are of particular interest to the shoppers within each group. Then an individual shopper can be preferentially shown the items in which he or she is particularly likely to be interested, based on the purchase histories of similar shoppers.
* Identify products that have similar purchasing behavior so that managers can manage them as product groups.

These questions, and many more, can be addressed with unsupervised learning. Moreover, often the results of an unsupervised model can be used as inputs to downstream supervised learning models.

## Machine learning interpretability

In his seminal 2001 paper, Leo Breiman popularized the phrase: \_窶徼he multiplicity of good models.窶拈 The phrase means that for the same set of input variables and prediction targets, complex machine learning algorithms can produce multiple accurate models with very similar, but not the exact same, internal architectures.

Figure @ref(fig:error-surface) is a depiction of a non-convex error surface that is representative of the error function for a machine learning algorithm with two inputs 窶? say, a customer窶冱 income and a customer窶冱 age, and an output, such as the same customer窶冱 probability of redeeming a coupon. This non-convex error surface with no obvious global minimum implies there are many different ways complex machine learning algorithms could learn to weigh a customer窶冱 income and age to make a good decision about if they are likely to redeem a coupon. Each of these different weightings would create a different function for making coupon redemption (and therefore marketing) decisions, and each of these different functions would have different explanations.

Non-convex error surface with many local minimas.

All of this is an obstacle to data scientists. On one hand, **different models** can have widely different predictions based on the same feature set. Even models built from the same algorithm but with **different hyperparameters** can lead to different results. Consequently, practitioners should understand how different implementation of algorithms differ, which can cause variance in their results (i.e., a default xgboost model can produce very different results from a default gbm model, even though they both implement gradient boosting machines).

Alternatively, data scientists can experience very similar predictions from different models based on the same feature set. However, these models will have very different logic and structure leading to different interpretations. Consequently, preactitionaer should understand how to interpret different types of models.

This book provide you with a fundamental understanding to compare and contrast models and even package implementations of similar algorithms. Several machine learning interpretability techniques will be demonstrated to help you understand what is driving model and prediction performance. This will allow you to be more effective and efficient in applying and understandin multiple good models.

## The data sets

The **XX** data sets chosen for this book allow us to illustrate the different features of our machine learning algorithms. Since the goal of this book is to demonstrate how to implement R’s ML stack, I make the assumption that you have already spent significant time cleaning and getting to know your data via exploratory data analysis. This would allow you to perform many necessary tasks prior to the ML tasks outlined in this book such as:

* feature selection: removing unnecessary variables and retaining only those variables you wish to include in your modeling process,
* recoding variable names and values so that they are meaningful and interpretable,
* recoding or removing missing values.

Consequently, the exemplar data sets I use throughout this book have, for the most part, gone through the necessary cleaning process. These data sets are all freely available and include:

* Property sales information as described in @de2011ames.
  + **problem type**: supervised regression
  + **response variable**: sale price (i.e. $195,000, $215,000)
  + **features**: 80
  + **observations**: 2,930
  + **objective**: use property attributes to predict the sale price of a home
  + **access**: provided by the AmesHousing package [@R-ames]
  + **more details**: See ?AmesHousing::ames\_raw

# access data  
ames <- AmesHousing::make\_ames()  
  
# initial dimension  
dim(ames)  
## [1] 2930 81  
  
# response variable  
head(ames$Sale\_Price)  
## [1] 215000 105000 172000 244000 189900 195500

PK ] 8t}\_6-n&P9[!GΣ<U+0081>1s,U\*@z KKw=“XL<c5><e4>\*<U+00B1><U+0598><U+00BC><ec><e8>s<eb>K<e3>,yj<fb>j<U+00AB><U+009C> <U+FD52><U+00A9><U+031B>\*?:@<U+01DE><U+024D>-<ff><e0><U+00A5>?hM6=q<U+00A5>|<U+00BC>=X<U+00BC>SA<U+0095><U+0090><cc>D<c0>7aH<U+008F>o](<f9><U+0080>H<U+00B1><U+012C><U+009F><U+DDE2>\*%<U+00A1>trmh$<U+00AB>}pb$=<U+00A3><U+00B3>\_<e2><c3>$vm<U+00A8><f7><f2><U+009D><f4>0E<U+00BC>Z<f1><e4>$<U+0090><f6><da><e7><f8>l<U+008E>Ff<U+0633>"<U+0087><U+00B9>Xh<U+00B8><fc><c1>;<e9>^ <U+009A><U+009F><e7>#'<U+00B7><U+00B3><U+00A0><U+00BE>I%<U+0094>)<U+0081>x<U+00A8><U+00A0>?<f6><ce>\<U+00BA><f0><U+008F>w<U+00AF><U+009E>><U+0572><dd>6<dc>k<U+00B8><U+01AE>5<U+00BA><c3><fe><U+009D><c2><c5> <U+00BA><U+008A><U+029E><U+00B2>~<U+00AD><c3>kd<d1>Iw<U+00BC><f9><eb><df>PK <U+0083><U+00B7>|<U+0097><ec>s<e0><U+00A8>GF<fc>JT2<c9><c8><da><c1>k<U+0087><ee><U+00B3><dc>T,<<U+00AE>s<f5><U+009F>:<U+00BC>W<U+008E><U+008E><dd>,K<ed><f5>\<U+00BE><U+009D><U+00AA><ce>}-<U+00A4><U+009C>O\*]<U+009E><U+00AF><f4><f7><f6>1<U+00B0><U+0092>#%<U+00B4>I<f8><U+00B4><d0>G<e2><cb><U+00A1>PK \* Employee attrition information originally provided by [IBM Watson Analytics Lab](https://www.ibm.com/communities/analytics/watson-analytics-blog/hr-employee-attrition/). - \_\_problem type\_\_: supervised binomial classification - \_\_response variable\_\_:Attrition(i.e. "Yes", "No") - \_\_features\_\_: 30 - \_\_observations\_\_: 1,470 - \_\_objective\_\_: use employee attributes to predict if they will attrit (leave the company) - \_\_access\_\_: provided by thersamplepackage [@R-rsample] - \_\_more details\_\_: See?rsample::attrition`

# access data  
attrition <- rsample::attrition  
  
# initial dimension  
dim(attrition)  
## [1] 1470 31  
  
# response variable  
head(attrition$Attrition)  
## [1] Yes No Yes No No No   
## Levels: No Yes

* Image information for handwritten numbers originally presented to AT&T Bell Lab窶冱 to help build automatic mail-sorting machines for the USPS. Has been used since early 1990s to compare machine learning performance on pattern recognition (i.e. @lecun1990handwritten; @lecun1998gradient; @cirecsan2012multi).
  + **Problem type**: supervised multinomial classification
  + **response variable**: V785 (i.e. numbers to predict: 0, 1, …, 9)
  + **features**: 784
  + **observations**: 60,000 (train) / 10,000 (test)
  + **objective**: use attributes about the “darkness” of each of the 784 pixels in images of handwritten numbers to predict if the number is 0, 1, …, or 9.
  + **access**: see the code chunk that follows for download instructions
  + **more details**: See [online MNIST documentation](http://yann.lecun.com/exdb/mnist/)

# load training data https://h2o-public-test-data.s3.amazonaws.com/bigdata/laptop/mnist/train.csv.gz  
  
train <- data.table::fread("C:/Users/kojikm.mizumura/Desktop/Data Science/Hands-on ML/data/mnist\_train.csv", data.table = FALSE)  
  
# load test data https://h2o-public-test-data.s3.amazonaws.com/bigdata/laptop/mnist/test.csv.gz  
  
test <- data.table::fread("C:/Users/kojikm.mizumura/Desktop/Data Science/Hands-on ML/data/mnist\_test.csv", data.table = FALSE)  
  
# initial dimension  
dim(train)  
  
# response variable  
head(train$V785)

**TODO**: get unsupervised data sets for clustering and dimension reduction examples

# Supervised Learning

## Preparing for Supervised Machine Learning

Machine learning is a very iterative process. If performed and interpreted correctly, we can have great confidence in our outcomes. If not, the results will be useless. Approaching machine learning correctly means approaching it strategically by spending our data wisely on learning and validation procedures, properly pre-processing variables, minimizing data leakage, tuning hyperparameters, and assessing model performance. Before introducing specific algorithms, this chapter introduces concepts that are commonly required in the supervised machine learning process and that you’ll see briskly covered in each chapter.

## Prerequisites

library(rsample)  
library(caret)  
library(h2o)  
library(dplyr)  
  
# turn off progress bars  
h2o.no\_progress()  
  
# launch h2o  
h2o.init()  
##   
## 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\Rtmpcru90T/h2o\_KojiKM\_Mizumura\_started\_from\_r.out  
## C:\Users\KOJIKM~1.MIZ\AppData\Local\Temp\Rtmpcru90T/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 447 milliseconds   
## H2O cluster timezone: Asia/Tokyo   
## H2O data parsing timezone: UTC   
## H2O cluster version: 3.20.0.8   
## H2O cluster version age: 2 months and 11 days   
## H2O cluster name: H2O\_started\_from\_R\_KojiKM.Mizumura\_uwi630   
## H2O cluster total nodes: 1   
## H2O cluster total memory: 1.96 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 3.5.0 (2018-04-23)

To illustrate some of the concepts, we will use the Ames Housing data and employee attrition data introduced in Chapter @ref(intro). Throughout this book, I’ll demonstrate approaches with regular data frames. However, since many of the supervised machine learning chapters leverage the **h2o** package, we’ll also show how to do some of the tasks with H2O objects. This requires your data to be in an H2O object, which you can convert any data frame easily with as.h2o.

PK ] 8t}\_6-n&P9[!GΣ<U+0081>1s,U\*@z KKw=“XL<c5><e4>\*<U+00B1><U+0598><U+00BC><ec><e8>s<eb>K<e3>,yj<fb>j<U+00AB><U+009C> <U+FD52><U+00A9><U+031B>\*?:@<U+01DE><U+024D>-<ff><e0><U+00A5>?hM6=q<U+00A5>|<U+00BC>=X<U+00BC>SA<U+0095><U+0090><cc>D<c0>7aH<U+008F>o](<f9><U+0080>H<U+00B1><U+012C><U+009F><U+DDE2>\*%<U+00A1>trmh$<U+00AB>}pb$=<U+00A3><U+00B3>\_<e2><c3>$vm<U+00A8><f7><f2><U+009D><f4>0E<U+00BC>Z<f1><e4>$<U+0090><f6><da><e7><f8>l<U+008E>Ff<U+0633>"<U+0087><U+00B9>Xh<U+00B8><fc><c1>;<e9>^ <U+009A><U+009F><e7>#'<U+00B7><U+00B3><U+00A0><U+00BE>I%<U+0094>)<U+0081>x<U+00A8><U+00A0>?<f6><ce>\<U+00BA><f0><U+008F>w<U+00AF><U+009E>><U+0572><dd>6<dc>k<U+00B8><U+01AE>5<U+00BA><c3><fe><U+009D><c2><c5> <U+00BA><U+008A><U+029E><U+00B2>~<U+00AD><c3>kd<d1>Iw<U+00BC><f9><eb><df>PK <U+0083><U+00B7>|<U+0097><ec>s<e0><U+00A8>GF<fc>JT2<c9><c8><da><c1>k<U+0087><ee><U+00B3><dc>T,<<U+00AE>s<f5><U+009F>:<U+00BC>W<U+008E><U+008E><dd>,K<ed><f5>\<U+00BE><U+009D><U+00AA><ce>}-<U+00A4><U+009C>O\*]<U+009E><U+00AF><f4><f7><f6>1<U+00B0><U+0092>#%<U+00B4>I<f8><U+00B4><d0>G<e2><cb><U+00A1>PK <U+0098>!<U+00A9><U+0099>!|+<U+0084><c9>F8| ZNmqLqx-<cd><d7><U+076D><U+00AE>x\*AZ>70鸙mH;\_a9+hYsL[d3|O0Y`6k}{Hwb!@i<c4>zGA/9 Sb=Xo[aB=OwHaEW%4:|;RFLQA kLHC}Rb#bF<$jC=ZV2b[qh2J@Kx<U+0094>A[#;Zζ1No

syVY>=(I#

OPK

# ames data  
library(tidyverse)  
ames <- AmesHousing::make\_ames()  
ames.h2o <- as.h2o(ames)  
  
# attrition data  
churn <- rsample::attrition %>%   
 mutate\_if(is.ordered, factor, ordered = FALSE)  
churn.h2o <- as.h2o(churn)

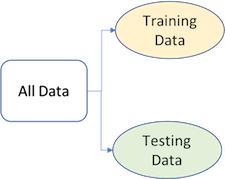
## data splitting

### Spending our data wisely

A major goal of the machine learning process is to find an algorithm that most accurately predicts future values () based on a set of inputs (). In other words, we want an algorithm that not only fits well to our past data, but more importantly, one that predicts a future outcome accurately. This is called the ***generalizability*** of our algorithm. How we *“spend”* our data will help us understand how well our algorithm generalizes to unseen data.

To provide an accurate understanding of the generalizability of our final optimal model, we split our data into training and test data sets:

* **Training Set**: these data are used to train our algorithms and tune hyper-parameters.
* **Test Set**: having chosen a final model, these data are used to estimate its prediction error (generalization error). These data should *not be used during model training!*



Splitting data into training and test sets.

Given a fixed amount of data, typical recommendations for splitting your data into training-testing splits include 60% (training) - 40% (testing), 70%-30%, or 80%-20%. Generally speaking, these are appropriate guidelines to follow; however, it is good to keep in mind that as your overall data set gets smaller,

* spending too much in training () won’t allow us to get a good assessment of predictive performance. We may find a model that fits the training data very well, but is not generalizable (overfitting),
* sometimes too much spent in testing () won’t allow us to get a good assessment of model parameters

In today’s data-rich environment, typically, we are not lacking in the quantity of observations, so a 70-30 split is often sufficient. The two most common ways of splitting data include ***simple random sampling*** and ***stratified sampling***.

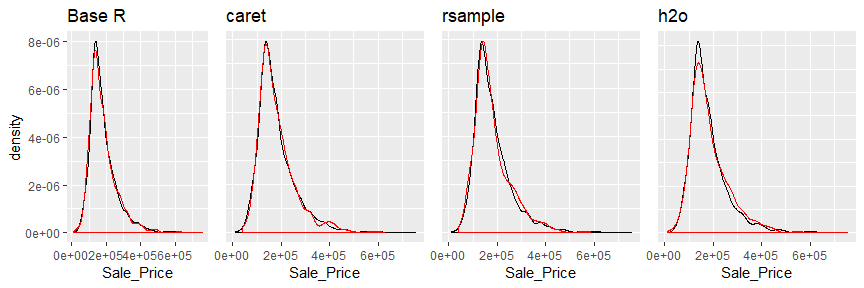
### simple random sampling

The simplest way to split the data into training and test sets is to take a simple random sample. This does not control for any data attributes, such as the percentage of data represented in your response variable (). There are multiple ways to split our data. Here we show four options to produce a 70-30 split (note that setting the seed value allows you to reproduce your randomized splits):

PK ] 8t}\_6-n&P9[!GΣ<U+0081>1s,U\*@z KKw=“XL<c5><e4>\*<U+00B1><U+0598><U+00BC><ec><e8>s<eb>K<e3>,yj<fb>j<U+00AB><U+009C> <U+FD52><U+00A9><U+031B>\*?:@<U+01DE><U+024D>-<ff><e0><U+00A5>?hM6=q<U+00A5>|<U+00BC>=X<U+00BC>SA<U+0095><U+0090><cc>D<c0>7aH<U+008F>o](<f9><U+0080>H<U+00B1><U+012C><U+009F><U+DDE2>\*%<U+00A1>trmh$<U+00AB>}pb$=<U+00A3><U+00B3>\_<e2><c3>$vm<U+00A8><f7><f2><U+009D><f4>0E<U+00BC>Z<f1><e4>$<U+0090><f6><da><e7><f8>l<U+008E>Ff<U+0633>"<U+0087><U+00B9>Xh<U+00B8><fc><c1>;<e9>^ <U+009A><U+009F><e7>#'<U+00B7><U+00B3><U+00A0><U+00BE>I%<U+0094>)<U+0081>x<U+00A8><U+00A0>?<f6><ce>\<U+00BA><f0><U+008F>w<U+00AF><U+009E>><U+0572><dd>6<dc>k<U+00B8><U+01AE>5<U+00BA><c3><fe><U+009D><c2><c5> <U+00BA><U+008A><U+029E><U+00B2>~<U+00AD><c3>kd<d1>Iw<U+00BC><f9><eb><df>PK <U+0083><U+00B7>|<U+0097><ec>s<e0><U+00A8>GF<fc>JT2<c9><c8><da><c1>k<U+0087><ee><U+00B3><dc>T,<<U+00AE>s<f5><U+009F>:<U+00BC>W<U+008E><U+008E><dd>,K<ed><f5>\<U+00BE><U+009D><U+00AA><ce>}-<U+00A4><U+009C>O\*]<U+009E><U+00AF><f4><f7><f6>1<U+00B0><U+0092>#%<U+00B4>I<f8><U+00B4><d0>G<e2><cb><U+00A1>PK B<f7><c4>I<U+0083><U+00A7><U+0097><U+00A2><c0>n<da>=<U+0080>,<U+04F1>0I(9^<de>~<U+008C><ff>t<e8><U+00B0>lAz<U+00B1> <U+0090><fc>}<fa>H≫<U+01DF><de><c1> 9Y <U+0095><U+06AE>7 0jl8V<ea><fb><eb><f3><ea><U+00AB>z<dc><da> eC<U+00A6><f7>2HAH<e5>P<U+00A9>.<e7>XE2z<U+009D><d6>1H<U+00AC>%<f6>:<U+02D5><U+008F><c5>@<dc>D&<U+0083>) <U+03FB><e2>a<U+00B3><f9>Rxm<U+0083><U+009A>1<fe><U+00B5><U+00AD>5<f8>4?nFOiQzZykyee>\*N-yvsBa=T ?MAE9\_8~</]-|ЬL3na6XO/G’:h

# baseR  
set.seed(123)  
index\_1 <- sample(1:nrow(ames), round(nrow(ames)\*0.7))  
train\_1 <- ames[index\_1,]  
test\_1 <- ames[-index\_1,]  
  
dim(train\_1)  
## [1] 2051 81  
  
# caret package  
set.seed(123)  
library(caret)  
library(AmesHousing)  
index\_2 <- createDataPartition(ames$Sale\_Price, p=0.7, list=FALSE)  
train\_2 <- ames[index\_2,]  
test\_2 <- ames[-index\_2,]  
  
# rsample package  
library(rsample)  
split\_1 <- initial\_split(ames,prop=0.7)  
train\_3 <- training(split\_1)  
test\_3 <- testing(split\_1)  
  
# h2o package  
library(h2o)  
split\_2 <- h2o.splitFrame(ames.h2o, ratios = 0.7, seed = 123)  
train\_4 <- split\_2[[1]]  
test\_4 <- split\_2[[2]]

Since this samplint approach will randomly sample across the distribution of (Sale\_Price in our example), you will typically result in a similar distribution between your training and test sets as illustrated below.



Training (black) vs. test (red) distribution.

### Stratified sampling

However, if we want to explicitly control our sampling so that our training and test sets have similar distributions, we can use **stratified sampling**. This is more common with classification problems where the response variable may be imbalanced (90% of observations with response “Yes” and 10% with response “No”).

However, we can als apply to regression problems for data sets that have a small sample size and where the response variable diviates strongly from normality. With a continuous response variable, **stratified sampling** will break down into quartiles and randomly sample from each quantile. Consequently, this will help ensure a balanced representation of the response distribution in both training and test sets.

The easiest way to perform **stratified sampling** on a response variable is to use the **rsample** package, where you specify the response variable to stratafy. The following illustrates that in our original employee attrition data we have an imbalanced response (No:84%, Yes:16%). By enforcing stratified sampling both our training and testing sets have approximately equal response distributions.

# original response distribution  
table(churn$Attrition) %>% prop.table()  
##   
## No Yes   
## 0.8387755 0.1612245  
  
# stratified sampling with rsample package  
set.seed(123)  
  
split\_strata <- initial\_split(churn, prop=0.7, strata="Attrition")  
train\_strat <- training(split\_strata)  
test\_strat <- testing(split\_strata)  
  
# consistent response ratio between train & test  
table(train\_strat$Attrition) %>% prop.table()  
##   
## No Yes   
## 0.838835 0.161165  
table(test\_strat$Attrition) %>% prop.table()  
##   
## No Yes   
## 0.8386364 0.1613636

### Feature engineering

**Feature engineering** generally refers to the process of adding, deleting, and transforming the variables to be applied to your machine learning algorithms. Feature engineering is a significant process and requires you to speed substantial time understanding your data… or as Leo Breiman said *“live with your data before you plunge into modeling.”*

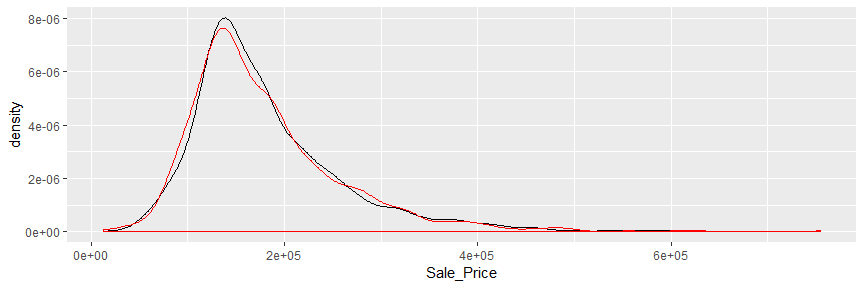
Although this book primarily focused on applying machine learning algorithms, feature engineering can make or break an algorithm’s predictive ability. We will not cover all the potential ways of implementing feature engineering: however, we will cover a few fundamental pre-processing tasks that can significantly improve modeling performance.

To learn more about feature engineering check out [Feature Engineering for Machine Learning](http://shop.oreilly.com/product/0636920049081.do) by @zheng2018feature and Max Kuhn’s upcoming book [Feature Engineering and Selection: A Practical Approach for Predictive Models](http://www.feat.engineering/).

### Response transformation

Although not a requirement, normalizing the distribution of the response variable by using a *transformation* can lead to a big improvement, especially for parametric models. As we saw in the data splitting section, our response variable Sale\_Price is right skewed.

ggplot(train\_1, aes(x=Sale\_Price))+  
 geom\_density(trim=T)+  
 geom\_density(data=test\_1, trim=T, col="red")



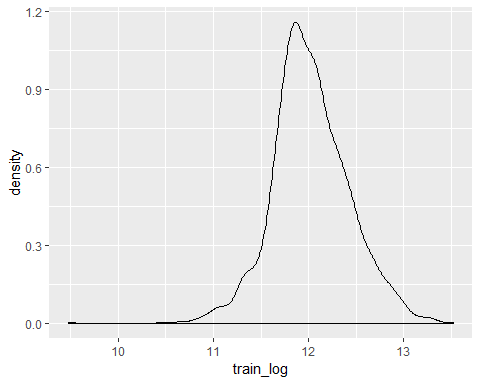
Right skewed response variable.

To normalize, we have a few options:

**Option 1**: normalize with alog transformation. This will transform most right skewed distributions to be approximately normal.

# log transformation  
train\_log\_y <- log(train\_1$Sale\_Price)  
test\_log\_y <- log(test\_1$Sale\_Price)  
  
library(dplyr)  
train\_log\_y %>% as\_tibble() %>% dim()  
## [1] 2051 1

train\_log\_y %>%   
 as.tibble() %>%   
 rename(train\_log=value) %>%   
 ggplot(aes(train\_log))+  
 geom\_density()



IF your response has negative values that a log transformation will produce NaNs. IF these neative values are small (between -0.99 to 0) then you can apply log1p which adds 1 to the value prior to applying a log transformation. If your data consists of negative equal to or less than -1, use the **Yeo Johnson transformation** mentioned next.

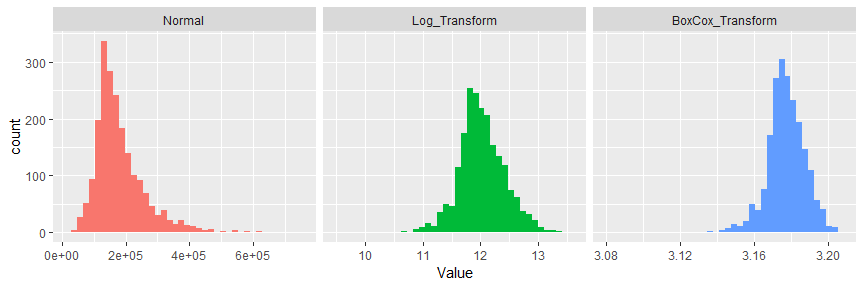
log(-.5)  
## [1] NaN  
log1p(-.5)  
## [1] -0.6931472

**Option 2**: Use a **Box Cox Transformation**. A Box Cox transformation is more flexibile and will find the transformation from a family of [power transforms](https://en.wikipedia.org/wiki/Power_transform#Box%E2%80%93Cox_transformation) that will transform the variable as close as possible to a normal distribution.

Be sure to compute the lamda on the training set and apply the same lambda to both the training and test set to minimize data leakage.

# Box Cox transformation  
library(forecast)  
  
lambda <- forecast::BoxCox.lambda(train\_1$Sale\_Price)  
train\_bc\_y <- forecast::BoxCox(train\_1$Sale\_Price, lambda)  
test\_bc\_y <- forecast::BoxCox(test\_1$Sale\_Price, lambda)

We can see that in this example, the log transformation and Box Cox both do about equally well in transforming our response variable to be normally distributed.



Response variable transformations.

Note that when your model with a transformed response variable, your predictions will also be in the transformed value. You will likely want to re-transform your predicted values to their normal state so that decision-makers can interpret the results. The following code can do this for your:

# log transform a value  
y <- log(10)  
  
# re-transforming the log-transformed value  
exp(y)  
## [1] 10  
  
# Box Cox transform a value  
y <- forecast::BoxCox(10,lambda)  
  
# Inverse box cox function  
inv\_box\_cox <- function(x,lambda){  
 if(lambda==0) exp(x)  
 else (lambda\*x+1)^(1/lambda)  
}  
  
# re-transforming the box cox-transformed values  
inv\_box\_cox(y, lambda)  
## [1] 10  
## attr(,"lambda")  
## [1] -0.3067918

Notes: If your response has negative values, you can use the Yeo-Johnson transformation. To apply, use car::powerTransform to identify the lambda, car::yjPower to apply the transformation, and VGAM::yeo.johnson to apply the transformation and/or the inverse transformation.

### Predictor transformation

### One-hot encoding

Many models requrire all predictor variables to be numeric. Consequently, we need to transform any categorical variables into numeric representations so that these algorithms can compute. Some packages automate this process (i.e., h2o, glm, caret), while others do not (i.e., glmnet, keras). Furthermore, there are many ways to encode categorical variables as numeric representations (i.e., one-hot, ordinal, binary, sum, Helmert).

The most common is referred to as one-hot encoding, where we transpose our categorical variables so that each level of the feature is represented as a boolean value. For example, one-hot encoding variable x in the following.

set.seed(123)  
ex1 <- data.frame(id=1:8, x=sample(letters[1:3],8, replace=TRUE))  
knitr::kable(ex1)

|  |  |
| --- | --- |
| id | x |
| 1 | a |
| 2 | c |
| 3 | b |
| 4 | c |
| 5 | c |
| 6 | a |
| 7 | b |
| 8 | c |

results in the following representation:

one\_hot <- dummyVars(~., ex1, fullRank=F)  
ex2 <- predict(one\_hot, ex1)  
knitr::kable(ex2)

|  |  |  |  |
| --- | --- | --- | --- |
| id | x.a | x.b | x.c |
| 1 | 1 | 0 | 0 |
| 2 | 0 | 0 | 1 |
| 3 | 0 | 1 | 0 |
| 4 | 0 | 0 | 1 |
| 5 | 0 | 0 | 1 |
| 6 | 1 | 0 | 0 |
| 7 | 0 | 1 | 0 |
| 8 | 0 | 0 | 1 |

This is called less than *full rank* encoding where we retain all variables for each level of x. However, this creates perfect collinearity which causes problems with some machine learning algorithms (i.e., generalized regression models, neural networks). Alternatively, we can create full-rank one-hot encoding by dropping one of the levels (level a has been dropped):

|  |  |  |
| --- | --- | --- |
| id | x.b | x.c |
| 1 | 0 | 0 |
| 2 | 0 | 1 |
| 3 | 1 | 0 |
| 4 | 0 | 1 |
| 5 | 0 | 1 |
| 6 | 0 | 0 |
| 7 | 1 | 0 |
| 8 | 0 | 1 |

If you needed to manually implement one-hot encoding yourself you can with caret::dummyVars. Sometimes you may have a feature level with very few observations and all these observations show up in the test set but not the training set. The benefit of using dummyVars on the full data set and then applying the result to both the train and test data sets is that it will guarantee that the same features are represented in both the train and test data.

# full rank one-hot encode - recommended for generalized linear models and  
# neural networks  
full\_rank <- dummyVars( ~ ., data = ames, fullRank = TRUE)  
train\_oh <- predict(full\_rank, train\_1)  
test\_oh <- predict(full\_rank, test\_1)  
  
# less than full rank --> dummy encoding  
dummy <- dummyVars( ~ ., data = ames, fullRank = FALSE)  
train\_oh <- predict(dummy, train\_1)  
test\_oh <- predict(dummy, test\_1)

Two things to note: \* since one-hot encoding add new features it can significantly increase the dimensionality of our data. If you have a data set with many categorical variables and those categorical variables in turn have many unique levels, the number of features can explode. In these cases, you may want to explorer ordinal encoding of your data.

* if you use h2o you do not need to explicitly encode your categorical variables but you can overrride the default encoding. This can be considered a tuning parameter as some encoding approaches will improve modeling accuracy over other encodings. See the encoding options for h2o [here](http://docs.h2o.ai/h2o/latest-stable/h2o-docs/data-science/algo-params/categorical_encoding.html)

Reference <http://docs.h2o.ai/h2o/latest-stable/h2o-docs/welcome.html>

### Standardizing

Some models (K-NN, SVMs, PLS, Neural networks) require that the predictor variables have the same units. **Centering**, and **scaling** can be used for this purpose, and is often referred to as **standardizing** the features. Standardizing numeric variables results in zero mean and unit variance, which provides a common comparable unit of measure across all the variables.

Some packages have built-in arguments (i.e., h2o, caret) to standardize and some do not(i.e., glm, keras). If you need to manually standardize your variables you can use the preProcess function provided by the caret package. For example, here we center and scale our Ames predictor variables.

PK ] 8t}\_6-n&P9[!GΣ<U+0081>1s,U\*@z KKw=“XL<c5><e4>\*<U+00B1><U+0598><U+00BC><ec><e8>s<eb>K<e3>,yj<fb>j<U+00AB><U+009C> <U+FD52><U+00A9><U+031B>\*?:@<U+01DE><U+024D>-<ff><e0><U+00A5>?hM6=q<U+00A5>|<U+00BC>=X<U+00BC>SA<U+0095><U+0090><cc>D<c0>7aH<U+008F>o](<f9><U+0080>H<U+00B1><U+012C><U+009F><U+DDE2>\*%<U+00A1>trmh$<U+00AB>}pb$=<U+00A3><U+00B3>\_<e2><c3>$vm<U+00A8><f7><f2><U+009D><f4>0E<U+00BC>Z<f1><e4>$<U+0090><f6><da><e7><f8>l<U+008E>Ff<U+0633>"<U+0087><U+00B9>Xh<U+00B8><fc><c1>;<e9>^ <U+009A><U+009F><e7>#'<U+00B7><U+00B3><U+00A0><U+00BE>I%<U+0094>)<U+0081>x<U+00A8><U+00A0>?<f6><ce>\<U+00BA><f0><U+008F>w<U+00AF><U+009E>><U+0572><dd>6<dc>k<U+00B8><U+01AE>5<U+00BA><c3><fe><U+009D><c2><c5> <U+00BA><U+008A><U+029E><U+00B2>~<U+00AD><c3>kd<d1>Iw<U+00BC><f9><eb><df>PK <U+0083><U+00B7>|<U+0097><ec>s<e0><U+00A8>GF<fc>JT2<c9><c8><da><c1>k<U+0087><ee><U+00B3><dc>T,<<U+00AE>s<f5><U+009F>:<U+00BC>W<U+008E><U+008E><dd>,K<ed><f5>\<U+00BE><U+009D><U+00AA><ce>}-<U+00A4><U+009C>O\*]<U+009E><U+00AF><f4><f7><f6>1<U+00B0><U+0092>#%<U+00B4>I<f8><U+00B4><d0>G<e2><cb><U+00A1>PK !<ec> <U+008B>qn]<e2><U+008D>+<dd>S(W<d4><ce>;C<U+0091>c<U+0093>w&<fa>r<U+00A1><U+00AC>aS<ff>S<U+00BE><db><U+038D>w<U+0087>+<d8>S<d1>6\<e3><cc><U+008C><ff>(or<e6><U+0082><U+00A8>rO\_<f6><c6>t<e3><ef><U+00AB>Z6u-<U+008D><f4><ed>e<U+008A>iN4<U+0087><U+0097>|<e9>P<U+008D><e5> <U+00A8><eb>\_Y9VΧH<U+00A4>UsY]b<U+00AF>X<U+00A6><U+023A>?(<U+008B><f2> ge!<U+0083>zF<U+00AB>bX<U+0096>Ij'y\*<U+008F><U+0094>H<d4> <c2>T<U+00A2><U+00BD>zm<e5><U+008B>^T^<U+063C>0;<U+0084>8<e3>~<ca>:<U+03F9><d3><e2>JZW PK

# identify only the predictor variables  
features <- setdiff(names(train\_1), "Sale\_Price")  
features  
## [1] "MS\_SubClass" "MS\_Zoning" "Lot\_Frontage"   
## [4] "Lot\_Area" "Street" "Alley"   
## [7] "Lot\_Shape" "Land\_Contour" "Utilities"   
## [10] "Lot\_Config" "Land\_Slope" "Neighborhood"   
## [13] "Condition\_1" "Condition\_2" "Bldg\_Type"   
## [16] "House\_Style" "Overall\_Qual" "Overall\_Cond"   
## [19] "Year\_Built" "Year\_Remod\_Add" "Roof\_Style"   
## [22] "Roof\_Matl" "Exterior\_1st" "Exterior\_2nd"   
## [25] "Mas\_Vnr\_Type" "Mas\_Vnr\_Area" "Exter\_Qual"   
## [28] "Exter\_Cond" "Foundation" "Bsmt\_Qual"   
## [31] "Bsmt\_Cond" "Bsmt\_Exposure" "BsmtFin\_Type\_1"   
## [34] "BsmtFin\_SF\_1" "BsmtFin\_Type\_2" "BsmtFin\_SF\_2"   
## [37] "Bsmt\_Unf\_SF" "Total\_Bsmt\_SF" "Heating"   
## [40] "Heating\_QC" "Central\_Air" "Electrical"   
## [43] "First\_Flr\_SF" "Second\_Flr\_SF" "Low\_Qual\_Fin\_SF"   
## [46] "Gr\_Liv\_Area" "Bsmt\_Full\_Bath" "Bsmt\_Half\_Bath"   
## [49] "Full\_Bath" "Half\_Bath" "Bedroom\_AbvGr"   
## [52] "Kitchen\_AbvGr" "Kitchen\_Qual" "TotRms\_AbvGrd"   
## [55] "Functional" "Fireplaces" "Fireplace\_Qu"   
## [58] "Garage\_Type" "Garage\_Finish" "Garage\_Cars"   
## [61] "Garage\_Area" "Garage\_Qual" "Garage\_Cond"   
## [64] "Paved\_Drive" "Wood\_Deck\_SF" "Open\_Porch\_SF"   
## [67] "Enclosed\_Porch" "Three\_season\_porch" "Screen\_Porch"   
## [70] "Pool\_Area" "Pool\_QC" "Fence"   
## [73] "Misc\_Feature" "Misc\_Val" "Mo\_Sold"   
## [76] "Year\_Sold" "Sale\_Type" "Sale\_Condition"   
## [79] "Longitude" "Latitude"  
  
# pre-process estimation based on training features  
pre\_process <- preProcess(  
 x=train\_1[,features],  
 method=c("center","scale")  
)  
  
# apply to both training & test  
train\_x <- predict(pre\_process, train\_1[,features])  
test\_x <- predict(pre\_process, test\_1[,features])

### Alternative Feature Transformation

There are some alternative transformation that you can perform:

* Normalizing the predictor variables with Box Cox transformation can improve parametric model performance.
* Collapsing highly correlated variables with PCA can reduce the number of features and increase the stability of generalized linear models. However, this reduces the amount of information at your disposal and we show you how to use regularization as a better alternative to PCA.
* Remove near-zero variance variables. Variables with vary little variance tend to not improve model performance and can be removed. Notes: preProcess provides many other transformation options which you can read more about [here](https://topepo.github.io/caret/pre-processing.html).

For example, the following normalizes predictors with Box Cox transformation, center and scales continuous variables, peforms principal component analysis to reduce the predictor dimensions, and removes predictors with near zero variance.

# identify only the predictor variables  
features <- setdiff(names(train\_1), "Sale\_Price")  
  
# pre-process estimation based on training features  
pre\_process <- preProcess(  
 x = train\_1[, features],  
 method = c("BoxCox", "center", "scale", "pca", "nzv")   
 )  
  
# apply to both training & test  
train\_x <- predict(pre\_process, train\_1[, features])  
test\_x <- predict(pre\_process, test\_1[, features])

## Basic model formulation

There are **many** packages to perform machine learning and there are almost always more than one to perform each algorithm (i.e., there are over 20 packages to perform random forecasts). There are pros and cons to each package: some may be more computationally efficient while others may have more hyperparameter tuning options. Future chapters will expose you to many of the packages and algorithms that perform and scale best to most organization’s problems and data sets. Just realize there are *more ways than one to skin a* scream cat.

For example, these three functions will all produce the same linear regression model output.

lm.lm <- lm(Sale\_Price ~., data=train\_1)  
lm.glm <- glm(Sale\_Price ~ ., data = train\_1, family = gaussian)  
lm.caret <- train(Sale\_Price ~ ., data = train\_1, method = "lm")

One thing you will notice throughout this guide is that we cano specify our model formulation in different ways. In the above examples, we use the *model formulation* (Sale\_Price ~., which says explain Sale\_Price based on all features) approach. Alternative approaches, which you will see more often throughout this guide, are the matrix formulation and variable name specification approaches.

*Matrix formulation* requires that we separate our response variable from our features. For example, in the regularization session, we’ll use glmnet which requires our features (x) and response (y) variable to be specified separately.

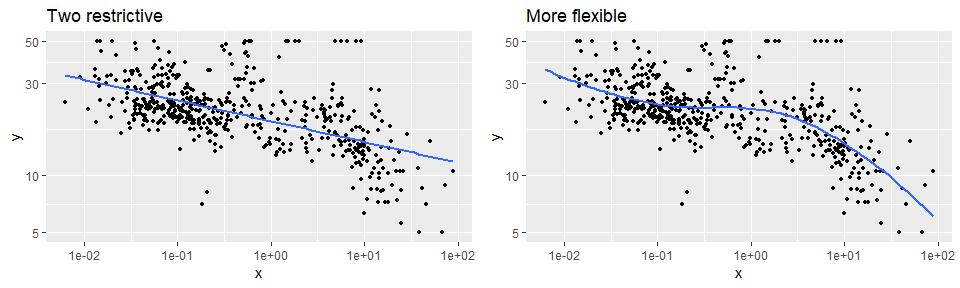
library(glmnet)  
  
# get feature names  
features <- setdiff(names(train\_1), "Sale\_Price")  
  
# create feature and response set  
train\_x <- train\_1[, features]  
train\_y <- train\_1$Sale\_Price  
  
# example of matrix formulation  
glmnet.m1 <- glmnet(x = train\_x, y = train\_y)

Alternatively, h2o uses *variable name specification* where we provide all the data combined in one training\_frame but we specify the features and response with character strings:

# create variable names and h2o training frame  
y <- "Sale\_Price"  
x <- setdiff(names(train\_1),y)  
train.h2o <- as.h2o(train\_1)  
  
# example of variable name specification  
h2o.m1 <- h2o.glm(x = x, y = y, training\_frame = train.h2o)

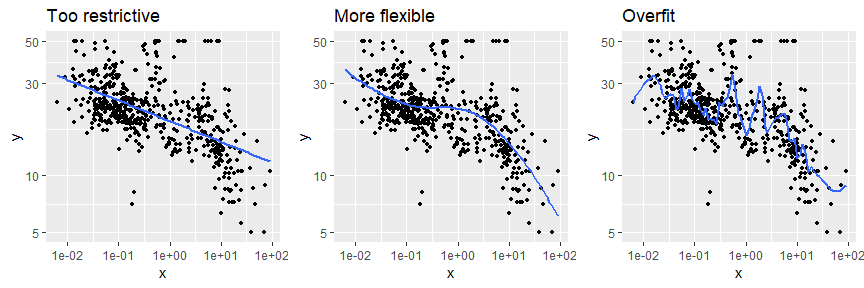
## Model tuning

Hyperparameters control the level of model complexity. Some algorithms have many tuning parameters while others have only one or two. Tuning can be a good thing as it allows us to transform our model to better align with patterns within our data. For example, the simple illustration below shows how the more flexible model aligns more closely to the data than the fixed linear model.



Tuning allows for more flexible patterns to be fit.

However, highly tunable models can also be dangerous because they allow us to overfit our model to the training data, which will not generalize well to future unseen data.



Highly tunable models can overfit if we are not careful.

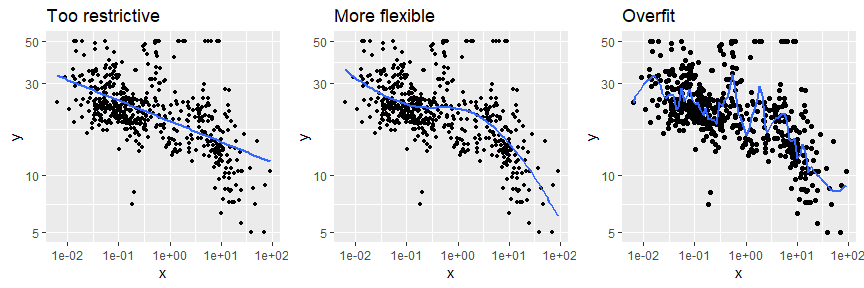
Throughout this guide, we will demonstrate how to tune the different parameters for each model. One way to performing hyperparameter tuning is to fiddle with hyperparameters manually until you find a great combination of hyperparameter values that result in high predictive accuracy. However, this would be very tedious work.

An alternative approach is to peform **grid search**. A **grid search** is anautomated approach to searching across many combinations of hyperparameter values. Throughout this guide, you will be exposed to different approaches to performing grid searches.

## Cross validation for generalization

Our goal is not only find a model that performs well on training data but to find one that performs well on *future unseen data*. So although we can tune our model to reduce some error metric to near zero on our training data.

Let’s go back to this emage

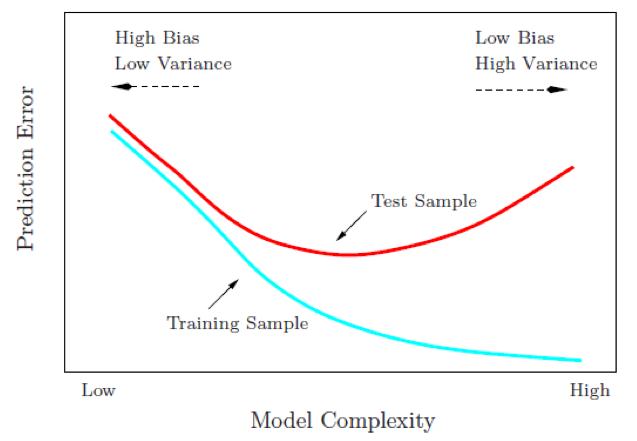


Bias versus variance.

The model on the left is considered rigid and consistent. If we provided it to a new training sample, with slightly different values the model would not change that much, if at all. Although it is consistent, the model does not accurately capture the underlying relationship. This is considered a model with high **biase** .

The model on the right is far more inconsistent. Even with small changes to our training sample, this model would likely change significantly. This is considered a model with high ***variance***.

The model in the middle balances the two and, likely, will minimize the error on future unseen data compared to the high bias and high variance models. This is our goal.



Bias-variance tradeoff.

To find the model that balances the **biase-variance tradeoff**, we search for a model that minimizes a *k*-fold cross-validation error metric. *k*-fold cross validation is a resampling method that randomly divides the training data into *k* groups (aka folds) of approximately equal size.The model is fit on folds and then the held-out validation fold is used to compute the error. This process is repeated *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 estimate is computed by averaging these values, which provides us with an approximation of the error to expect on unseen data.



Illustration of the k-fold cross validation process.

The algorithms we cover in this guide all have built-in cross validation capabilities. One typically use a 5 or 10 fold CV ( or ). For example, h2o implements CV with the nfolds argument:

# example of 10 fold CV in h2o  
h2o.cv <- h2o.glm(  
 x = x,   
 y = y,   
 training\_frame = train.h2o,  
 nfolds = 10  
 )

## Model evaluation

This leads us to our final topic, error metrics to evaluate performance. There are several metrics we can choose from to assess the error of a supervised machine learning model. The most common include:

### Regression models

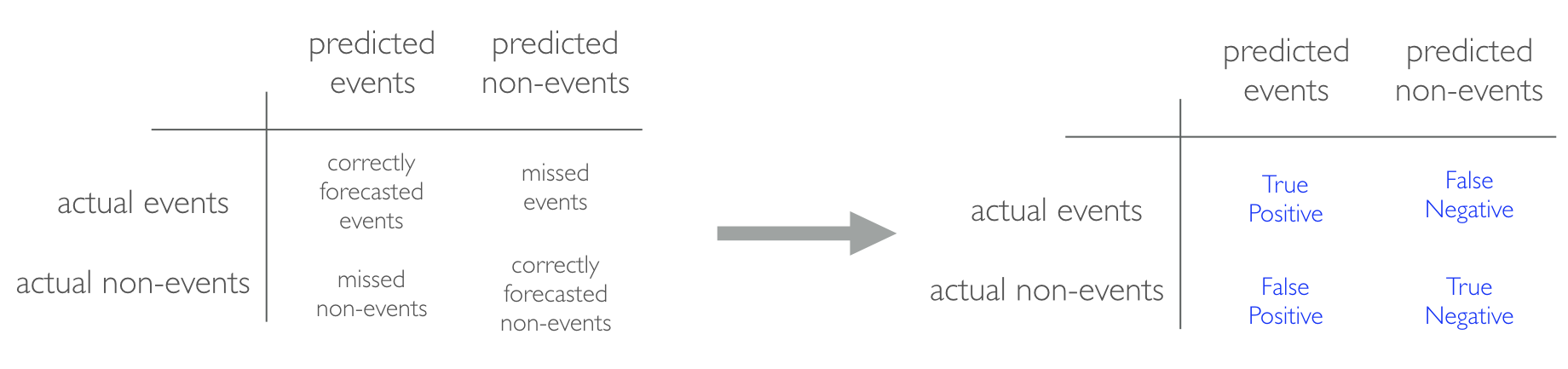
* **MSE**: Mean squared error is the average of the squared error (). The squared component results in larger errors having larger penalties. This (along with RMSE) is the most common error metric to use. **Objective: minimize**
* **RMSE**: Root mean squared error. This simply takes the square root of the MSE metric () so that your error is in the same units as your response variable. If your response variable units are dollars, the units of MSE are dollars-squared, but the RMSE will be in dollars. **Objective: minimize**
* **Deviance**: Short for mean residual deviance. In essence, it provides a measure of *goodness-of-fit* of the model being evaluated when compared to the null model (intercept only). If the response variable distribution is gaussian, then it is equal to MSE. When not, it usually gives a more useful estimate of error. **Objective: minimize**
* **MAE**: Mean absolute error. Similar to MSE but rather than squaring, it just takes the mean absolute difference between the actual and predicted values (). **Objective: minimize**
* **RMSLE**: Root mean squared logarithmic error. Similiar to RMSE but it performs a log() on the actual and predicted values prior to computing the difference (). When your response variable has a wide range of values, large repsonse values with large errors can dominate the MSE/RMSE metric. RMSLE minimizes this impact so that small response values with large errors can have just as meaningful of an impact as large response values with large errors. **Objective: minimize**
* : This is a popular metric that represents the proportion of the variance in the dependent variable that is predictable from the independent variable. Unfortunately, it has several limitations. For example, two models built from two different data sets could have the exact same RMSE but if one has less variability in the response variable then it would have a lower than the other. You should not place too much emphasis on this metric. **Objective: maximize**

Most models we assess in this guide will report most, if not all, of these metrics. We will emphasize MSE and RMSE but its good to realize that certain situations warrant emphasis on some more than others.

### Classification models

* **Misclassification**: This is the overall error. For example, say you are predicting 3 classes ( *high*, *medium*, *low* ) and each class has 25, 30, 35 observations respectively (90 observations total). If you misclassify 3 observations of class *high*, 6 of class *medium*, and 4 of class *low*, then you misclassified 13 out of 90 observations resulting in a 14% misclassification rate. **Objective: minimize**
* **Mean per class error**: This is the average error rate for each class. For the above example, this would be the mean of , which is 12%. If your classes are balanced this will be identical to misclassification. **Objective: minimize**
* **MSE**: Mean squared error. Computes the distance from 1.0 to the probability suggested. So, say we have three classes, A, B, and C, and your model predicts a probabilty of 0.91 for A, 0.07 for B, and 0.02 for C. If the correct answer was A the , if it is B , if it is C . The squared component results in large differences in probabilities for the true class having larger penalties. **Objective: minimize**
* **Cross-entropy (aka Log Loss or Deviance)**: Similar to MSE but it incorporates a log of the predicted probability multiplied by the true class. Consequently, this metric disproportionately punishes predictions where we predict a small probability for the true class, which is another way of saying having high confidence in the wrong answer is really bad. **Objective: minimize**
* **Gini index**: Mainly used with tree-based methods and commonly referred to as a measure of *purity* where a small value indicates that a node contains predominantly observations from a single class. **Objective: minimize**

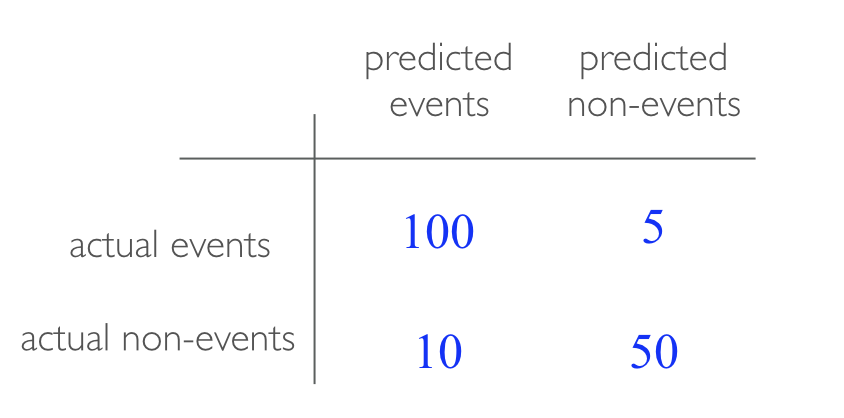
When applying classification models, we often use a *confusion matrix* to evaluate certain performance measures. A confusion matrix is simply a matrix that compares actual categorical levels (or events) to the predicted categorical levels. When we predict the right level, we refer to this as a *true positive*. However, if we predict a level or event that did not happen this is called a *false positive* (i.e. we predicted a customer would redeem a coupon and they did not). Alternatively, when we do not predict a level or event and it does happen that this is called a *false negative* (i.e. a customer that we did not predict to redeem a coupon does).



Confusion matrix.

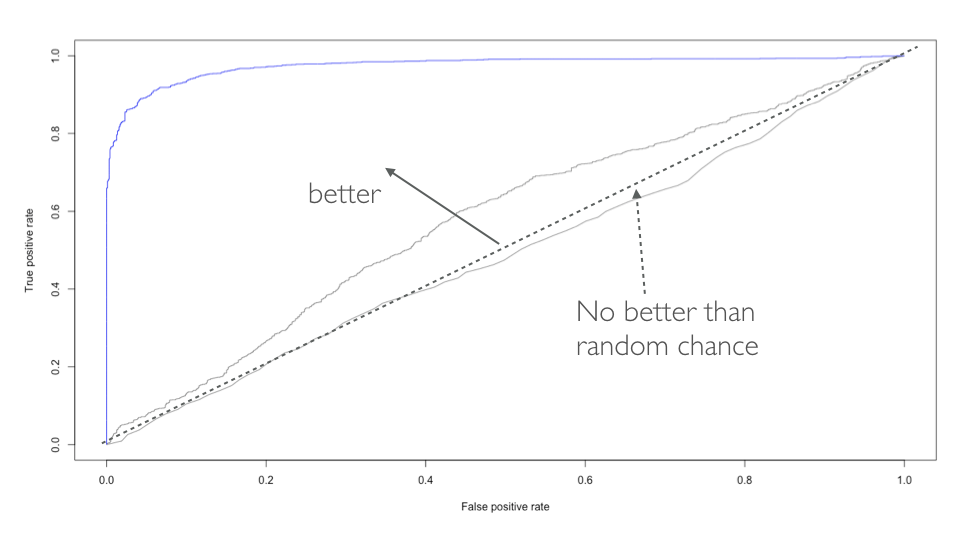
We can extract different levels of performance from these measures. For example, given the classification matrix below we can assess the following:

* **Accuracy**: Overall, how often is the classifier correct? Opposite of misclassification above. Example: . **Objective: maximize**
* **Precision**: How accurately does the classifier predict events? This metric is concerned with maximizing the true positives to false positive ratio. In other words, for the number of predictions that we made, how many were correct? Example: . **Objective: maximize**
* **Sensitivity (aka recall)**: How accurately does the classifier classify actual events? This metric is concerned with maximizing the true positives to false negatives ratio. In other words, for the events that occurred, how many did we predict? Example: . **Objective: maximize**
* **Specificity**: How accurately does the classifier classify actual non-events? Example: . **Objective: maximize**



Example confusion matrix.

* **AUC**: Area under the curve. A good classifier will have high precision and sensitivity. This means the classifier does well when it predicts an event will and will not occur, which minimizes false positives and false negatives. To capture this balance, we often use a ROC curve that plots the false positive rate along the x-axis and the true positive rate along the y-axis. A line that is diagonal from the lower left corner to the upper right corner represents a random guess. The higher the line is in the upper left-hand corner, the better. AUC computes the area under this curve. **Objective: maximize**



ROC curve.

# Regulaization regression

Generalized linear models (GLMs) such as ordinary least squares regression and logistic regression are simple and fundamentally approaches for supervised learning. Moreover, when the assumptions required by GMLs are met, the coefficients produced are unbiased and of all unbiased linear techniques, have the lowest variance.

However, in today’s world, data sets beign analyzed typically have a large amount of features. As the number of features grow, our GLM assumptions typically breakdown and our models often overfit (aka have high variance) to the training sample, causing our out of sample error to increase.

**Regularization** methods provide a means to control our coefficients, which can reduce the variance and decrease out of sample error.

## Prerequisites

PK ] 8t}\_6-n&P9[!GΣ<U+0081>1s,U\*@z KKw=“XL`*sK,yjj* ?:@-?hM6=q|=XSAD7aHo](H\*%trmh$<U+00AB>}pb$=\_$vm<U+00A8><f7><f2><U+009D><f4>0E<U+00BC>Z<f1><e4>$lFf“Xh;^ #’I%)x?<U+00BA>w>

6k5 ~kdIwPK |sGFJT2kT,<s:W

,K<U+00BE>}-O\*]1#%IGPK

This chapter leverages the following packages. Some of these packages are playing a supporting role while the main emphasis will be on the [**glmnet**](https://web.stanford.edu/~hastie/glmnet/glmnet_alpha.html) [@R-glmnet] and [**h2o**](http://docs.h2o.ai/?_ga=2.52641192.1255510165.1524074778-1013458140.1519825157) [@R-h2o] packages.

library(glmnet) # implementing regularized regression approaches  
library(h2o) # implementing regularized regression approaches  
library(rsample) # training vs testing data split  
library(dplyr) # basic data manipulation procedure  
library(ggplot2) #plotting

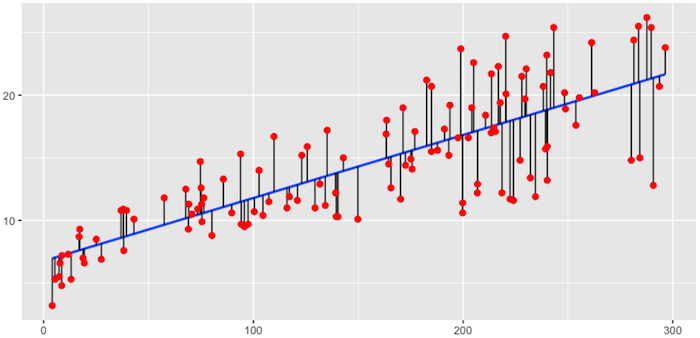
## Advantages & disadvantages

**Advantages:** \* Normal GLM models require that you have more observations than variables (); regularized regression allows you to model wide data where . \* Minimizes the impact of multicollinearity. \* Provides automatic feature selection (at least when you apply a **Lasso** or **elastic net** penalty). \* Minimal hyperparameters making it easy to tune. \* Computationally efficient - relatively fast compared to other algorithms in this guide and does not require large memory.

**Disadvantages:** \* Requires data pre-processing - requires **all variables to be numeric** (i.e. one-hot encode). However, **h2o** helps to automate this process. \* Does not handle **missing data** - must impute or remove observations with missing values. \* Not robust to outliers as they can still bias the coefficients. \* Assumes relationships between predictors and response variable to be monotonic linear (always increasing or decreasing in a linear fashion). \* Typically does not perform as well as more advanced methods that allow non-monotonic and non-linear relationships (i.e. random forests, gradient boosting machines, neural networks).

## The idea

The easiest way to understand regularized regression is to explain how it is applied to ordinary least squares regression (OLS). The objective of OLS regression is to find the plane that minimizes the sum of squared errors (SSE) between the observed and predicted response. Illustrated below, this means identifying the plane that minimizes the grey lines, which measure the distance between the observed (red dots) and predicted response (blue plane).



Fitted regression line using Ordinary Least Squares.

More formally, this objective function is written as:

$$\text{minimize} \bigg \{ SSE = \sum^n\_{i=1} (y\_i - \hat{y}\_i)^2 \bigg \} \tag{1}$$

The OLS objective function performs quite well when our data align to the key assumptions of OLS regression:

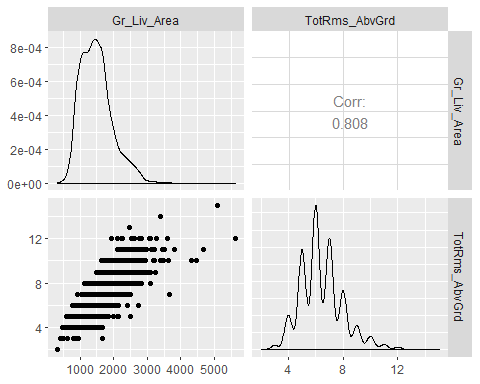
* Linear relationship
* Multivariate normality
* No autocorrelation
* Homoscedastic (constant variance in residuals)
* There are more observations (*n*) than features (*p*) ()
* No or little multicollinearity

However, for many real-life data sets, we have very *wide* data, meaning we have a large number of features (*p*) that we believe are informative in predicting some outcomes. As *p* increases, we can quickly violate some of the OLS assumptions and we require alternative approaches to provide predictive analytic solutions. Specifically, as *p* increases, there are three main issues we most commonly run into:

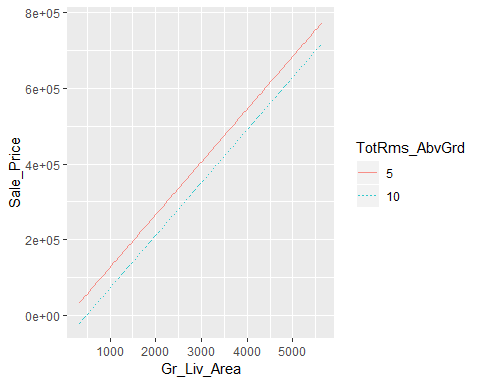
### 1. Multicollenearity

As *p* increases, we are more likely to capture multiple features that have some multicollenearity. When multicollinearity exists, we often see high variability in our coefficients terms. For example, in our Ames data, Gr\_Liv\_Area and TotRms\_AbvGrd are two variables that have a correlation of 0.808 and both variables are strongly correlated to our response variable (Sale\_Price). When we fit a model with both these variables, we get a positive coefficient for Gr\_Liv\_Area but a negative coefficient for TotRms\_AbvGrd, suggesting one has a positive impact to Sale\_Price and the other a negative impact.

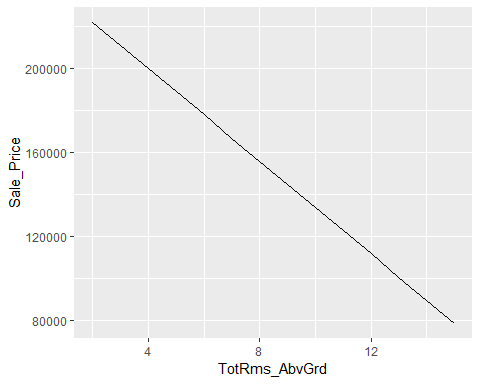
## GGally correlation check  
GGally:: ggpairs(ames[,c("Gr\_Liv\_Area","TotRms\_AbvGrd")])



# fit with two strongly correlated variables  
lm1 <- lm(Sale\_Price~Gr\_Liv\_Area+TotRms\_AbvGrd, data=ames)  
lm1  
##   
## Call:  
## lm(formula = Sale\_Price ~ Gr\_Liv\_Area + TotRms\_AbvGrd, data = ames)  
##   
## Coefficients:  
## (Intercept) Gr\_Liv\_Area TotRms\_AbvGrd   
## 42767.6 139.4 -11025.9  
  
library(statisticalModeling)  
fmodel(lm1)

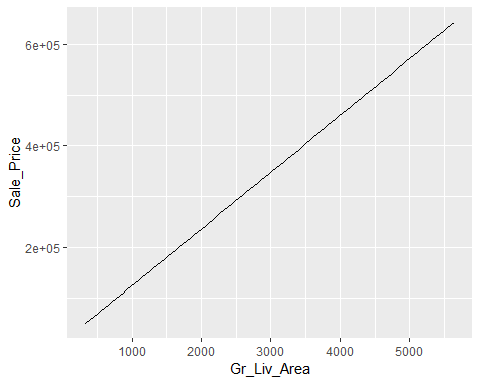


fmodel(lm1,~TotRms\_AbvGrd)

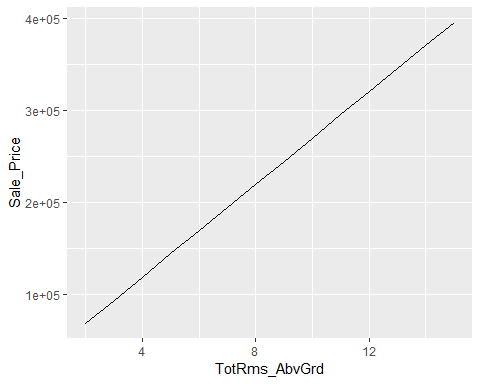


However, if we refit the model with each variable independently, they both show a positive impact. However, the Gr\_Liv\_Area effect is now smaller and the TotRms\_AbvGrd is positive with a much larger magnitude.

# fit with just Gr\_Liv\_Area  
lm2 <- lm(Sale\_Price~Gr\_Liv\_Area, data=ames)  
lm2  
##   
## Call:  
## lm(formula = Sale\_Price ~ Gr\_Liv\_Area, data = ames)  
##   
## Coefficients:  
## (Intercept) Gr\_Liv\_Area   
## 13289.6 111.7  
fmodel(lm2)



# fit with just TotRms\_Area  
lm3 <- lm(Sale\_Price~TotRms\_AbvGrd, data=ames)  
lm3   
##   
## Call:  
## lm(formula = Sale\_Price ~ TotRms\_AbvGrd, data = ames)  
##   
## Coefficients:  
## (Intercept) TotRms\_AbvGrd   
## 18665 25164  
lm3 %>% fmodel()



This is a common result when collenearity exists. Coefficients for correlated features become over-inflated and can flucturate signinificantly. One consequence of these large fluctuations in the coefficient terms is overfitting, which means we have high variance in the bias-variance tradeoff space. Although an analyst can use tools such as **variance inflaction factors** [@myers1990classical] to identify and remove those strongly correlated variables, it is not always clear which variable(s) to remove. Nor do we always wish to remove variables as this may be removing signal in our data.

### 2. Insufficient solution

When the number of features exceed the number of observations (), the OLS solution matrix is *not* invertible. This causes significant issues because it means: (1) The least-squares estimates are not unique. In fact, there are an infinite set of solutions available and most of these solutions overfit the data. (2) In many instances the result will be computationally infeasible.

Consequently, to resolve this issue an analyst can remove variables until and then fit an OLS regression model. Although an analyst can use pre-processing tools to guide this manual approach ([Kuhn & Johnson, 2013, pp. 43-47](http://appliedpredictivemodeling.com/)), it can be cumbersome and prone to errors.

### 3. Interpretability

With a large number of features, we often would like to identify a smaller subset of these features that exhibit the strongest effects. In essence, we sometimes prefere techniques that provide feature selection.

One approach to this is called *hard threshholding* feature selection, which can be performed with [linear model selection](http://uc-r.github.io/model_selection) approaches. However, model selection approaches can be computationally inefficient, do not scale well, and they simply assume a feature as in or out. We may wish to use a *soft threshholding* approach that slowly pushes a feature’s effect towards zero. As will be demonstrated, this can provide additional understanding regarding predictive signals.

### Regularized models

When we experience these concerns, one alternative to OLS regression is to use regularized regression (also commonly referred to as *penalized* models or *shrinkage* methods) to control the parameter estimates. Regularized regression puts contraints on the magnitude of the coefficients and will progressively shrink them towards zero. This constraint helps to reduce the magnitude and fluctuations of the coefficients and will reduce the variance of our model.

The objective function of regularized regression methods is very similar to OLS regression; however, we add a penalty parameter (*P*).

$$\text{minimize} \big \{ SSE + P \big \} \tag{2}$$

This penalty parameter constrains the size of the coefficients such that the only way the coefficients can increase is if we experience a comparable decrease in the sum of squared errors (SSE).

PK ] 8t}\_6-n&P9[!GΣ<U+0081>1s,U\*@z KKw=“XL`*sK,yjj* ?:@-?hM6=q|=XSAD7aHo](H\*%trmh$<U+00AB>}pb$=\_$vm<U+00A8><f7><f2><U+009D><f4>0E<U+00BC>Z<f1><e4>$lFf“Xh;^ #’I%)x?<U+00BA>w>

6k5 ~kdIwPK |sGFJT2kT,<s:W

,K<U+00BE>}-O*]1#%IGPK ;$VaL Hj<XhA*GFJsT6$2<CfcJI< A ZpqU3+l>A-U~!Gi;vOu3hsxJ

# 6J

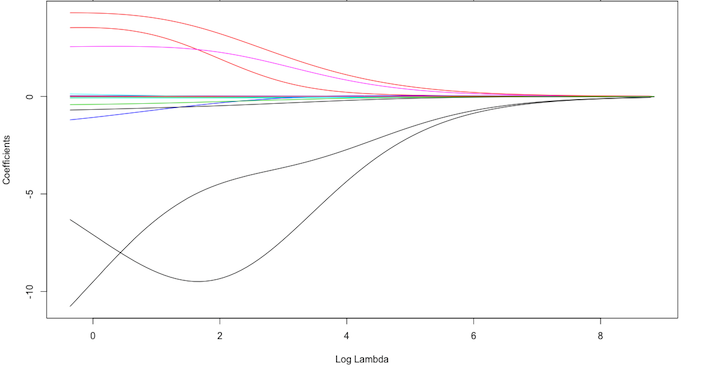
There are three types of penalty parametersx we can implement: 1. Ridge 2. Lasso 3. Elastic net, which is a combination of Ridge and Lasso

#### Ridge penalty

Ridge regression [@hoerl1970ridge] controls the coefficients by adding to the objective function. This penalty parameter is also referred to as “” as it signifies a second-order penalty being used on the coefficients.[[1]](#footnote-88)

$$\text{minimize } \bigg \{ SSE + \lambda \sum^p\_{j=1} \beta\_j^2 \bigg \} \tag{3}$$

This penalty parameter can take on a wide range of values, which is controlled by the *tuning parameter* . When there is no effect and our objective function equals the normal OLS regression objective function of simply minimizing SSE. However, as , the penalty becomes large and forces our coefficients to *near zero*. This is illustrated in Figure @ref(fig:ridge-coef-example) where exemplar coefficients have been regularized with ranging from 0 to over 8,000 ().



Ridge regression coefficients as grows from .

Although these coefficients were scaled and centered prior to the analysis, you will notice that some are extremely large when . Furthermore, you’ll notice the large negative parameter that fluctuates until where it then continuously skrinks to zero. This is indicitive of multicollinearity and likely illustrates that constraining our coefficients with may reduce the variance, and therefore the error, in our model.

In essence, the ridge regression model pushes many of the correlated features towards each other rather than allowing for one to be wildly positive and the other wildly negative. Furthermore, many of the non-important features get pushed to near zero. This allows us to reduce the noise in our data, which provides us more clarity in identifying the true signals in our model.

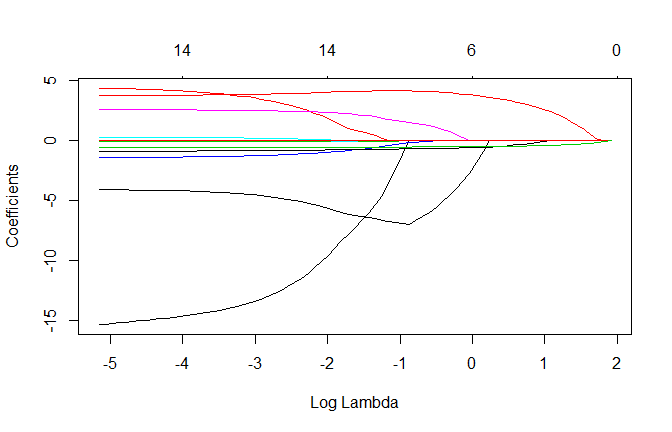
However, a ridge model will retain all variables. Therefore, a ridge model is good if you believe there is a need to retain all features in your model yet reduce the noise that less influential variables may create and minimize multicollinearity. However, a ridge model does not perform feature selection. If greater interpretation is necessary where you need to reduce the signal in your data to a smaller subset then a lasso or elastic net penalty may be preferable.

#### Lasso penalty

The *least absolute shrinkage and selection operator* (lasso) model [@tibshirani1996regression] is an alternative to the ridge penalty that has a small modification to the penalty in the objective function. Rather than the penalty we use the following penalty in the objective function.

$$\text{minimize } \bigg \{ SSE + \lambda \sum^p\_{j=1} | \beta\_j | \bigg \} \tag{4}$$

Whereas the ridge penalty approach pushes variables to *approximately but not equal to zero*, the lasso penalty will actually push coefficients to zero as illustrated in Figure @ref(fig:lasso-coef-example). Thus the lasso model not only improves the model with regularization but it also conducts automated feature selection.



Lasso regression coefficients as grows from . Numbers on top axis illustrate how many non-zero coefficients remain.

In the figure above we see that when all 15 variables are in the model, when 12 variables are retained, and when only 3 variables are retained. Consequently, when a data set has many features, lasso can be used to identify and extract those features with the largest (and most consistent) signal.

#### Elastic nets

A generalization of the ridge and lasso penalties is the *elastic net* penalty [@zou2005regularization], which combines the two penalties.

$$\text{minimize } \bigg \{ SSE + \lambda\_1 \sum^p\_{j=1} \beta\_j^2 + \lambda\_2 \sum^p\_{j=1} | \beta\_j | \bigg \} \tag{5}$$

Although lasso models perform feature selection, a result of their penalty parameter is that typically when two strongly correlated features are pushed towards zero, one may be pushed fully to zero while the other remains in the model. Furthermore, the process of one being in and one being out is not very systematic. In contrast, the ridge regression penalty is a little more effective in systematically reducing correlated features together. Consequently, the advantage of the elastic net penalty is that it enables effective regularization via the ridge penalty with the feature selection characteristics of the lasso penalty.

### Tuning

Regularized models are simple to tune as there are only two tuning parameters:

* **Size of penalty ()**: Controls how much we want to constrain our coefficients. Small penalties where is close to zero allow our coefficients to be larger; however, larger values of penalize our coefficients and forces them to take on smaller values. Hence, this parameter is often called the *shrinkage* parameter.
* **Alpha**: The alpha parameter tells our model to perform a ridge (alpha = 0), lasso (alpha = 1), or elastic net ().

### Package implementation

There are a few packages that impleent variants of regularized regression. You can find a comprehensive list on 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 chapter include:

* + [glmnet](https://cran.r-project.org/web/packages/glmnet/index.html): The original implementation of regularized regression in R. The **glmnet** R package provides an extremely efficient procedures for fitting the entire lasso or elastic-net regularization path for linear regression, logistic and multinomial regression models, Poisson regression and the Cox model. Two recent additions are the multiple-response Gaussian, and the grouped multinomial regression.

A nice vignette is available [here](https://cran.r-project.org/web/packages/glmnet/vignettes/glmnet_beta.pdf). Features include[[2]](#footnote-98): - The code can handle sparse input-matrix formats, as well as range constraints on coefficients. - Automatically standardizes your feature set. - Built-in cross validation. - The core of glmnet is a set of fortran subroutines, which make for very fast execution. - The algorithms use coordinate descent with warm starts and active set iterations. - Supports the following distributions: “gaussian”,“binomial”,“poisson”,“multinomial”,“cox”,“mgaussian”

* [h2o](https://cran.r-project.org/web/packages/gamboostLSS/index.html): The **h2o** R package is a powerful and efficient java-based interface that allows for local and cluster-based deployment. It comes with a fairly comprehensive [online resource](http://docs.h2o.ai/h2o/latest-stable/h2o-docs/index.html) that includes methodology and code documentation along with tutorials. Features include:
  + Fits both regularized and non-regularized GLMs.
  + Automated feature pre-processing (one-hot encode & standardization).
  + Built-in cross validation.
  + Built-in grid search capabilities.
  + Supports the following distributions: “guassian”, “binomial”, “multinomial”, “ordinal”, “poisson”, “gamma”, “tweedie”.
  + 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.

## Impelementation: regression

To illustrate various regularization concepts for a regression problem we will use the Ames. IA housing data, where our intent is to predict Sale\_Price.

# Create training (70%) and test (30%) sets for the AmesHousing::make\_ames() data.  
# Use set.seed for reproducibility  
  
set.seed(123)  
  
library(rsample)  
ames\_split <- initial\_split(AmesHousing::make\_ames(), prop=.7, strata="Sale\_Price")  
ames\_train <- training(ames\_split)  
ames\_test <- testing(ames\_split)

### glmnet

The glmnet package is a fast implementation, but it requires some extra processing up-front to your data if it’s not already represented as a numeric matrix. **glmnet** does not use the formula method (y~x) so prior to modeling, we need to create our feature and target set. Furthermore, we use the model.matrix function on our feature set (see Matrix::sparse.model.matrix) for increased efficiency on large dimension data). We also log transform our response variable due to its skewness.

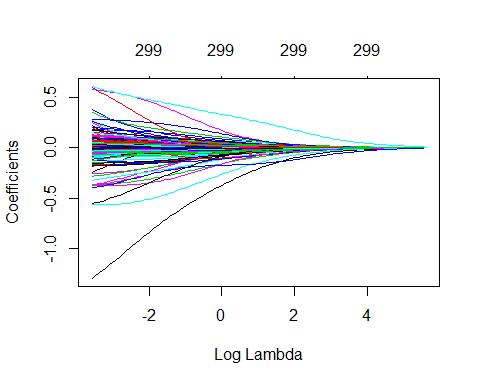
# create training and testing feature matrices  
# we use model.matrix(...)[,-1] to discard the intercept  
  
train\_x <- model.matrix(Sale\_Price~., ames\_train)[,-1]  
test\_x <- model.matrix(Sale\_Price~., ames\_test)[,-1]  
  
## create training and testing response vectors  
## transform y with log transformation  
train\_y <- log(ames\_train$Sale\_Price)  
test\_y <- log(ames\_test$Sale\_Price)

### Basic impelementation

To apply a regularized model we can use the glmnet::glmnet function. The alpha parameter tells **glmnet** to perform a ridge (alpha=0), lasso (alpha = 1), or elastic net () model. Behind the scenes, **glmnet** is doing two things that you should be aware of:

1. It is essential that predictor variables are standardized when performing regularized regression. **glmnet** performs this for you. If you standardize your predictors prior to **glmnet** you can turn this argument off with standardize = FALSE.
2. **glmnet** will perform ridge models across a wide range of parameters, which are illustrated in the figure below.

# apply ridge regression to attrition data  
ridge <- glmnet(  
 x=train\_x,  
 y=train\_y,  
 alpha=0  
)  
  
plot(ridge, xvar="lambda")



Coefficients for our ridge regression model as grows from .

In fact, we can see the exact values applied with ridge$lambda. Although you can specify your own values, by default **glmnet** applies 100 values that are data derived.

Notes: glmnet has built-in functions to auto-generate the appropriate values based on the data so the vast majority of the time you will have little need to adjust the default values.

We can also directly access the coefficients for a model using coef. **glmnet** stores all the coefficients for each model in order of largest to smallest . Due to the number of features, here I just peak at the two largest coefficients (Latitude & Overall\_QualVery\_Excellent) features for the largest (279.1035) and smallest (0.02791035). You can see how the largest value has pushed these coefficients to nearly 0.

# lambdas applied to penalty parameter  
ridge$lambda %>% head()  
## [1] 279.1035 254.3087 231.7166 211.1316 192.3752 175.2851  
  
# small lambda results in large coefficients  
coef(ridge)[c("Latitude", "Overall\_QualVery\_Excellent"), 100]  
## Latitude Overall\_QualVery\_Excellent   
## 0.60585376 0.09800466  
  
# large lambda results in small coefficients  
coef(ridge)[c("Latitude", "Overall\_QualVery\_Excellent"), 1]   
## Latitude Overall\_QualVery\_Excellent   
## 6.228028e-36 9.372514e-37

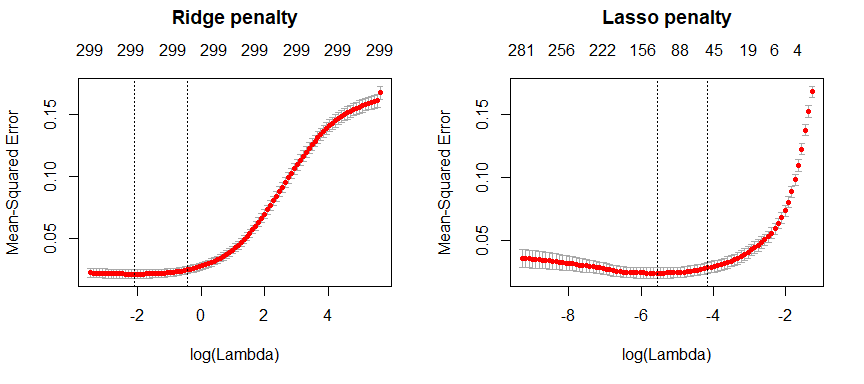
However, at this point, we do not understand how much improvement we are experiencing in our loss function across various values.

#### Tuning

Recall that is a tuning parameter that helps to control our model from over-fitting to the training data. However, to identify the optimal value we need we need to perform [cross-validation](#cv) (CV). cv.glmnet provides a built-in option to perform k-fold CV, and by default, performs 10-fold CV. Here we perform a CV glmnet model for both a ridge and lasso penalty.

Notes: By default, cv.glmnet uses MSE as the loss function but you can also use mean absolute error by changing the type.measure argument.

# apply CV ridge regression to Ames data  
ridge <- cv.glmnet(  
 x = train\_x,  
 y = train\_y,  
 alpha = 0  
)  
  
# apply cv lasso regression to Ames data  
lasso <- cv.glmnet(  
 x = train\_x,  
 y = train\_y,  
 alpha = 1  
)  
  
# plot results  
par(mfrow=c(1,2))  
plot(ridge, main="Ridge penalty\n\n")  
plot(lasso, main="Lasso penalty\n\n")



10-fold cross validation MSE for a ridge and lasso model. First dotted vertical line in each plot represents the with the smallest MSE and the second represents the with an MSE within one standard error of the minimum MSE.

Our plots above illustrate the 10-fold CV mean squared error (MSE) across the values. In both models we see a slight improvement in the MSE as our penalty gets larger , suggesting that a regular OLS model likely overfits our data. But as we constrain it further (continue to increase the penalty), our MSE starts to increase. The numbers at the top of the plot refer to the number of variables in the model. Ridge regression does not force any variables to exactly zero so all features will remain in the model but we see the number of variables retained in the lasso model go down as our penalty increases.

The first and second vertical dashed lines represent the value with the minimum MSE and the largest value within one standard error of the minimum MSE.

# Ridge model  
min(ridge$cvm) # minimum MSE  
## [1] 0.02147691  
ridge$lambda.min # lambda for this min MSE  
## [1] 0.1236602  
  
ridge$cvm[ridge$lambda == ridge$lambda.1se] # 1 st.error of min MSE  
## [1] 0.02488411  
ridge$lambda.1se # lambda for this MSE  
## [1] 0.6599372  
  
# Lasso model  
min(lasso$cvm) # minimum MSE  
## [1] 0.02411134  
lasso$lambda.min # lambda for this min MSE  
## [1] 0.003865266  
  
lasso$cvm[lasso$lambda == lasso$lambda.1se] # 1 st.error of min MSE  
## [1] 0.02819356  
lasso$lambda.1se # lambda for this MSE  
## [1] 0.01560415

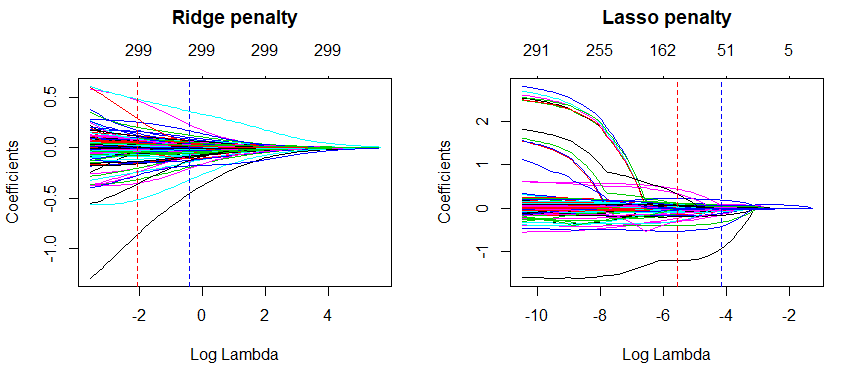
We can assess this visually. Here we plot the coefficients across the values and the dashed red line represents the with the smallest MSE and the dashed blue line represents largest that falls within one standard error of the minimum MSE. This shows you how much we can constrain the coefficients while still maximizing predictive accuracy.

PK ] 8t}\_6-n&P9[!GΣ<U+0081>1s,U\*@z KKw=“XL`*sK,yjj* ?:@-?hM6=q|=XSAD7aHo](H\*%trmh$<U+00AB>}pb$=\_$vm<U+00A8><f7><f2><U+009D><f4>0E<U+00BC>Z<f1><e4>$lFf“Xh;^ #’I%)x?<U+00BA>w>

6k5 ~kdIwPK |sGFJT2kT,<s:W

,K<U+00BE>}-O\*]1#%IGPK 4@gd4c;jloZB^<%Is}I/M[}zY6@B N1cOm xY\_l6Wk.3t8“j0ivm<$<ec><U+00A8>s<ef><U+008C>p<e4>2<U+009D><f6><U+00BB>e9N<fd><ff>"z<U+07DC><fc><U+0086><ed><e6> <d8>Y<U+00B4>S<U+00BE><U+00A5>2+8<fc>C<de><U+0599>w hT<ea><U+4E7C><U+00A1>}<c6><U+07ED><U+00BF><U+009B><U+008D><c5>A<e9><c8>v<cc>g;=d:<d3><d8><U+00BB><U+009B>}q<ef>$[p~vbE+Iz\*G

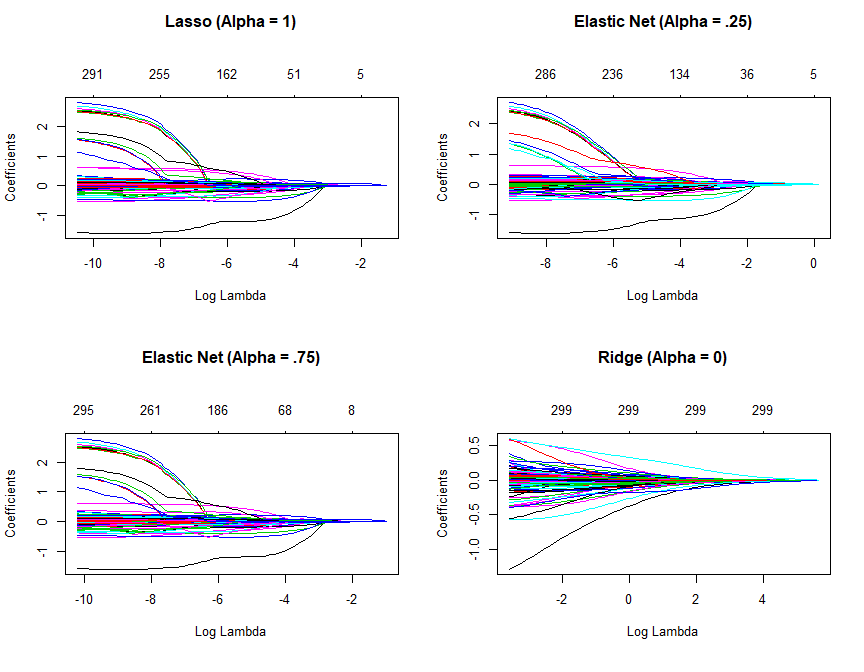
# Ridge model  
ridge\_min <- glmnet(  
 x = train\_x,  
 y = train\_y,  
 alpha = 0  
)  
  
# Lasso model  
lasso\_min <- glmnet(  
 x = train\_x,  
 y = train\_y,  
 alpha = 1  
)  
  
par(mfrow = c(1, 2))  
# plot ridge model  
plot(ridge\_min, xvar = "lambda", main = "Ridge penalty\n\n")  
abline(v = log(ridge$lambda.min), col = "red", lty = "dashed")  
abline(v = log(ridge$lambda.1se), col = "blue", lty = "dashed")  
  
# plot lasso model  
plot(lasso\_min, xvar = "lambda", main = "Lasso penalty\n\n")  
abline(v = log(lasso$lambda.min), col = "red", lty = "dashed")  
abline(v = log(lasso$lambda.1se), col = "blue", lty = "dashed")



Coefficients for our ridge and lasso models. First dotted vertical line in each plot represents the with the smallest MSE and the second represents the with an MSE within one standard error of the minimum MSE.

So far we’ve implemented a pure ridge and pure lasso model. However, we can implement an elastic net the same way as the ridge and lasso models, by adjusting the alpha parameter. Any alpha value between 0-1 will perform an elastic net. When alpha = 0.5 we perform an equal combination of penalties whereas alpha will have a heavier ridge penalty applied and alpha will have a heavier lasso penalty.

lasso <- glmnet(train\_x, train\_y, alpha = 1.0)   
elastic1 <- glmnet(train\_x, train\_y, alpha = 0.25)   
elastic2 <- glmnet(train\_x, train\_y, alpha = 0.75)   
ridge <- glmnet(train\_x, train\_y, alpha = 0.0)  
  
par(mfrow = c(2, 2), mar = c(6, 4, 6, 2) + 0.1)  
plot(lasso, xvar = "lambda", main = "Lasso (Alpha = 1)\n\n\n")  
plot(elastic1, xvar = "lambda", main = "Elastic Net (Alpha = .25)\n\n\n")  
plot(elastic2, xvar = "lambda", main = "Elastic Net (Alpha = .75)\n\n\n")  
plot(ridge, xvar = "lambda", main = "Ridge (Alpha = 0)\n\n\n")



Coefficients for various penalty parameters.

Often, the optimal model contains an alpha somewhere between 0-1, thus we want to tune both the and the alpha parameters. To set up our tuning, we create a common fold\_id, which just allows us to apply the same CV folds to each model. We then create a tuning grid that searches across a range of alpha’s from 0-1, and empty columns where we’ll dump our model results into.

PK ] 8t}\_6-n&P9[!GΣ<U+0081>1s,U\*@z KKw=“XL`*sK,yjj* ?:@-?hM6=q|=XSAD7aHo](H\*%trmh$<U+00AB>}pb$=\_$vm<U+00A8><f7><f2><U+009D><f4>0E<U+00BC>Z<f1><e4>$lFf“Xh;^ #’I%)x?<U+00BA>w>

6k5 ~kdIwPK |sGFJT2kT,<s:W

,K<U+00BE>}-O\*]1#%IGPK

1${)yG5a&ie]],N#4V<TRp|b]ea材’#m1Yx)xДh@Z/;ZhGa<Z\_‘t’xe7=yv@(6 (H05pZ“(0)@k{^[FWβV

# maintain the same folds across all models  
fold\_id <- sample(1:10, size = length(train\_y), replace = TRUE)  
  
# search across a range of alphas  
tuning\_grid <- tibble::tibble(  
 alpha = seq(0, 1, by = .1),  
 mse\_min = NA,  
 mse\_1se = NA,  
 lambda\_min = NA,  
 lambda\_1se = NA  
)

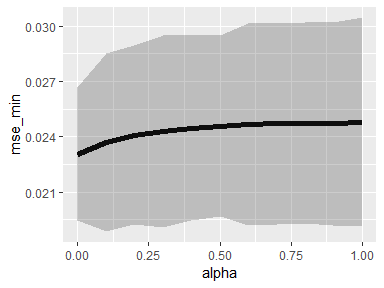
Now we can iterate over each alpha value, apply a CV elastic net, and extract the minimum and one standard error MSE values and respective values.

Notes: This grid search took 41 seconds to compute.

# perform grid search  
for (i in seq\_along(tuning\_grid$alpha)){  
 # fit CV model for each alpha value  
 fit <- cv.glmnet(train\_x, train\_y, alpha=tuning\_grid$alpha[i],foldid=fold\_id)  
   
 # extract MSE and lambda values  
 tuning\_grid$mse\_min[i] <- fit$cvm[fit$lambda == fit$lambda.min]  
 tuning\_grid$mse\_1se[i] <- fit$cvm[fit$lambda == fit$lambda.1se]  
 tuning\_grid$lambda\_min[i] <- fit$lambda.min  
 tuning\_grid$lambda\_1se[i] <- fit$lambda.1se  
}  
  
tuning\_grid  
## # A tibble: 11 x 5  
## alpha mse\_min mse\_1se lambda\_min lambda\_1se  
## <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 0 0.0230 0.0267 0.179 0.795   
## 2 0.1 0.0237 0.0285 0.0387 0.156   
## 3 0.2 0.0241 0.0289 0.0193 0.0856  
## 4 0.3 0.0243 0.0295 0.0129 0.0627  
## 5 0.4 0.0245 0.0295 0.00966 0.0470  
## 6 0.5 0.0246 0.0295 0.00773 0.0376  
## 7 0.6 0.0247 0.0301 0.00644 0.0344  
## 8 0.7 0.0247 0.0302 0.00552 0.0295  
## 9 0.8 0.0247 0.0302 0.00483 0.0258  
## 10 0.9 0.0247 0.0302 0.00429 0.0229  
## 11 1 0.0248 0.0304 0.00387 0.0206

Iwe we plot the MSE ﾂｱ one standard error for the optimal value for each alpha setting, we see that they all fall within the same level of accuracy. Consequently, we could select a full lasso model (alpha=1) with , gain the benefits of its feature selection capability and reasonably assume no loss in accuracy.

tuning\_grid %>%   
 mutate(se=mse\_1se - mse\_min) %>%   
 ggplot(aes(alpha, mse\_min)) +  
 geom\_line(size=2)+  
 geom\_ribbon(aes(ymax=mse\_min+se,ymin=mse\_min-se),alpha=.25)



MSE ﾂｱ one standard error for different alpha penalty parameters.

#### Feature interpretation

Regularized regression **assumes a monotonic linear releationship** between the predictor variables, and the response. The *linear* relationship part of that statement just means, for a given predictor variable, it assumes for every one unit change in a give predictor variable there is a constant change in the response. This constant change is given by the given coefficient for a predictor. The *monotonic* relationship means that a given predictor variable will always have a positive or negative relationship.

Consequently, this makes understanding variable relationships simple with regularized models. And since the predictors have all been standardized, comparing the coefficients against one another allows us to identify the most influential predictors. Those predictors with the largest positive coefficients have the largest positive impact on our response variable and those variables with really small negative coefficients have a very small negative impact on our response. Furthermore, there is no difference between the global and local model interpretation (see model interpretation chapter for details).

We’ll illustrate with the following models.

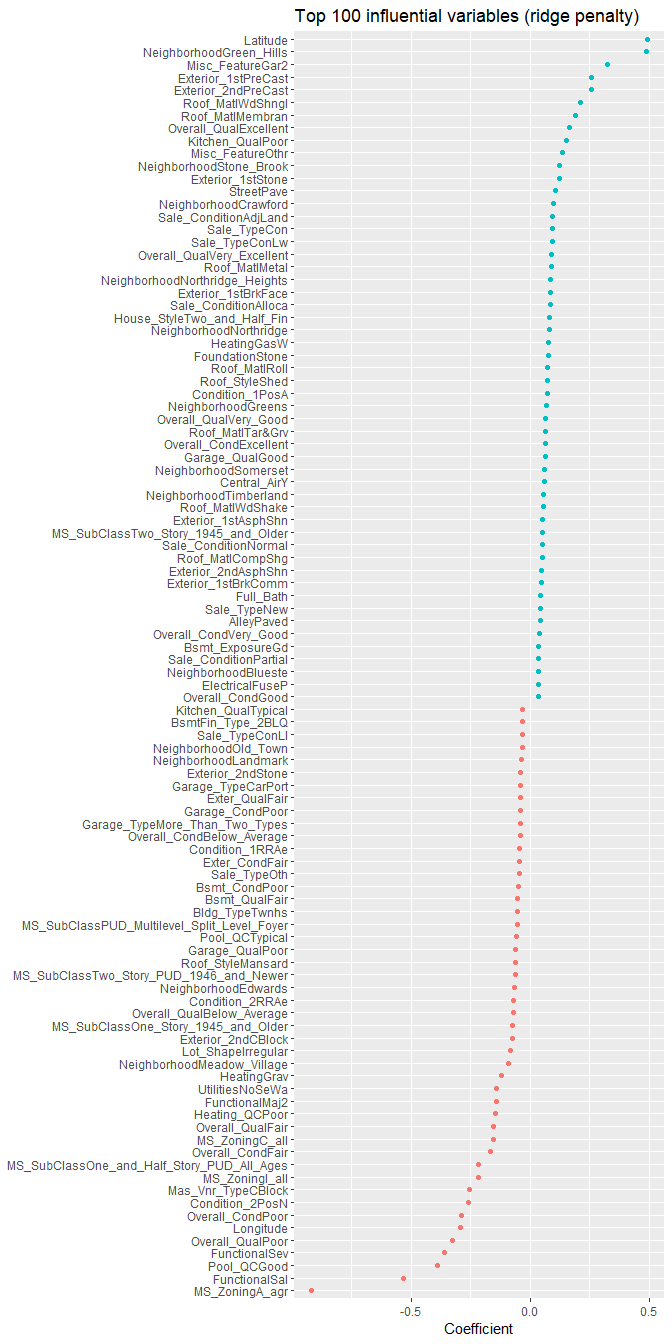
lasso <- cv.glmnet(train\_x, train\_y,alpha=1)  
ridge <- cv.glmnet(train\_x, train\_y,alpha=0)

Our ridge model will retain all variables. Therefore, a ridge model is good if you believe there is a need to retain all features in your model yet reduce the noise that less influential variables may create and minimize multicollinearity. However, a ridge model does not perform feature selection but it will typically push most variables to near zero and allow the most influential variables to be more prominent. The following extracts the coefficients of our ridge model and plots the top 100 most influential variables. You can see that many of the variables have coefficients closer to zero but there a handful of predictor variables that have noticable influence.

For example, properties with above average latitude (northern end of Ames), are in the Green Hills neighborhood (which is actually in the south part of Ames), or have 2 extra miscellaneous garage features will have a positive influence on the sales price. Alternatively, properties that are zoned agricultural, have a home functional code of “Sal” (salvage only) or “Sev” (severely damaged), or have a pool quality assessment of “Good” tend have a negative influence on the sales price.

PK ] 8t}\_6-n&P9[!GΣ<U+0081>1s,U\*@z KKw=“XL<c5><e4>\*<U+00B1><U+0598><U+00BC><ec><e8>s<eb>K<e3>,yj<fb>j<U+00AB><U+009C> <U+FD52><U+00A9><U+031B>\*?:@<U+01DE><U+024D>-<ff><e0><U+00A5>?hM6=q<U+00A5>|<U+00BC>=X<U+00BC>SA<U+0095><U+0090><cc>D<c0>7aH<U+008F>o](<f9><U+0080>H<U+00B1><U+012C><U+009F><U+DDE2>\*%<U+00A1>trmh$<U+00AB>}pb$=<U+00A3><U+00B3>\_<e2><c3>$vm<U+00A8><f7><f2><U+009D><f4>0E<U+00BC>Z<f1><e4>$<U+0090><f6><da><e7><f8>l<U+008E>Ff<U+0633>"<U+0087><U+00B9>Xh<U+00B8><fc><c1>;<e9>^ <U+009A><U+009F><e7>#'<U+00B7><U+00B3><U+00A0><U+00BE>I%<U+0094>)<U+0081>x<U+00A8><U+00A0>?<f6><ce>\<U+00BA><f0><U+008F>w<U+00AF><U+009E>><U+0572><dd>6<dc>k<U+00B8><U+01AE>5<U+00BA><c3><fe><U+009D><c2><c5> <U+00BA><U+008A><U+029E><U+00B2>~<U+00AD><c3>kd<d1>Iw<U+00BC><f9><eb><df>PK <U+0083><U+00B7>|<U+0097><ec>s<e0><U+00A8>GF<fc>JT2<c9><c8><da><c1>k<U+0087><ee><U+00B3><dc>T,<<U+00AE>s<f5><U+009F>:<U+00BC>W<U+008E><U+008E><dd>,K<ed><f5>\<U+00BE><U+009D><U+00AA><ce>}-<U+00A4><U+009C>O\*]<U+009E><U+00AF><f4><f7><f6>1<U+00B0><U+0092>#%<U+00B4>I<f8><U+00B4><d0>G<e2><cb><U+00A1>PK 6<fd>:<U+00AB>c<f4>E<U+009E>U<U+0093><U+0070>p<U+009E><U+00AC><ec><ed><U+008C><f2><ca>U<de>;.=;E!<U+0088><U+009E>i<f2>7<cb><e5><db><U+0720><U+00B6><d9>$cΑq<U+00BB><U+009D>Vt; 0<U+008B><c4>z<e1><e7><U+008A>05<c5>x<U+00A8><U+00B5><U+00B3><U+009A>[g-<db>b<U+0092><U+00BA>0Z<U+00B1>n/<U+0094>3<U+0168>2-sG<f7><U+00A7><U+008E><ce>4s]<U+00B9><C;A<U+009B>;<f0>g%c<U+00AF><d7><ea> <d2>[><d8><eb><fd>34<U+008E><U+00AF><fe>v<dc><cc>6<U+00A4><U+00AD>+<f7>i<f5><c3><U+3787><e5>s<dc>7}<d1>a<U+00B3><ce><ee>4<U+0087>x<U+008F><U+008C><U+00A3><U+0097>[<ce>7<U+00AB><fc>P:<<U+0086><U+0063><U+00A2>u<e6><U+0099>qG<d9><e6>#<U+0091>m<c1>h[B<U+00AC><e5>gP<U+008E>V-w <U+00B8><U+0091>P<d0><d7>V<U+00BA><U+009A><fd><eb><U+00A1>^<db>H,G<c5>!1<e0>v<U+00BF>0u<c7>X<U+00A8><U+0086><U+0090><f0><U+0083>V<d8>$<U+00AD>t><c9><U+00B7><U+0087><d6><ea><U+0098>D<U+0092><U+00B9>8X<fc><U+00AB>A<f8>]-O>oeRsS#<?sXhjmyZP-UmH\

coef(ridge, s = "lambda.min") %>%  
 broom::tidy() %>%  
 filter(row != "(Intercept)") %>%  
 top\_n(100, wt = abs(value)) %>%  
 ggplot(aes(value, reorder(row, value), color = value > 0)) +  
 geom\_point(show.legend = FALSE) +  
 ggtitle("Top 100 influential variables (ridge penalty)") +  
 xlab("Coefficient") +  
 ylab(NULL)

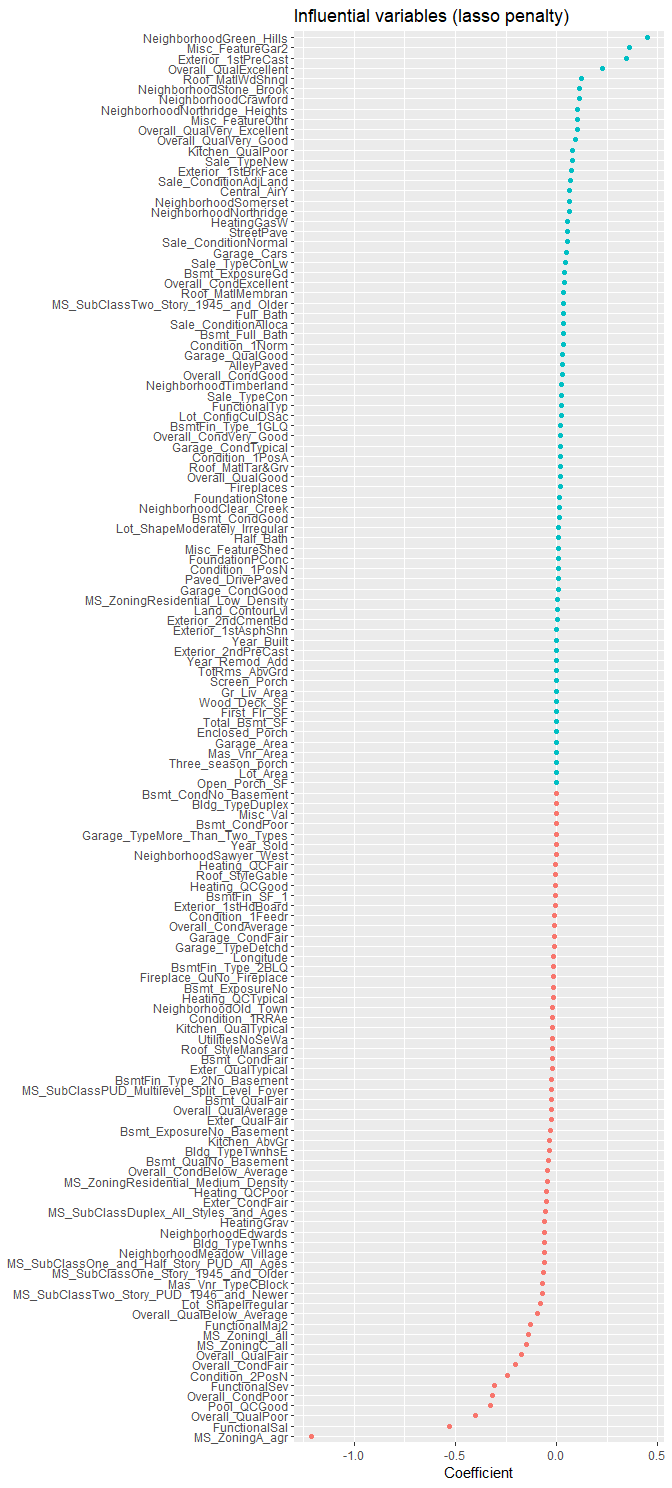


A ridge penalty will retain all variables but push most coefficients to near zero. Consequently, we retain any minor signals that all features provide but we can visualize those coefficients with the largest absolute values to identify the most influential predictors.

Similar to ridge, the lasso pushes many of the collinear features towards each other rather than allowing for one to be wildly positive and the other wildly negative. However, unlike ridge, the lasso will actually push coefficients to zero and perform feature selection. This simplifies and automates the process of identifying those features most influential to predictive accuracy. If we select the model with the minimum MSE and plot all variables in that model we see similar results to the ridge regarding the most influential variables.

Notes: Although the ordering typically differs, we often see some commonality between lasso and ridge models regarding the most influential variables.

coef(lasso, s="lambda.min") %>%   
 broom::tidy() %>%   
 filter(row!="(Intercept)") %>%   
 ggplot(aes(value, reorder(row,value),color=value>0))+  
 geom\_point(show.legend = FALSE)+  
 ggtitle("Influential variables (lasso penalty)")+  
 xlab("Coefficient")+  
 ylab(NULL)



A lasso penalty will perform feature selection by pushing most coefficients to zero. Consequently, we can view all coefficients to see which features were selected; however, our objective usually is still to identify those features with the strongest signal (largest absolute coefficient values).

#### Predicting {# regression-glmnet-predict}

Once you have identified your preferred model, you can simply use predict to predict the same model on a new data set. The only caveat is you need to supply predict an s parameter with the preferred models value. For example, here we create a lasso model, which provides me a minimum MSE of 0.02398 (RMSE = 0.123). However, our response variable is log transformed so we must re-transform it to get an interpretable RMSE (our average generalized error is $25,156.77).

PK M ] 8t}\_6-n&P9[!GΣ<U+0081>1s,U\*@z KKw=“XL<c5><e4>\*<U+00B1><U+0598><U+00BC><ec><e8>s<eb>K<e3>,yj<fb>j<U+00AB><U+009C> <U+FD52><U+00A9><U+031B>\*?:@<U+01DE><U+024D>-<ff><e0><U+00A5>?hM6=q<U+00A5>|<U+00BC>=X<U+00BC>SA<U+0095><U+0090><cc>D<c0>7aH<U+008F>o](<f9><U+0080>H<U+00B1><U+012C><U+009F><U+DDE2>\*%<U+00A1>trmh$<U+00AB>}pb$=<U+00A3><U+00B3>\_<e2><c3>$vm<U+00A8><f7><f2><U+009D><f4>0E<U+00BC>Z<f1><e4>$<U+0090><f6><da><e7><f8>l<U+008E>Ff<U+0633>"<U+0087><U+00B9>Xh<U+00B8><fc><c1>;<e9>^ <U+009A><U+009F><e7>#'<U+00B7><U+00B3><U+00A0><U+00BE>I%<U+0094>)<U+0081>x<U+00A8><U+00A0>?<f6><ce>\<U+00BA><f0><U+008F>w<U+00AF><U+009E>><U+0572><dd>6<dc>k<U+00B8><U+01AE>5<U+00BA><c3><fe><U+009D><c2><c5> <U+00BA><U+008A><U+029E><U+00B2>~<U+00AD><c3>kd<d1>Iw<U+00BC><f9><eb><df>PK <U+0083>M<U+008E><ec><dc> <U+0083><U+00B7>|<U+0097><ec>s<e0><U+00A8>GF<fc>JT2<c9><c8><da><c1>k<U+0087><ee><U+00B3><dc>T,<<U+00AE>s<f5><U+009F>:<U+00BC>W<U+008E><U+008E><dd>,K<ed><f5>\<U+00BE><U+009D><U+00AA><ce>}-<U+00A4><U+009C>O\*]<U+009E><U+00AF><f4><f7><f6>1<U+00B0><U+0092>#%<U+00B4>I<f8><U+00B4><d0>G<e2><cb><U+00A1>PK <U+0083>M<ea><f2><U+0093><ca># <c2>vj<e2>t<U+0086><cd>K<d2>KQl<U+00B7> <U+0094>/{<fa>Q<fe><d3>n<U+0087> A<U+00B1>9 9Y v BAWvf%7p\*%!w9D1eH*N8’D&J02R@5G)aI[cf ^=tFFS[tU%v&rzEf5| Ht9t!Ia,^LH<d0>6MyJw<U+0093>v<d0>$4i<U+02C6><e9><f8><U+0089><e8>Y<U+00A4>,<U+00B7>%<U+00BF>D<e7><e0>"S<U+00A6><U+DE70><e5>B<c4><f2>lJlMV\_g.xC1BB<;drV* PK Mz

# optimal model  
cv\_lasso <- cv.glmnet(train\_x, train\_y, alpha=1.0)  
min(cv\_lasso$cvm)  
## [1] 0.02529035  
  
# predict and get RMSE  
pred <- predict(cv\_lasso, s = cv\_lasso$lambda.min, test\_x)  
caret::RMSE(pred, test\_y)  
## [1] 0.1220084  
  
# re-transform predicted values and get interpretable RMSE  
caret::RMSE(exp(pred), exp(test\_y))  
## [1] 24740.36

### h2o

To perform regularized regression with **h2o**, we first need to initiate our **h2o** session.

Next, we do not need to one-hot encode or standardize our variables as **h2o** will do this for us. However, we do want to normalize our response variable due to its skewness and then convert our training and test data to **h2o** objects.

# convert training data to h2o object  
train\_h2o <- ames\_train %>%  
 mutate(Sale\_Price=log(Sale\_Price)) %>%   
 as.h2o()  
  
# set the response column to Sale\_Price  
response <- "Sale\_Price"  
  
# set the predictor names  
predictors <- setdiff(colnames(ames\_train), response)

#### Basic implementation

h2o.glm allows us to perform a generalized linear model. If the response variable is continuous, h2o.glm will use a gaussian distribution (see family parameter ?h2o.glm). By default, h2o.glm performs an elastic net model with alpha = .5. Similar to **glmnet**, h2o.glm will perform an automated search across internally generated lambda values.

Notes: You can override the automated lambda search by supplying different values to the lambda parameters in h2o.glm but this is not recommended as the default parameters typically perform best.

The following performs as default h2o.glm model with alpha=.5 and it performs a 10 fold cross validation (nfolds=10).

# train your model, where you specify alpha(performs 10-fold cv)  
h2o\_fit1 <- h2o.glm(  
 x=predictors,  
 y=response,  
 training\_frame=train\_h2o,  
 nfolds=10,  
 keep\_cross\_validation\_predictions=T,  
 alpha=.5,  
 family="gaussian"  
)  
  
# print the MSe and RMSE for the validation data  
h2o.mse(h2o\_fit1, xval=T)  
## [1] 0.0405933  
  
h2o.rmse(h2o\_fit1, xval=TRUE)  
## [1] 0.2014778

If we check out the summary results of our model we get a bunch of information. Below is truncated printout which provides important model information such as the alpha applied and the optimal lambda value identified (), the number of predictors retained with these penalty parameters (11), and performance results for the training and validation sets.

summary(h2o\_fit1)  
## Model Details:  
## ==============  
##   
## H2ORegressionModel: glm  
## Model Key: GLM\_model\_R\_1531935157122\_1   
## GLM Model: summary  
## family link regularization number\_of\_predictors\_total number\_of\_active\_predictors number\_of\_iterations #training\_frame  
## 1 gaussian identity Elastic Net (alpha = 0.5, lambda = 0.05581 ) 345 11 2 file13d1f4ffcacf3\_sid\_be66\_16  
##   
## H2ORegressionMetrics: glm  
## \*\* Reported on training data. \*\*  
##   
## MSE: 0.03800294  
## RMSE: 0.1949434  
## MAE: 0.1277845  
## RMSLE: 0.01521899  
## Mean Residual Deviance : 0.03800294  
## R^2 : 0.7737851  
## Null Deviance :345.0615  
## Null D.o.F. :2053  
## Residual Deviance :78.05805  
## Residual D.o.F. :2042  
## AIC :-861.7687  
##   
##   
##   
## H2ORegressionMetrics: glm  
## \*\* Reported on cross-validation data. \*\*  
## \*\* 10-fold cross-validation on training data (Metrics computed for combined holdout predictions) \*\*  
##   
## MSE: 0.0405933  
## RMSE: 0.2014778  
## MAE: 0.1310502  
## RMSLE: 0.01567771  
## Mean Residual Deviance : 0.0405933  
## R^2 : 0.7583658  
## Null Deviance :345.4348  
## Null D.o.F. :2053  
## Residual Deviance :83.37864  
## Residual D.o.F. :2042  
## AIC :-726.3293  
##   
## truncated.....  
##

#### Tuning

As previously stated, a full grid search to identify the optimal alpha is not always necessary; changing its value to 0.5 (or 0 to 1 if we only want Ridge or Lasso, respectively) works in most cases. However, if a full grid search is desired then we need to supply our grid of alpha values in a list. We can then use h20.grid to perform our grid search. The results show that (a full ridge penalty) performed best.

This grid search took 5 seconds to compute!

# create hyperparameter grid  
hyper\_params <- list(alpha = seq(0, 1, by = .1))  
  
# perform grid search  
grid <- h2o.grid(  
 x = predictors,   
 y = response,   
 training\_frame = train\_h2o,   
 nfolds = 10,  
 keep\_cross\_validation\_predictions = TRUE,  
 algorithm = "glm",  
 grid\_id = "grid\_search",   
 hyper\_params = hyper\_params  
)  
  
# Sort the grid models by MSE  
sorted\_grid <- h2o.getGrid("grid\_search", sort\_by = "mse", decreasing = FALSE)  
sorted\_grid  
## H2O Grid Details  
## ================  
##   
## Grid ID: grid\_search   
## Used hyper parameters:   
## - alpha   
## Number of models: 11   
## Number of failed models: 0   
##   
## Hyper-Parameter Search Summary: ordered by increasing mse  
## alpha model\_ids mse  
## 1 [0.0] grid\_search\_model\_0 0.022711429277773212  
## 2 [0.6] grid\_search\_model\_6 0.0395992947950626  
## 3 [0.7] grid\_search\_model\_7 0.03968294882574923  
## 4 [1.0] grid\_search\_model\_10 0.03983076501706063  
## 5 [0.4] grid\_search\_model\_4 0.039850863900647  
## 6 [0.8] grid\_search\_model\_8 0.03990607242146858  
## 7 [0.3] grid\_search\_model\_3 0.04019054535587161  
## 8 [0.5] grid\_search\_model\_5 0.04042328742741519  
## 9 [0.9] grid\_search\_model\_9 0.04044324344598643  
## 10 [0.2] grid\_search\_model\_2 0.041434990130949347  
## 11 [0.1] grid\_search\_model\_1 0.04233746428242222

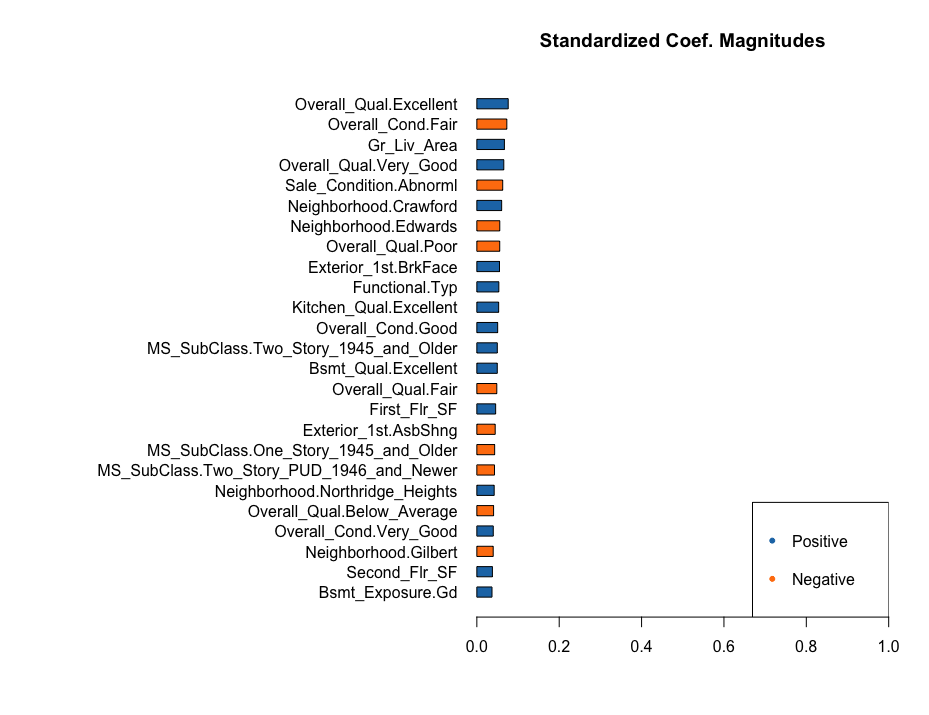
We can check out more details of the best performing model. Our RMSE (0.12279) is an improvement on our default model (RMSE=0.2015). Also, we can access the optimal model parameters, which show the optimal value for our model was 0.0279.

# grab top model id  
best\_h2o\_model <- sorted\_grid@model\_ids[[1]]  
best\_model <- h2o.getModel(best\_h2o\_model)  
  
# assess performance  
h2o.mse(best\_model)  
## [1] 0.0163625  
  
h2o.rmse(best\_model)  
## [1] 0.127916  
  
# get optimal parameters  
best\_model@parameters$lambda  
## [1] 0.02790355  
  
best\_model@parameters$alpha  
## [1] 0

### Feature interpretation

**h2o** provides a built-in function that plots variable importance. To compute variable importance for regularized models, **h2o** uses the standardized coefficient values (whichi s the same tjat we plotted in the **glmnet** example). You will notice that the the largest influential variables produced by this H2O model differ from the glmnet model. This is because we are assessing the ridge model here, where in the glmnet interpretation section we assessed the lasso and H2O and glmnet use differing approaches to generate the search path. However, you will notice that both Overall\_Qual.Excellent and Overall\_Cond.Fair were also top influencers in the glmnet model suggesting they may have a strong signal regardless of the regularization penalty we use.

# get top 25 influential variables  
h2o.varimp\_plot(best\_model, num\_of\_features = 25)



H2O’s variable importance plot. Provides the same output as plotting the standardized coefficients (h2o.std\_coef\_plot).

Although H2o’s variable importance uses standardized coefficients, a convenient function of H2o is that you can extract the “normal” coefficients which are obtained from the standardized coefficients by reversing the data standardization process (de-scaled, with the interecept adjusted by an added offset) so that they can be applied to data in its original form (i.e. no standardization prior to scoring). For example, every one unit increase in pool size has a $1 decrease in sales price (since our response is log transformed we need to take the exponent ).

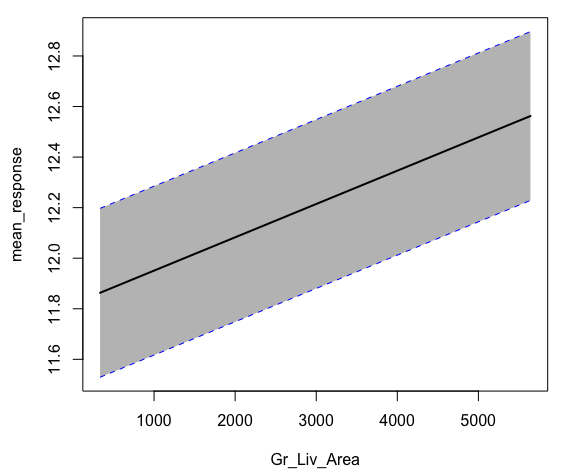
These are not the same as coefficients of a model built on non-standardized data.

best\_model@model$coefficients\_table  
## Coefficients: glm coefficients  
## names coefficients standardized\_coefficients  
## 1 Intercept -47.473537 11.713031  
## 2 Neighborhood.Bloomington\_Heights -0.019249 -0.019249  
## 3 Neighborhood.Blueste 0.004638 0.004638  
## 4 Neighborhood.Briardale -0.009169 -0.009169  
## 5 Neighborhood.Brookside 0.008538 0.008538  
##   
## ---  
## names coefficients standardized\_coefficients  
## 341 Pool\_Area -0.000183 -0.005963  
## 342 Misc\_Val -0.000038 -0.022608  
## 343 Mo\_Sold 0.000444 0.001224  
## 344 Year\_Sold -0.007414 -0.009728  
## 345 Longitude -0.303402 -0.007786  
## 346 Latitude 0.948544 0.017592

In the **glmnet** section we discussed how regularized regression assumes a monotonic linear relationship between the predictor variables, and the response. The *linear* relationship part of that statement just means, for a give predictor variable, it assumes for every one unit change in a given predictor variable there is constant change in the response. This constant change is given by the given coefficient for a predictor. The *monotonic* relationship means that a given predictor variable will always have a positive or negative relationship.

We can illustrate this with a partial dependence plot of the ground living area (square footage) variable. The PDP plot shows that the relationship is monotonic linear (assumes a constant increasing relationships). This PDP plot helps to show the typical values (and one standard error) of our response variable as the square footage of the ground floor living space increases.

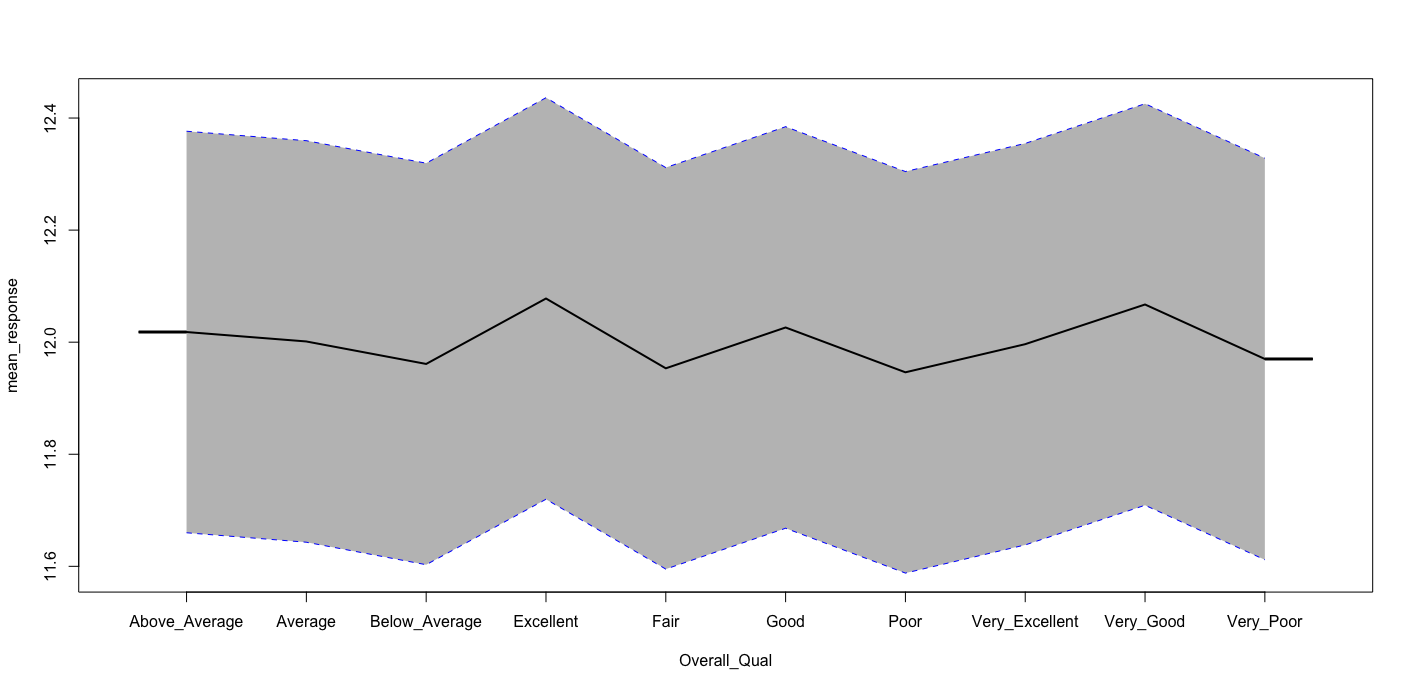
# partial dependence plots for top 2 influential variables  
h2o.partialPlot(best\_model, data = train\_h2o, cols = "Gr\_Liv\_Area")



As the ground living area (square footage) of a home increases, we experience a constant increase in the mean predicted sale price.

But what about the two most influential variables (Overall\_Qual.Excellent and Overall\_Cond.Fair)? These variables are a result of one-hot encoding the original overall quality (Overall\_Qual) variable. We can assess a similar plot but must supply the original non-one-hot encoded variable name.

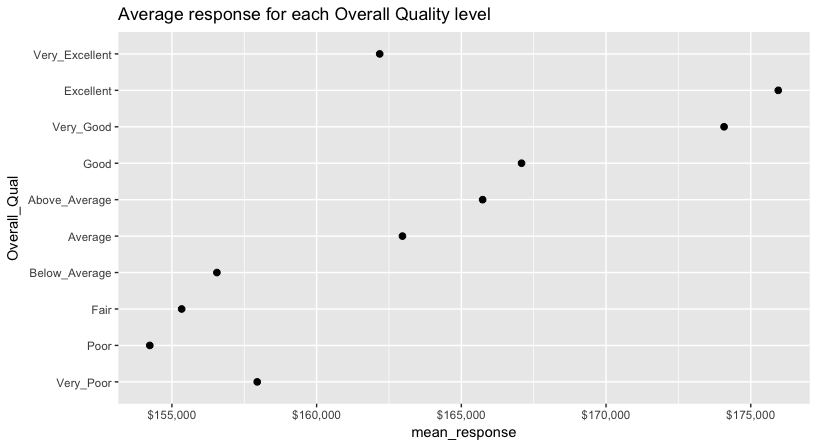
h2o.partialPlot(best\_model, data = train\_h2o, cols = "Overall\_Qual")



The mean predicted sale price for each level of the overall quality variable. Note how the plot does not align with the natural ordering of the predictor variable.

Unfortunately, H2O窶冱 function plots the categorical levels in alphabetical order. Alternatively, we can extract the results and plot them in their proper level order to make inference more intuitive. The following shows the marginal effect of the overall quality variable on sales price. It illustrates an interesting finding - the highest and lowest categories do not have the largest marginal sales price effects. It also shows that a house with a *very good* overall quality has, on average, a $6K higher sale price than a house with only a *good* overall quality. Alternatively, homes with *below average* quality receive a substantially lower sale price than homes with *average* quality.

h2o.partialPlot(best\_model, data=train\_h2o, cols="OVerall-Qual", plot=FALSE) %>%   
 as.data.frame() %>%   
 mutate(  
 Overall\_Qual=factor(OVerall\_Qual, levels=levels(ames$Overall\_Qual)),  
 mean\_response=exp(mean\_response) %>%   
 ) %>%   
 ggplot(aes(mean\_response,Overall\_Qual))+  
 scale\_x\_continuous(labels=scales::dollar())+  
 ggtitle("Average response for each overall Quality level")



The mean predicted sale price for each level of the overall quality variable. This plot now helps to illustrate how the mean predicted sale price changes based on the natural ordering of the overall quality predictor variable.

See the model interpretation chapter for more details on variable importance and partial dependence plots.

#### Predicting

Lastly, we can use h2o.predict and h2o.performance to predict and evaluate our models performance on our hold out test data. Similar to **glmnet**, we need to re-transform our predicted values to get a interpretable generalizable error. Our generalizable error is $24,147.60 which is about $1,000 lower than the **glmnet** model produced.

# make predictions  
pred <- h2o.predict(object = best\_model, newdata = test\_h2o)  
head(pred)  
## predict  
## 1 11.65765  
## 2 11.48417  
## 3 12.50938  
## 4 12.32368  
## 5 13.20056  
## 6 12.69091  
  
# assess performance  
h2o.performance(best\_model, newdata = test\_h2o)  
## H2ORegressionMetrics: glm  
##   
## MSE: 0.01364287  
## RMSE: 0.1168027  
## MAE: 0.0826363  
## RMSLE: 0.009030871  
## Mean Residual Deviance : 0.01364287  
## R^2 : 0.915491  
## Null Deviance :141.57  
## Null D.o.F. :875  
## Residual Deviance :11.95116  
## Residual D.o.F. :530  
## AIC :-582.0349

# conver predicted values to non-transformed  
caret::RMSE(as.vector(exp(pred)), ames\_test$Sale\_Price)  
## [1] 24740.36  
## [1] 24147.6

# shutdown h2o  
h2o.removeAll()  
## [1] 0  
h2o.shutdown(prompt = FALSE)  
## [1] TRUE

## Implementation: Binary Classification

To illustrate various regularization concepts for a binary classification problem we will use the employee attrition data, where the goal is to predict whether or not an employee attrits (“Yes” vs. “No”).

The easiest way to have consistent interpretable results is to recode the response as 1 for the positive class (“Yes”) and 0 for the other class (“No”).

library(dplyr)  
attrition <- rsample::attrition %>%   
 mutate(Attrition = recode(Attrition, "Yes" = 1, "No" = 0)) %>%  
 mutate\_if(is.ordered, factor, ordered = FALSE)

### glmnet

Similar to the regression application, for the classification data set we need to perfom some extra processing up-front to prepare for modeling with **glmnet**. First, many of the features are categorical; consequently, we need to either ordinal encode or one-hot encode these variables so that all features are numeric. Also, **glmnet** does not use the formula method (y ~ x) so prior to modeling we need to create our feature and target set and convert our features to a matrix.

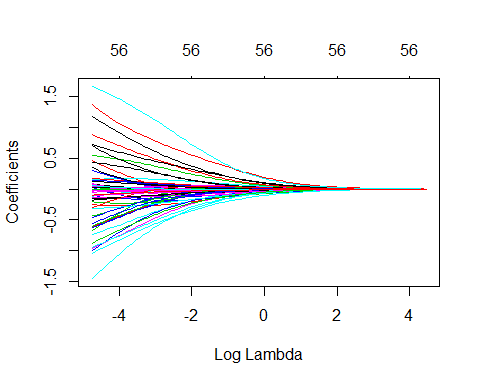
Since our response variable is imbalanced, we use rsample and strat to perform stratified sampling so that both our training and testing data sets have similar proportion of response levels.

# one-hot encode our data with model.matrix  
one\_hot <- model.matrix(~., attrition[,-1]) %>% as.data.frame()  
  
# create training and testing sets  
set.seed(123)  
split <- initial\_split(one\_hot, prop=.8, strata="Attrition")  
train <- training(split)  
test <- testing(split)  
  
# separate features from response variable for glmnet  
train\_x <- train %>% select(-Attrition) %>% as.matrix()  
train\_y <- train$Attrition  
test\_x <- test %>% select(-Attrition) %>% as.matrix()  
test\_y <- test$Attrition  
  
# check that train % test sets have common response ratio  
table(train\_y) %>% prop.table()  
## train\_y  
## 0 1   
## 0.8385726 0.1614274  
table(test\_y) %>% prop.table()  
## test\_y  
## 0 1   
## 0.8395904 0.1604096

#### Basic implementation

Similar to the regression problem, we apply a regularized classification model with glmnet::glmnet(). The primary difference is that for binary classification models we need to include family = binomial. Remember, the alpha parameter tells **glmnet** to perform a ridge (alpha = 0), lasso (alpha = 1), or elastic net () model.

# Apply Ridge regression to attrition data  
  
ridge <- glmnet(  
 x=train\_x,  
 y=train\_y,  
 family="binomial",  
 alpha=0  
)  
  
plot(ridge,xvar="lambda")



Coefficients for our ridge regression model as grows from .

We can also directly access the coefficients for a model using coef and tidy the output with tidy. Here, we check out the top 10 largest absolute coefficient terms when using the smallest and largest lambda values. We see that regardless of a large or small penalty parameter, working overtime and being a Sales Rep has the largest positive influence on the probability of attrition. Whereas being a Research Director and having high job involvement (among others) are the strongest influencers for reducing the probability of attrition.

# lambdas applied to penalty parameter  
ridge$lambda %>% head()  
  
# small lambda results in large coefficients  
coef(ridge)[,100] %>%   
 tidy() %>%   
 arrange(desc(abs(x)))  
# names  
# <chr>  
# x  
# <dbl>  
# OverTimeYes 1.670349e+00   
# JobInvolvementVery\_High -1.450717e+00   
# JobRoleSales\_Representative 1.377262e+00   
# BusinessTravelTravel\_Frequently 1.175904e+00   
# JobRoleResearch\_Director -1.038779e+00   
# JobInvolvementHigh -1.003056e+00   
# EnvironmentSatisfactionVery\_High -9.570745e-01   
# JobSatisfactionVery\_High -9.423902e-01   
# JobRoleLaboratory\_Technician 8.838645e-01   
# WorkLifeBalanceBetter -8.773846e-01  
  
# large lambda results in small coefficients  
coef(ridge)[, 1] %>%   
 tidy() %>%   
 arrange(desc(abs(x)))   
  
# names <chr> x<dbl>  
# (Intercept) -1.647646e+00   
# JobRoleSales\_Representative 2.622098e-37   
# OverTimeYes 1.945071e-37   
# JobRoleResearch\_Director -1.545065e-37   
# MaritalStatusSingle 1.391505e-37   
# JobRoleManager -1.237927e-37   
# BusinessTravelTravel\_Frequently 1.156814e-37   
# JobRoleLaboratory\_Technician 9.157843e-38   
# JobRoleManufacturing\_Director -8.954434e-38   
# JobInvolvementVery\_High -7.657277e-38

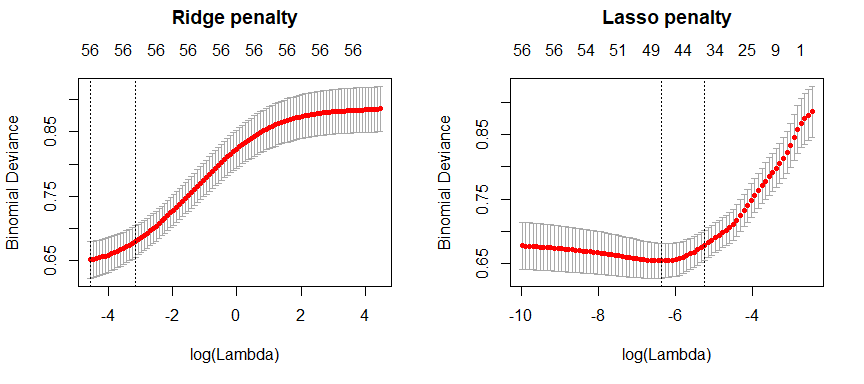
However, at this point, we do not understand how much improvement we are experiencing in our loss function across various values.

#### Tuning

Recall that is a tuning parameter that helps to control our model from over-fitting to the training data. However, to identify the optimal value we need to perform cross-validation with cv.glmnet. Here we perform a 10-fold CV glmnet model for both a ridge and lasso penalty.

By default, cv.glmnet uses deviance as the loss function for binomial classification but you could adjust type.measure to “auc”, “mse”, “mae”, or “class” (missclassification).

# Apply CV Ridge regression to attrition data  
ridge <- cv.glmnet(  
 x = train\_x,  
 y = train\_y,  
 family = "binomial",  
 alpha = 0  
)  
  
# Apply CV Ridge regression to attrition data  
lasso <- cv.glmnet(  
 x = train\_x,  
 y = train\_y,  
 family = "binomial",  
 alpha = 1  
)  
  
# plot results  
par(mfrow = c(1, 2))  
plot(ridge, main = "Ridge penalty\n\n")  
plot(lasso, main = "Lasso penalty\n\n")



10-fold cross validation deviance for a ridge and lasso model. First dotted vertical line in each plot represents the with the smallest deviance and the second represents the with a deviance within one standard error of the minimum deviance.

Our plots above illustrate the 10-fold CV deviance across the values. With the ridge model, we don’t see any improvement in the loss function as increases but, with the lasso model, we see a slight improvement in the deviance as our penalty gets larger, suggesting that a regular logistic regression model likely overfits our data. But as we constrain it further (continue to increase the penalty), our deviance starts to increase. The numbers at the top of the plot refer to the number of variables in the model. Ridge regression does not force any variables to exactly zero so all features will remain in the model but we see the number of variables retained in the lasso model go down as our penalty increases, with the optimal model containing between 38-48 predictor variables.

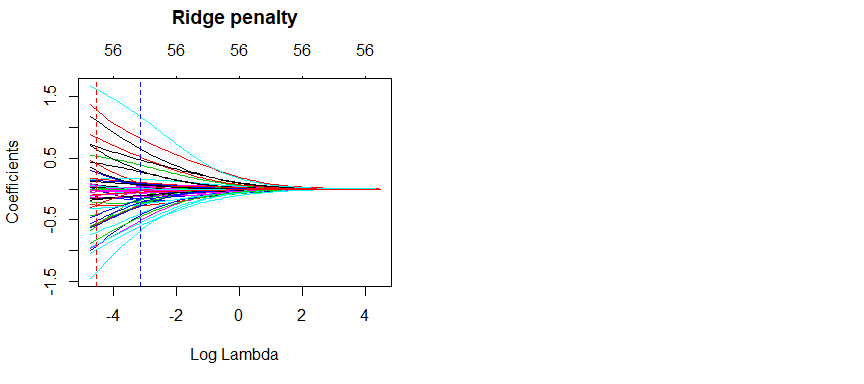
The first and second vertical dashed lines represent the value with the minimum deviance and the largest value within one standard error of the minimum deviance.

# Ridge model  
min(ridge$cvm) # minimum deviance  
## [1] 0.6510459  
ridge$lambda.min # lambda for this min deviance  
## [1] 0.0104184  
  
ridge$cvm[ridge$lambda == ridge$lambda.1se] # 1 st.error of min deviance  
## [1] 0.6796199  
ridge$lambda.1se # lambda for this deviance  
## [1] 0.04205924  
  
# Lasso model  
min(lasso$cvm) # minimum deviance  
## [1] 0.6547591  
lasso$lambda.min # lambda for this min deviance  
## [1] 0.001737893  
  
lasso$cvm[lasso$lambda == lasso$lambda.1se] # 1 st.error of min deviance  
## [1] 0.6787246  
lasso$lambda.1se # lambda for this deviance  
## [1] 0.005307275

We can assess this visually. Here we plot the coefficients across the values and the dashed red line represents the with the smallest deviance and the dashed blue line represents largest that falls within one standard error of the minimum deviance. This shows you how much we can constrain the coefficients while still maximizing predictive accuracy.

Above, we saw that both ridge and lasso penalties provide similiar minimum deviance scores; however, these plots illustrate that ridge is still using all 57 variables whereas the lasso model can get a similar deviance by reducing our feature set from 57 down to 48. However, there will be some variability with this MSE and we can reasonably assume that we can achieve a similar MSE with a slightly more constrained model that uses only 38 features. Although this lasso model does not offer significant improvement over the ridge model, we get approximately the same accuracy by using only 38 features! If describing and interpreting the predictors is an important outcome of your analysis, this may significantly aid your endeavor.

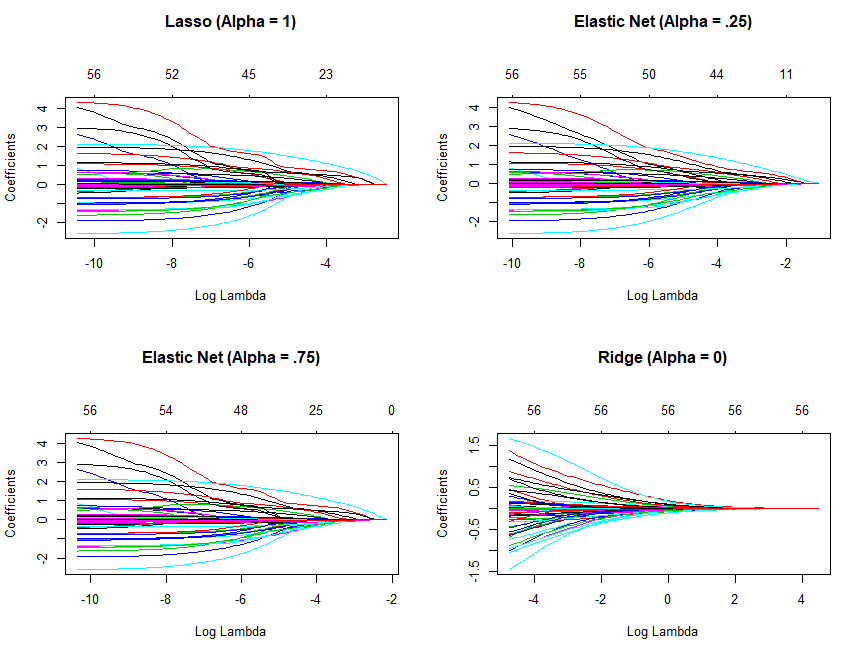
# Ridge model  
ridge\_min <- glmnet(  
 x=train\_x,  
 y=train\_y,  
 family="binomial",  
 alpha=0  
)  
  
# Lasso model  
lasso\_min <- glmnet(  
 x = train\_x,  
 y = train\_y,  
 family = "binomial",  
 alpha = 1  
)  
  
par(mfrow = c(1, 2))  
# plot ridge model  
plot(ridge\_min, xvar = "lambda", main = "Ridge penalty\n\n")  
abline(v = log(ridge$lambda.min), col = "red", lty = "dashed")  
abline(v = log(ridge$lambda.1se), col = "blue", lty = "dashed")



Coefficients for our ridge and lasso models. First dotted vertical line in each plot represents the with the smallest MSE and the second represents the with an MSE within one standard error of the minimum MSE.

So far we’ve implemented a pure ridge and pure lasso model. However, we can implement an elastic net the same way as the ridge and lasso models, by adjusting the alpha parameter. Any alpha value between 0-1 will perform an elastic net. When alpha = 0.5 we perform an equal combination of penalties whereas alpha will have a heavier ridge penalty applied and alpha will have a heavier lasso penalty.

library(glmnet)  
lasso <- glmnet(train\_x, train\_y, family="binomial", alpha=1.0)  
elastic1 <- glmnet(train\_x, train\_y, family="binomial", alpha=.25)  
elastic2 <- glmnet(train\_x, train\_y, family="binomial", alpha=.75)  
ridge <- glmnet(train\_x, train\_y, family="binomial", alpha=0.0)  
  
par(mfrow = c(2, 2), mar = c(6, 4, 6, 2) + 0.1)  
plot(lasso, xvar = "lambda", main = "Lasso (Alpha = 1)\n\n\n")  
plot(elastic1, xvar = "lambda", main = "Elastic Net (Alpha = .25)\n\n\n")  
plot(elastic2, xvar = "lambda", main = "Elastic Net (Alpha = .75)\n\n\n")  
plot(ridge, xvar = "lambda", main = "Ridge (Alpha = 0)\n\n\n")



Coefficients for various penalty parameters.

Use caution when including or in the grid search. will produce a dense solution and it can be very slow (or even impossible) to compute in large *N* situations. has no penalty, so it is therefore less numerically stable and can be very slow as well as due to slower convergence. If you experience slow computation, we recommend searching across values of 0.1, 0.25, 0.5, .75, .9.

# maintain the same folds across all models   
fold\_id <- sample(1:10, size=length(train\_y), replace=T)  
  
# search across a range of alphas  
tuning\_grid <- tibble::tibble(  
 alpha=seq(0,1, by=.1),  
 dev\_min=NA,  
 dev\_1se=NA,  
 lambda\_min=NA,  
 lambda\_1se=NA  
)

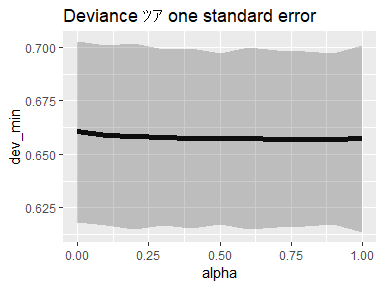
Now we can iterate over each alpha value, apply a CV elastic net, and extract the minimum and one standard error deviance values and their respective values.

**Notes** This grid search took 36 seconds.

for (i in seq\_along(tuning\_grid$alpha)){  
 # fit CV model for each alpha value  
 fit <- cv.glmnet(  
 train\_x,   
 train\_y,   
 family = "binomial",  
 alpha = tuning\_grid$alpha[i],   
 foldid = fold\_id  
 )  
   
 # extract MSE and lambda values  
 tuning\_grid$dev\_min[i] <- fit$cvm[fit$lambda == fit$lambda.min]  
 tuning\_grid$dev\_1se[i] <- fit$cvm[fit$lambda == fit$lambda.1se]  
 tuning\_grid$lambda\_min[i] <- fit$lambda.min  
 tuning\_grid$lambda\_1se[i] <- fit$lambda.1se  
}  
  
tuning\_grid  
## # A tibble: 11 x 5  
## alpha dev\_min dev\_1se lambda\_min lambda\_1se  
## <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 0 0.661 0.703 0.00949 0.0735   
## 2 0.1 0.659 0.701 0.00685 0.0441   
## 3 0.2 0.658 0.702 0.00546 0.0320   
## 4 0.3 0.658 0.699 0.00481 0.0234   
## 5 0.4 0.658 0.699 0.00396 0.0192   
## 6 0.5 0.657 0.697 0.00381 0.0154   
## 7 0.6 0.657 0.700 0.00318 0.0141   
## 8 0.7 0.657 0.699 0.00299 0.0121   
## 9 0.8 0.657 0.698 0.00287 0.0106   
## 10 0.9 0.657 0.697 0.00255 0.00939  
## 11 1 0.657 0.701 0.00230 0.00927

If we plot the deviance of one standard error for the optimal value for each alpha setting. we see that all fall withing the same level of accruacy. Consequently, we could select a full lasso model (alpha=1) with , gain the benefits of its feature selection capability and reasonably assume no loss in accruacy.

tuning\_grid %>%  
 mutate(se = dev\_1se - dev\_min) %>%  
 ggplot(aes(alpha, dev\_min)) +  
 geom\_line(size = 2) +  
 geom\_ribbon(aes(ymax = dev\_min + se, ymin = dev\_min - se), alpha = .25) +  
 ggtitle("Deviance ﾂｱ one standard error")



MSE ﾂｱ one standard error for different alpha penalty parameters.

#### Feature interpretation (#classification-binary-glmnet-visualizing)

##### variable importance

Similar to the regularized regression models, regularized classification models **assume a monotonic linear relathionship** between the predictor variables and the response. The primary difference is in what the linear relationship means. For binary classification models, the linear relationship part of that statement just means, for a given predictor variable, it assumes for every one unit change in a given predictor variable there is a constant change in the *log-odds probability* of the response variable. This constant change is represented by the given coefficient for a predictor.

**Log odds** are an alternate way of expressing probabilities. The *odds* of a positive outcome are simply represented as . So if there is a .2 probability of an employee attriting, the odds ratio is . Consequently, the log odds for this employee is . An employee with 50% change of attriting has a log odds of so negative log odds means greater probability of not attriting and positive log odds means greater probability of attriting.

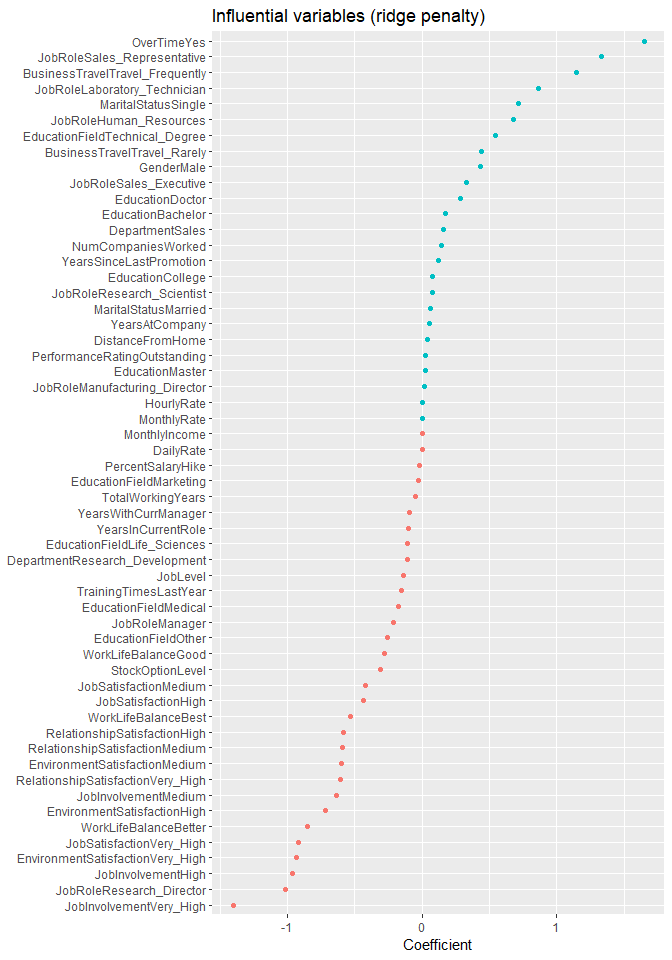
Consequently, this makes understanding variable relationships simple with regularized models. Those variables with largest positive coefficients have the strongest influence on increasing the probability of attrition whereas those variables with the largest negative coefficients have the stongest influence on decreasing the probability of attrition.

I’ll illustrate with the following models:

lasso <- cv.glmnet(train\_x, train\_y, family = "binomial", alpha = 1)  
ridge <- cv.glmnet(train\_x, train\_y, family = "binomial", alpha = 0)

Our ridge model will retain all variables. Therefore, a ridge model is good if you believe there is a need to retain all features in your model yet reduce the noise that less influential variables may create and minimize multicollinearity. However, a ridge model does not perform feature selection but it will typically push most variables to near zero and allow the most influential variables to be more prominent. The following extracts the coefficients of our ridge model and plots predictor coefficients. You can see that some of the variables have coefficients closer to zero but there are also many variables that have a strong positive influence on the probability of attrition (i.e. OverTimeYes, JobRoleSales\_Representative, BusinessTravelTravel\_Frequently) and there are others that have a strong negative influence on the probability of attrition (i.e. JobInvolvementVery\_High, JobRoleResearch\_Director, JobInvolvementHigh).

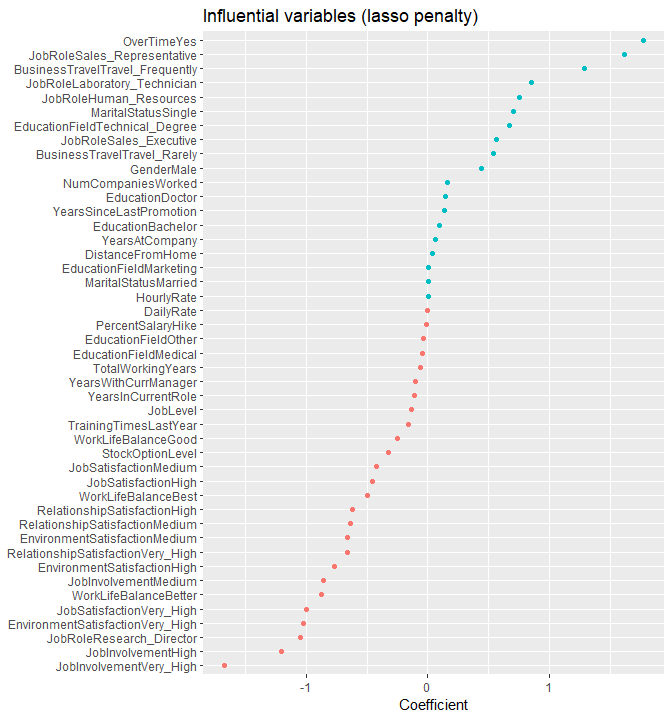
coef(ridge, s = "lambda.min") %>%  
 broom::tidy() %>%  
 filter(row != "(Intercept)") %>%  
 ggplot(aes(value, reorder(row, value), color = value > 0)) +  
 geom\_point(show.legend = FALSE) +  
 ggtitle("Influential variables (ridge penalty)") +  
 xlab("Coefficient") +  
 ylab(NULL)



A ridge penalty will retain all variables but push many coefficients to near zero. Consequently, we retain any minor signals that all features provide but those coefficients with the largest absolute values represent the most influential predictors in our model.

Similar to ridge, the lasso pushes many of the collnear features towards each other than allowing for one to be wildly positive and the other wildly negative. However, unlike ridge, the lassso will actually push coefficients to zero and perform feature selection. This simplifies and automates the process of identifying those features most influential to predictive accuracy. If we select the model with the minimum deviance and plot all variables in that model we see similar results to the ridge model regarding the most influential variables. However, the lasso model has pushed 9 variables to have zero coefficients and has retained the remaining 48, effectively performing automated feature selection.

coef(lasso, s = "lambda.min") %>%  
 broom::tidy() %>%  
 filter(row != "(Intercept)") %>%  
 ggplot(aes(value, reorder(row, value), color = value > 0)) +  
 geom\_point(show.legend = FALSE) +  
 ggtitle("Influential variables (lasso penalty)") +  
 xlab("Coefficient") +  
 ylab(NULL)



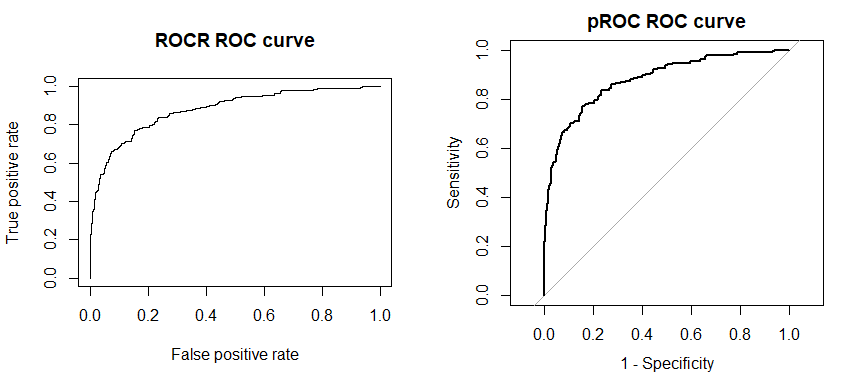
A lasso penalty will perform feature selection by pushing coefficients to zero. Consequently, we can view all coefficients to see which features were selected; however, our objective usually is still to identify those features with the strongest signal (largest absolute coefficient values).

##### ROC curve

We can visualize the ROC curve with the ROCR and pROC packages. Both packages compare the predicted log-odds output (pred) to the actual observed class. ROC curves become more interesting and useful when we compare multiple models, which we will see in later chapters.

**Notes** Ifyou do not include legacy.axes=TRUE, in the plot() call for the pROC curve, your x-axis will be revealed.

library(pROC)  
library(ROCR)  
  
# predict  
pred <- predict(ridge, s=ridge$lambda.min, train\_x)  
  
# plot structure  
par(mfrow=c(1,2))  
  
# ROCR plot  
prediction(pred, train\_y) %>%   
 performance("tpr","fpr") %>%   
 plot(main="ROCR ROC curve")  
  
# pROC plot  
roc(train\_y, as.vector(pred)) %>%   
 plot(main="pROC ROC curve", legacy.axes=T)



#### Predicting

Once you have identified your preferred model, you can simply use predict to predict the same model on a new data set. Two caveats:

1. You need to supply predict an s parameter with the preferred model’s value. For example, here we create a lasso model, which provides me a minimum deviance of 0.648. We use the for this model by specifying s=cv\_lasso$lambda.min
2. The default predicted values are the log odds. If you want the probability, include type=response.
3. If you want to predict the categorical response, include type="class".

library(glmnet)  
# optimal model  
cv\_lasso <- cv.glmnet(train\_x, train\_y, family="binomial", alpha=1.0)  
min(cv\_lasso$cvm)  
## [1] 0.6484555  
# predict and get log-odds  
pred\_log\_odds <- predict(cv\_lasso, s = cv\_lasso$lambda.min, test\_x)  
head(pred\_log\_odds)  
## 1  
## 2 -3.771081321  
## 3 0.255075052  
## 14 -3.175544046  
## 25 -2.129040504  
## 39 0.008148713  
## 46 -4.469012674  
  
# predict probability  
pred\_probs <- predict(cv\_lasso, s = cv\_lasso$lambda.min, test\_x, type = "response")  
head(pred\_probs)  
## 1  
## 2 0.02250884  
## 3 0.56342525  
## 14 0.04009649  
## 25 0.10630611  
## 39 0.50203717  
## 46 0.01132881  
  
# predict and get predicted class  
pred\_class <- predict(cv\_lasso, s = cv\_lasso$lambda.min, test\_x, type = "class")  
head(pred\_class)  
## 1   
## 2 "0"  
## 3 "1"  
## 14 "0"  
## 25 "0"  
## 39 "1"  
## 46 "0"

Lastly, to assess various performance metrics on our test data we can use caret::confusionMatrix, which provides the majority of the performance measures we are typically concerned with in classification models. We can see that the no information rate is 0.8396. This represents the ratio of non-attrition to attrition rates. The goal is to increase prediction accuracy over and above this rate. We see that our overall accuracy is 0.887. The primary weakness in our model is that for many employees that attrit, we tend to predict non-attrit. This is illustrated by our low sensitivity.

The positive argument allows you to specify which value corresponds to the “positive” result. This will impact how you interpret certain metrics that are based on true positive and false negative results (i.e. sensitivity, specificity).

caret::confusionMatrix(factor(pred\_class), factor(test\_y),positive="1")  
## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction 0 1  
## 0 236 24  
## 1 10 23  
##   
## Accuracy : 0.884   
## 95% CI : (0.8416, 0.9183)  
## No Information Rate : 0.8396   
## P-Value [Acc > NIR] : 0.02009   
##   
## Kappa : 0.5102   
## Mcnemar's Test P-Value : 0.02578   
##   
## Sensitivity : 0.4894   
## Specificity : 0.9593   
## Pos Pred Value : 0.6970   
## Neg Pred Value : 0.9077   
## Prevalence : 0.1604   
## Detection Rate : 0.0785   
## Detection Prevalence : 0.1126   
## Balanced Accuracy : 0.7244   
##   
## 'Positive' Class : 1   
##

### h2o

To perform regularized logistic regression with **h2o**, we first need to initiate our **h2o** session.

h2o::h2o.no\_progress()  
h2o::h2o.init(max\_mem\_size="5g")  
## Connection successful!  
##   
## R is connected to the H2O cluster:   
## H2O cluster uptime: 4 seconds 956 milliseconds   
## H2O cluster timezone: Asia/Tokyo   
## H2O data parsing timezone: UTC   
## H2O cluster version: 3.20.0.8   
## H2O cluster version age: 2 months and 11 days   
## H2O cluster name: H2O\_started\_from\_R\_KojiKM.Mizumura\_uwi630   
## H2O cluster total nodes: 1   
## H2O cluster total memory: 1.96 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 3.5.0 (2018-04-23)

Next, we do not need to one-hot or standardize our variables as **h2o** will do this for us. However, we do need to convert our training and test data to **h2o** objects.

# create training and test ses  
library(rsample)  
  
set.seed(123)  
split <- initial\_split(attrition, prop=.8, strata="Attrition")  
train <- training(split)  
test <- testing(split)  
  
# convert training data to h2o object  
train\_h2o <- as.h2o(train)  
  
# convert test data to h2o object  
test\_h2o <- as.h2o(test)  
  
# set the response column to Attrition  
response <- "Attrition"  
  
# set the predictor names  
predictors <- setdiff(colnames(train), "Attrition")

#### Basic implementation

Similar to our regression problem, we use h2o.glm to perform a regularized logistic regression model. The primary difference is that we need to set family=binomial to signal a inary classification problem. As before, by default, h2o.glm performs an elastic net model with alpha=.5 and will perform an automated search across internally generated lambda values.

The following performs a default h2o.glm model with alpha=.5 and it performs a 10 fold cross validation (nfolds=10).

# train your model, where you specify alpha (performs 10-fold CV)  
  
h2o\_fit1 <- h2o.glm(  
 x = predictors,   
 y = response,   
 training\_frame = train\_h2o,  
 nfolds = 10,  
 keep\_cross\_validation\_predictions = TRUE,  
 alpha = .5,   
 family = "binomial"  
)  
  
# print the MSE and AIC for the validation data   
h2o.mse(h2o\_fit1, xval = TRUE)  
## [1] 0.09739988  
  
h2o.auc(h2o\_fit1, xval=TRUE)  
## [1] 0.8345038

We can check out the cross-validated performance results of our model with h2o.performance. You can also get more results information using summary(h2o\_fit1).

h2o.performance(h2o\_fit1, xval=TRUE)  
  
## H2OBinomialMetrics: glm  
## \*\* Reported on cross-validation data. \*\*  
## \*\* 10-fold cross-validation on training data (Metrics computed for combined holdout predictions) \*\*  
##   
## MSE: 0.09739988  
## RMSE: 0.3120895  
## LogLoss: 0.3347975  
## Mean Per-Class Error: 0.258244  
## AUC: 0.8345038  
## Gini: 0.6690076  
## R^2: 0.2804837  
## Residual Deviance: 788.1133  
## AIC: 914.1133  
##   
## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:  
## 0 1 Error Rate  
## 0 898 89 0.090172 =89/987  
## 1 81 109 0.426316 =81/190  
## Totals 979 198 0.144435 =170/1177  
##   
## Maximum Metrics: Maximum metrics at their respective thresholds  
## metric threshold value idx  
## 1 max f1 0.351656 0.561856 156  
## 2 max f2 0.127301 0.631939 267  
## 3 max f0point5 0.497019 0.608696 105  
## 4 max accuracy 0.728176 0.875106 54  
## 5 max precision 0.989718 1.000000 0  
## 6 max recall 0.001056 1.000000 396  
## 7 max specificity 0.989718 1.000000 0  
## 8 max absolute\_mcc 0.497019 0.478350 105  
## 9 max min\_per\_class\_accuracy 0.140575 0.757852 257  
## 10 max mean\_per\_class\_accuracy 0.234064 0.762571 209

#### Tuning

Next, we’ll use h2o.grid to perform our grid search. The results show that peformed best: however, the improvmenet is marginal.

**Notes** This grid search took 7 seconds.

# create hyperparameter grid  
hyper\_params <- list(alpha = seq(0, 1, by = .1))  
  
# perform grid search  
grid <- h2o.grid(  
 x = predictors,   
 y = response,   
 training\_frame = train\_h2o,   
 nfolds = 10,  
 keep\_cross\_validation\_predictions = TRUE,  
 algorithm = "glm",  
 family = "binomial",  
 grid\_id = "grid\_search\_glm\_classification",   
 hyper\_params = hyper\_params  
)  
  
# sort the grid models by MSE  
sorted\_grid <- h2o.getGrid("grid\_search\_glm\_classification", sort\_by = "logloss", decreasing = FALSE)  
sorted\_grid  
## H2O Grid Details  
## ================  
##   
## Grid ID: grid\_search\_glm\_classification   
## Used hyper parameters:   
## - alpha   
## Number of models: 11   
## Number of failed models: 0   
##   
## Hyper-Parameter Search Summary: ordered by increasing logloss  
## alpha model\_ids logloss  
## 1 [0.1] grid\_search\_glm\_classification\_model\_1 0.32729191697214505  
## 2 [0.4] grid\_search\_glm\_classification\_model\_4 0.3308752559420129  
## 3 [0.2] grid\_search\_glm\_classification\_model\_2 0.33248215525410824  
## 4 [0.8] grid\_search\_glm\_classification\_model\_8 0.33249327956862135  
## 5 [0.7] grid\_search\_glm\_classification\_model\_7 0.3348307314321876  
## 6 [0.5] grid\_search\_glm\_classification\_model\_5 0.3354479575686281  
## 7 [1.0] grid\_search\_glm\_classification\_model\_10 0.33563334658821187  
## 8 [0.3] grid\_search\_glm\_classification\_model\_3 0.33779138839826356  
## 9 [0.0] grid\_search\_glm\_classification\_model\_0 0.338497245432312  
## 10 [0.6] grid\_search\_glm\_classification\_model\_6 0.33925009415446955  
## 11 [0.9] grid\_search\_glm\_classification\_model\_9 0.33927152577597053

We can check out more details of the best performing model. Our AUC (.84) is no better than our default cross-validated model. We can also extract other model parameters, which show the optimal value for our model was 0.0006.

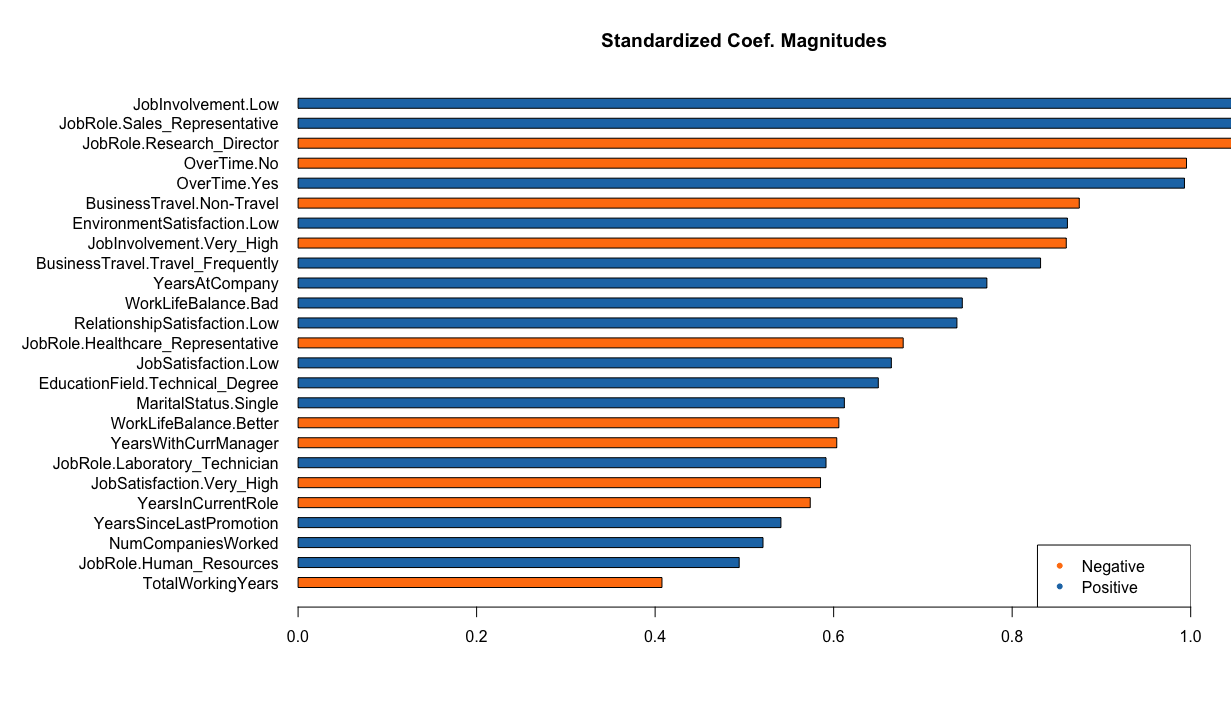
# grab top model id  
best\_h2o\_model <- sorted\_grid@model\_ids[[1]]  
best\_model <- h2o.getModel(best\_h2o\_model)  
  
# assess performance  
h2o.mse(best\_model, xval = TRUE)  
## [1] 0.09538328  
  
h2o.auc(best\_model, xval = TRUE)  
## [1] 0.8384072  
  
# get optimal parameters  
best\_model@parameters$lambda  
## [1] 0.0006111815  
  
best\_model@parameters$alpha  
## [1] 0.1

#### Feature interpreation

##### Feature importance

To identify the most influential variables, we can use **h2o**’s variable importance plot. Recall that for a GLM model, variable importance is simply represented by the standardized coefficients. We see that JobInvolvement.Low and JobRole.Sales\_representative have the largest influence in increasing the probability of attrition whereas JobRole.Research\_Director and OverTime.No have the largest influence in decreasing the probability of attrition.

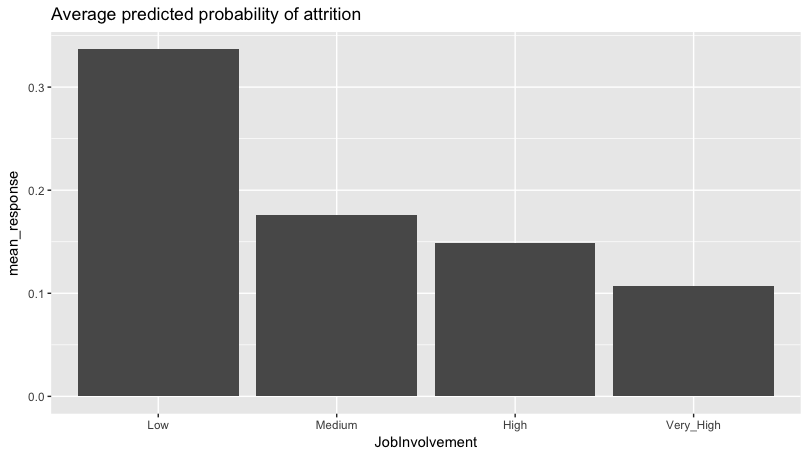
# get top 25 influential variables  
h2o.varimp\_plot(best\_model, num\_of\_features = 25)



H2O’s variable importance plot. Provides the same output as plotting the standardized coefficients (h2o.std\_coef\_plot).

To illustrate how the predicted response changes based on these influential variables, we can leverage the partial dependence inforamtion. For example, we can assess the partial dependence plots of the JobInvolvement predictor which shows up at the top of our variable importance plot. We see that mean predicted probability of attrition increases as job involvement decreases; but we can also see the significant increase in probability when an employee has a low job involvement.

h2o.partialPlot(best\_model, data=train\_h2o, cols="JobInvolvement", plot=F) %>%   
 as.data.frame() %>%   
 mutate(JobInvolvement=factor(JobInvolvement, levels=levels(attrition$JobInvolvement))) %>%   
 ggplot(aes(JobInvolvement, mean\_response))+  
 geom\_col()+  
 ggtitle("Average predicted probability of attrition")

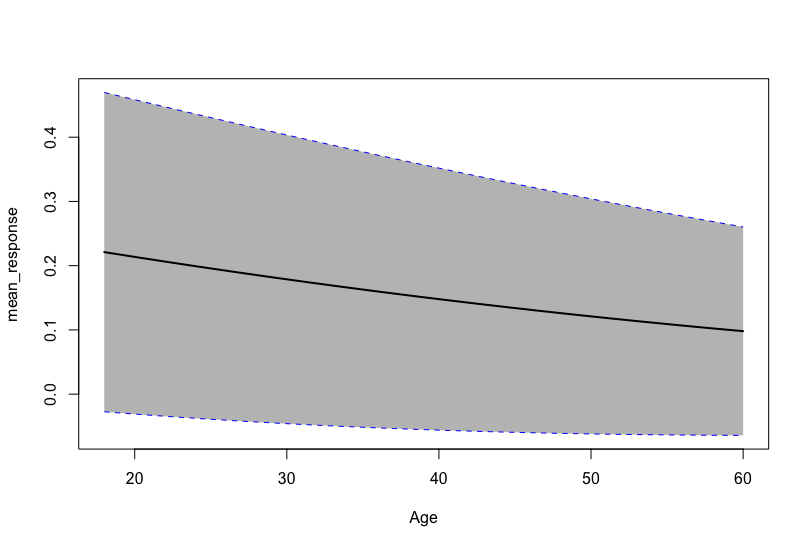


There is a significant increase in the predicted probability of attrition as an employee’s level of job involvement decreases to a low level.

Similarly, for a continous variable, we can assess the PDP. For example, age is one of the few continuous variables in this data set and we see that our regularized logistic regression model predicts a continuous decreasing probability of attirtion as employees get older.

h2o.partialPlot(best\_model, data=train\_h2o, cols="Age")

Similarly, for a continuous variable, we can assess the PDP. For example, age is one of the few continuous predictor variables in this data set and we see that our regularized logistic regression model predicts a continuously decreasing probability of attrition as employees get older.

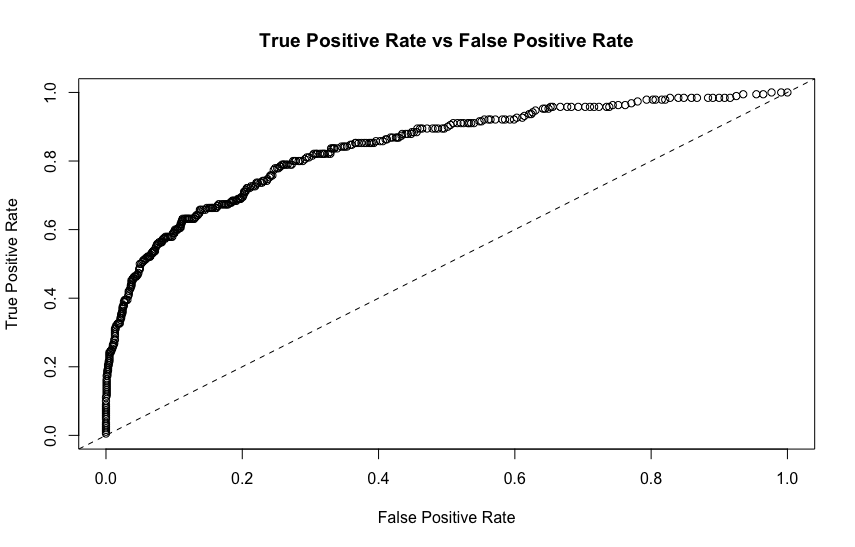


As an employee gets older, the predicted probability of attrition decreases.

##### ROC curve

Earlier, we saw that this model produced a 10-fold CV AUC of .84. We can visualize this by plotting the ROC curve using the following:

h2o.performance(best\_model, xval=T) %>%   
 plot()



ROC curve for our best performing regularized H2O GLM model.

#### PRedicting

Lastly, we can use h2o.predict and h2o.performance to predict and evaluate our models performance on our hold out test data. Note how the h2o.predict function provides 3 columns - the the predicted class and the probability of each class. We also produce our test set performance results with h2o.performance. If you compare the confusion matrix with the one produced by **glmnet** you will notice they produce very similar results. The **h2o** model does a slightly better job predicting true attrition but also does slightly worse in false positives.

pred <- h2o.predict(object=best\_model, newdata=test\_h2o)  
head(pred)  
## predict p0 p1  
## 1 0 0.9921441 0.007855913  
## 2 1 0.4230720 0.576927982  
## 3 0 0.9724310 0.027569028  
## 4 0 0.9093784 0.090621612  
## 5 1 0.4451610 0.554838982  
## 6 0 0.9863366 0.013663436  
  
# assess performance  
h2o.performance(best\_model, newdata = test\_h2o)  
## H2OBinomialMetrics: glm  
##   
## MSE: 0.0936628  
## RMSE: 0.3060438  
## LogLoss: 0.328802  
## Mean Per-Class Error: 0.2543678  
## AUC: 0.8418959  
## Gini: 0.6837917  
## R^2: 0.3045444  
## Residual Deviance: 192.678  
## AIC: 324.678  
##   
## Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:  
## 0 1 Error Rate  
## 0 236 10 0.040650 =10/246  
## 1 22 25 0.468085 =22/47  
## Totals 258 35 0.109215 =32/293  
##   
## Maximum Metrics: Maximum metrics at their respective thresholds  
## metric threshold value idx  
## 1 max f1 0.499766 0.609756 34  
## 2 max f2 0.215378 0.690299 79  
## 3 max f0point5 0.600664 0.677419 26  
## 4 max accuracy 0.600664 0.890785 26  
## 5 max precision 0.991619 1.000000 0  
## 6 max recall 0.003581 1.000000 274  
## 7 max specificity 0.991619 1.000000 0  
## 8 max absolute\_mcc 0.499766 0.555889 34  
## 9 max min\_per\_class\_accuracy 0.215378 0.787234 79  
## 10 max mean\_per\_class\_accuracy 0.215378 0.806219 79

# shutdown h2o  
h2o.removeAll()  
## [1] 0  
h2o.shutdown(prompt = FALSE)

# Implementation: Multinomial Classification

To illustrate various regularization concepts for a multinomial classification problem we will use the mnist data, where the goal is to predict handwritten numbers ranging from 0-9.

! why error?

# import mnist training and testing data  
# train <- data.table::fread("..data/mnist\_train.csv", data.table = FALSE)  
# test <- data.table::fread("../data/mnist\_test.csv", data.table = FALSE)  
  
library(data.table)  
  
# https://h2o-public-test-data.s3.amazonaws.com/bigdata/laptop/mnist/train.csv.gz  
train <- data.table::fread("C:/Users/kojikm.mizumura/Desktop/Data Science/Hands-on ML/data/mnist\_train.csv", data.table = FALSE)  
  
# load test data https://h2o-public-test-data.s3.amazonaws.com/bigdata/laptop/mnist/test.csv.gz  
test <- data.table::fread("C:/Users/kojikm.mizumura/Desktop/Data Science/Hands-on ML/data/mnist\_test.csv", data.table = FALSE)

### glmnet

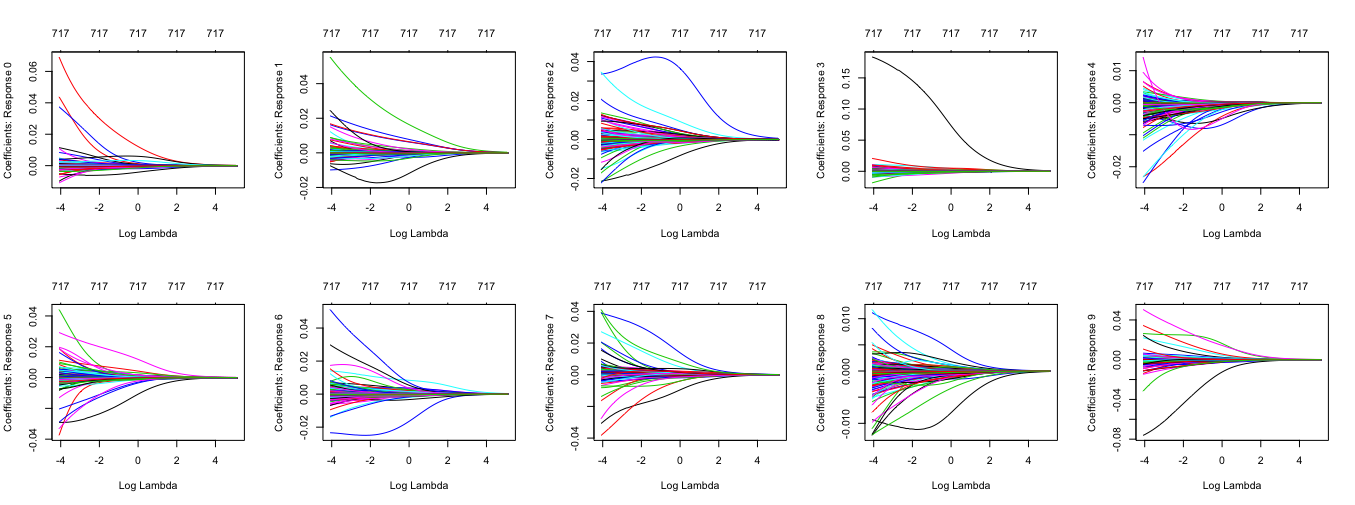
Since the mnist data contains all numeric predictors (darkness density ranging from 0-255), we do not need to one-hot encode our feature set. Consequently, we only need to convert our features into a matrix and separate the response variable (V785). For a multinomial problem our response needs to be either discrete integer values or set as a factor. Since our response values are discrete integer values from 0-9 we can leave them as is.

library(magrittr)  
library(tidyverse)  
  
train\_x <- train %>% select(-V785) %>% as.matrix()  
train\_y <- train$V785  
test\_x <- test %>% select(-V785) %>% as.matrix()  
test\_y <- test$V785  
  
# check the response ratios across the train & test sets  
table(train\_y) %>% prop.table() %>% round(2)  
table(test\_y) %>% prop.table() %>% round(2)

### Basic implemenetation

To perform a regularized multinomial GLM, we simply change the family parameter to “multinomial”. In this example we perform a full ridge penalty (alpha = 0) model. One difference with a multinomial model is that when you plot the model results, rather than seeing only one plot with the coefficient values across the spectrum of values, you will get an individual plot for the coefficients for each response category.

library(glmnet)  
  
ridge <- glmnet(  
 x=train\_x,  
 y=train\_y,  
 family="multinomial",  
 alpha=0  
)  
  
par(mfrow=c(1,2))  
plot(ridge,xvar="lambda")



Coefficients for each multinomial response as grows from $0 \rightarrow\infity$.

Also, the coefficients are stored in a list separated by the response category. We can access these coefficients in a miliar manner as we didthe regression and binary classification coefficients; However, we need to index for the response category of interest. For example, the following gets the coefficicents for response category “g” (note the indexing: coef(ridge)$9).

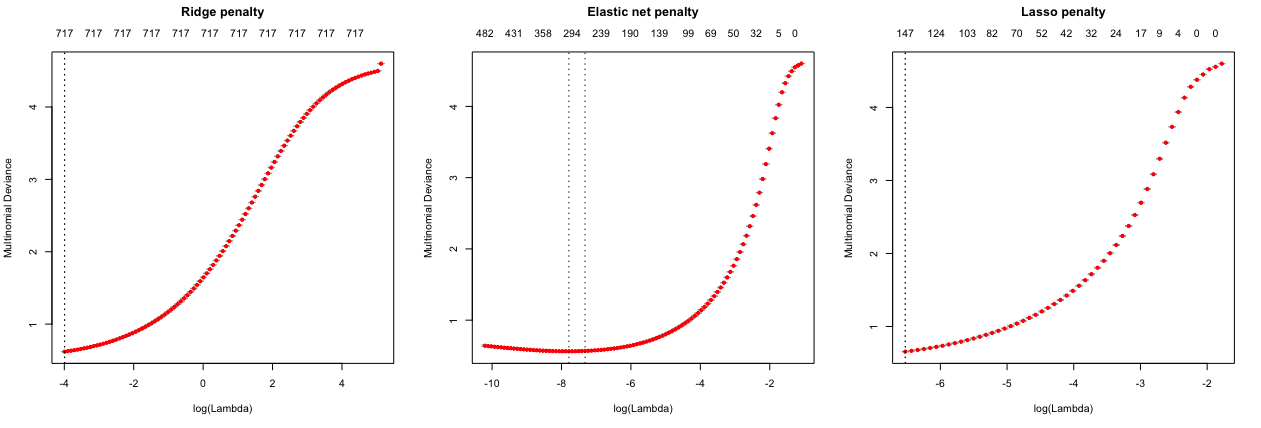
# coefficients for response "9" with small lambda penalty  
coef(ridge)$`9`[, 100] %>%   
 tidy() %>%   
 arrange(desc(abs(x)))  
  
## # A tibble: 785 x 2  
## names x  
## <chr> <dbl>  
## 1 "" -0.428   
## 2 V170 -0.0762  
## 3 V753 0.0505  
## 4 V703 0.0345  
## 5 V780 -0.0317  
## 6 V732 0.0264  
## 7 V726 0.0246  
## 8 V505 0.0220  
## 9 V421 -0.0147  
## 10 V225 -0.0139  
## # ... with 775 more rows  
  
  
# coefficients for response "9" with small lambda penalty  
coef(ridge)$`9`[, 1] %>%   
 tidy() %>%   
 arrange(desc(abs(x)))  
## # A tibble: 785 x 2  
## names x  
## <chr> <dbl>  
## 1 "" -7.12e- 3  
## 2 V753 3.18e-38  
## 3 V732 2.39e-38  
## 4 V170 -1.69e-38  
## 5 V16 -1.11e-38  
## 6 V703 8.13e-39  
## 7 V533 -6.68e-39  
## 8 V505 6.46e-39  
## 9 V33 -6.26e-39  
## 10 V752 4.64e-39  
## # ... with 775 more rows

#### Tuning

Recall that is a tuning parameter that helps to control our model from over-fitting to the training data. However, to identify the optimal vakue we need to perform cross-validation with cv.glmnet as we did with the regression and binary classification example. However, due to the magnitude of the data slowing **glmnet** down, rather than perform a full grid search across many alpha settings, we only perform a 5-fold CV glmnet model for three alpha values: 1(ridge), 0.5(elastic net), and 2(lasso).

**Notes** As your data set increases, **glmnet** begins to slow down considerably compared to **h2o**. Consequently, I reduced *k* to a 5-fold CV; however, to run a single 5-fold CV on this training set still took 109 minutes! If you parallelize the process with parallel = TRUE (which requires you to use **doMC** or some other parallelizer) you can achieve some speed improvements.

# parallelize the process  
install.packages("doMC")  
library(doMC)  
  
# apply CV ridge regression to mist data  
ridge <- cv.glmnet(  
 x=train\_x,  
 y=train\_y,  
 family="multinomial",  
 alpha=0,  
 nfolds=5,  
 parallel=TRUE  
)  
  
# Apply CV elastic net regression to mnist data  
elastic <- cv.glmnet(  
 x = train\_x,  
 y = train\_y,  
 family = "multinomial",  
 alpha = .5,  
 nfolds = 5,  
 parallel = TRUE  
)  
  
# Apply CV lasso regression to mnist data  
lasso <- cv.glmnet(  
 x = train\_x,  
 y = train\_y,  
 family = "multinomial",  
 alpha = 1,  
 nfolds = 5,  
 parallel = TRUE  
)  
  
# plot results  
par(mfrow = c(1, 3))  
plot(ridge, main = "Ridge penalty\n\n")  
plot(elastic, main = "Elastic net penalty\n\n")  
plot(lasso, main = "Lasso penalty\n\n")



5-fold cross validation deviance for a ridge, lasso, and elastic net model. First dotted vertical line in each plot represents the with the smallest deviance and the second represents the with a deviance within one standard error of the minimum deviance.

**Notes** Note how the top of the ridge penalty deviance plot indicates only 717 of the 784 predictors are in the model. The reason for this is **not** because the ridge model pushed 67 coefficients to zero because a ridge model does not do that. Rather, there are 67 predictors that have zero variance so they are not included in the model (see this with preProcess(train\_x, "zv")).

In the performance plots in Figure @ref(fig:glmnet-multi-class-ridge-lasso-cv-models-plot) it is difficult to see if the models differ in their minimum error rate (multinomial deviance). But if we look at the minimum deviance (and the largest deviance within one standard error) in the below code chunk, we see that the elastic net provides the optimal performance. Consequently, it appears we could use between 250-300 of the 784 predictors and still achieve optimal performance.

# ridge model  
min(ridge$cvm) # minimum deviance  
## [1] 0.6194693  
ridge$cvm[ridge$lambda == ridge$lambda.1se] # 1 st.error of min deviance  
## [1] 0.6194693  
  
# Elastic net model  
min(elastic$cvm) # minimum deviance  
## [1] 0.5627915  
elastic$cvm[elastic$lambda == elastic$lambda.1se] # 1 st.error of min deviance  
## [1] 0.5682305  
  
# Lasso model  
min(lasso$cvm) # minimum deviance  
## [1] 0.6557272  
lasso$cvm[lasso$lambda == lasso$lambda.1se] # 1 st.error of min deviance  
## [1] 0.6557272

So the elastic net model is minimizing the multinomial deviance loss function, but how does this translate to the accuracy of this model? We can assess the confusion matrix for this data on the training data. At first glance, the confusion matrix is difficult to discern differences. However, looking at the class statistics and specifically the *sensitivity* and *specificity* we can extract some useful insights. First, the specificity is 0.99 across all numbers indicating that the model does well in classifying non-events (basically, no number has a significantly higher false positive rate than the other numbers).

Second, looking at the sensitivity, we can see that our model does the best at accurately predicting the numbers 0, 1, and 6. However, it does worst at accurately predicting the numbers 2, 3, 5, 8, and 9. The number 8 has the lowest sensitivity rate and when we look at the confusion matrix we can see that our model often classifies the number 8 as the numbers 5, 3, and 1.

pred\_class <- predict(elastic, s=elastic$lambda.min,train\_x, type="class")  
caret::confusionMatrix(factor(pred\_class), factor(test\_y))  
  
## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction 0 1 2 3 4 5 6 7 8 9  
## 0 5780 1 26 16 11 43 27 10 28 19  
## 1 1 6580 48 28 27 20 15 25 115 25  
## 2 16 33 5452 126 24 38 33 59 56 14  
## 3 8 15 84 5547 9 145 0 19 125 78  
## 4 10 6 65 8 5506 50 30 47 26 135  
## 5 27 23 19 181 9 4882 69 10 129 36  
## 6 34 3 57 17 45 80 5714 3 36 2  
## 7 5 13 67 47 13 16 2 5911 15 157  
## 8 36 58 120 114 31 109 26 15 5251 41  
## 9 6 10 20 47 167 38 2 166 70 5442  
##   
## Overall Statistics  
##   
## Accuracy : 0.9344   
## 95% CI : (0.9324, 0.9364)  
## No Information Rate : 0.1124   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.9271   
## Mcnemar's Test P-Value : < 2.2e-16   
##   
## Statistics by Class:  
##   
## Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5 Class: 6 Class: 7 Class: 8 Class: 9  
## Sensitivity 0.97586 0.9760 0.91507 0.90475 0.94249 0.90057 0.96553 0.94350 0.89745 0.91478  
## Specificity 0.99665 0.9943 0.99262 0.99103 0.99304 0.99078 0.99488 0.99377 0.98984 0.99027  
## Pos Pred Value 0.96964 0.9558 0.93181 0.91990 0.93592 0.90659 0.95376 0.94637 0.90519 0.91186  
## Neg Pred Value 0.99735 0.9970 0.99066 0.98918 0.99379 0.99013 0.99622 0.99341 0.98893 0.99062  
## Prevalence 0.09872 0.1124 0.09930 0.10218 0.09737 0.09035 0.09863 0.10442 0.09752 0.09915  
## Detection Rate 0.09633 0.1097 0.09087 0.09245 0.09177 0.08137 0.09523 0.09852 0.08752 0.09070  
## Detection Prevalence 0.09935 0.1147 0.09752 0.10050 0.09805 0.08975 0.09985 0.10410 0.09668 0.09947  
## Balanced Accuracy 0.98625 0.9851 0.95384 0.94789 0.96776 0.94568 0.98020 0.96863 0.94365 0.95252

#### Feature interpretation

Interpreting the underlying predictor mechanisms with a multinomial problem is much like the regression and binary classification problems but with a few extra nuances. The first thing to remember is that although we started with 784 predictors, our full ridge model only used 717 because 67 predictors had zero variance. This is not unique to multinomial problems but its important to understand which variables these are because, in an organizational situation, we can assess whether or not we should continue collecting this information. In this example, they primarily represent pixels along the very edge of the images.

names(which(sapply(train,var)==0))  
## [1] "V1" "V2" "V3" "V4" "V5" "V6" "V7" "V8" "V9" "V10" "V11"   
## [12] "V12" "V17" "V18" "V19" "V20" "V21" "V22" "V23" "V24" "V25" "V26"   
## [23] "V27" "V28" "V29" "V30" "V31" "V32" "V53" "V54" "V55" "V56" "V57"   
## [34] "V58" "V83" "V84" "V85" "V86" "V112" "V113" "V141" "V142" "V169" "V477"  
## [45] "V561" "V645" "V646" "V672" "V673" "V674" "V700" "V701" "V702" "V728" "V729"  
## [56] "V730" "V731" "V755" "V756" "V757" "V758" "V759" "V760" "V781" "V782" "V783"  
## [67] "V784"

However, in a multinomial problem, each response category will not always use **all** the predictors that have variance. In fact, the below code shows that the number zero only uses 189 predictors whereas the number 3 uses 296.

**Notes** You can set type.multinomial="grouped within cv.glmnet() to force all predictors to be in or out together.

coef(elastic) %>%   
 purrr::map(tidy) %>%   
 bind\_rows(.id="response") %>%   
 count(response)  
## # A tibble: 10 x 2  
## Response n  
## <chr> <int>  
## 1 0 189  
## 2 1 206  
## 3 2 281  
## 4 3 296  
## 5 4 272  
## 6 5 264  
## 7 6 267  
## 8 7 278  
## 9 8 228  
## 10 9 283

Moreover, each response category will use the predictors in different ways. Similar to the binary classification problem, the coefficients represent the change in log-odds probability of the response variable for one unit change in the predictor. For example, below we see that for the response category “0”, a one unit increase in the darkness of the pixel represented by feature V41 causes a 0.0089 log odds increase that the response will be the number “0”. A 0.0089 log odds increase translates to probability increase.

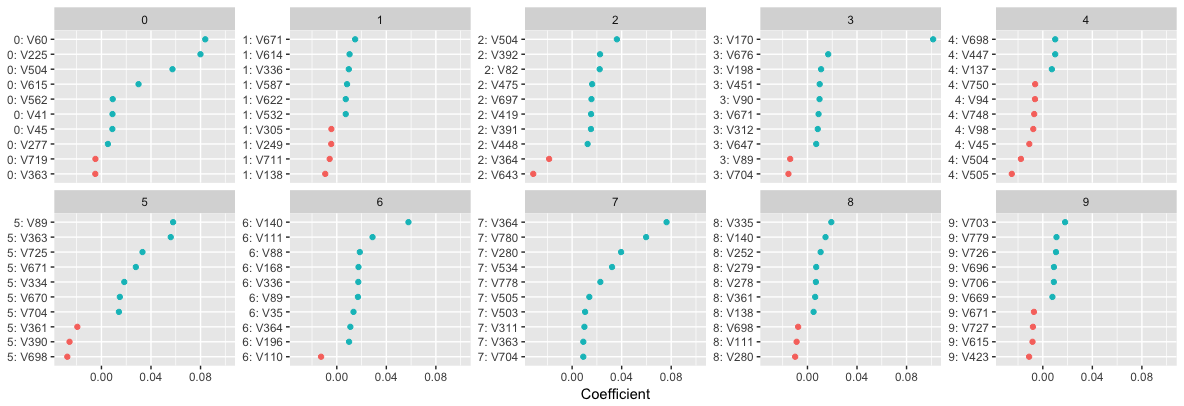
**Notes** The below code chunk extracts the coefficients for each response category, combines them all into a single data frame, and removes the intercept (since we just care about the predictor coefficients).

# tidy up the coefficients for downstream assessment  
  
vi <- coef(elastic) %>%   
 purrr::map(tidy) %>%   
 bind\_rows(.id="response") %>%   
 filter(row != "(Intercept)") %>%   
 select(response, feature=row, coefficient=value)  
  
head(vi)  
## response feature coefficient  
## 1 0 V41 8.912499e-03  
## 2 0 V45 8.816906e-03  
## 3 0 V60 8.381852e-02  
## 4 0 V72 -2.083206e-04  
## 5 0 V106 -1.630429e-03  
## 6 0 V124 1.720361e-05

Similar to the binary classification problem, one of our main concerns is to understand which predictors have the largest influence on each response category. The following identifies the top 10 predictors The following identifies the top 10 predictors with the largest absolute coefficients. These predictors represent those that have the largest impact to the probability of that response. For example, as features V60, V225, and V504 increase (those pixels become darker), they have the largest increase in the probability that the response will be “0”. Alternatively, as features V719 and V363 increase, they have the largest decrease in the probability that the response will be “0”.

**Notes** Since a given predictor can have a different coefficient for each response category, each response category can have their own unique variable importance list.

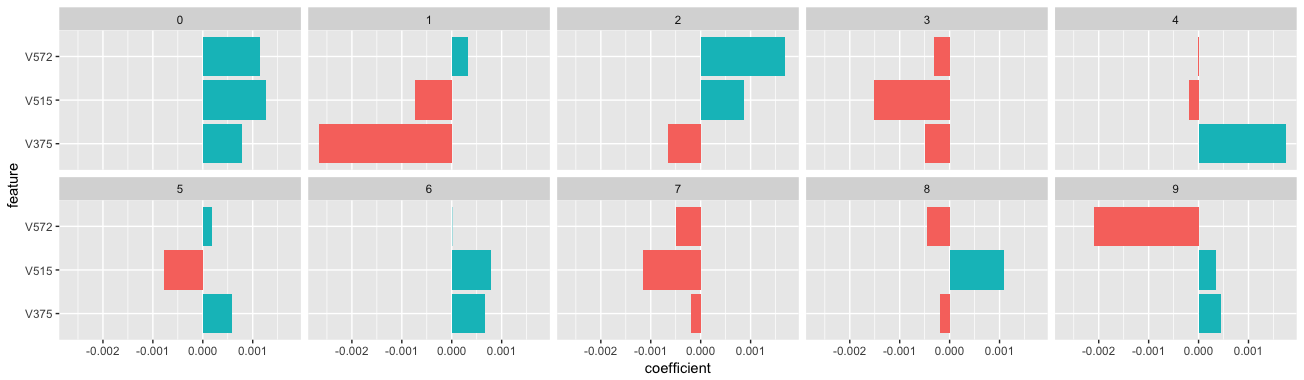
vi %>%  
 group\_by(response) %>%  
 top\_n(10, wt = abs(coefficient)) %>%  
 mutate(predictor = paste(response, feature, sep = ": ")) %>%  
 ggplot(aes(coefficient, reorder(predictor, coefficient), color = coefficient > 0)) +  
 geom\_point(show.legend = FALSE) +  
 facet\_wrap(~ response, scales = "free\_y", ncol = 5) +  
 ylab(NULL)



Top 10 most influential predictors for each multinomial response. Predictors with coefficients to the right of zero (blue) increase the probability of that response whereas predictors with coefficients to the left of zero (red) decrease the probability of that response.

The above plot identifies thse variables most influential for each given response category. However, we also want to understand which variables are influential across all, or most, of the responses. The below identifies three predictors (V375, V515, V572) that have non-zero coefficients for all the response categories. Plotting these variables allows us to see how the strength and direction of the signal varies across the response categories. These features represent pixels in the images that are used by each handwritten number.

# predictors that are influential across all or many of the responses  
vi %>%  
 count(feature) %>%  
 arrange(desc(n))  
## # A tibble: 653 x 2  
## feature n  
## <chr> <int>  
## 1 V375 10  
## 2 V515 10  
## 3 V572 10  
## 4 V402 9  
## 5 V239 8  
## 6 V268 8  
## 7 V301 8  
## 8 V324 8  
## 9 V326 8  
## 10 V353 8  
## # ... with 643 more rows  
  
vi %>%   
 filter(feature %in% c("V375", "V515", "V572")) %>%  
 ggplot(aes(coefficient, feature, fill = coefficient > 0)) +  
 geom\_col() +  
 coord\_flip() +  
 facet\_wrap(~ response, ncol = 5)



Three variables provide signals for all the response categories; however, the strength and direction of the signal can vary across the responses.

# only 82 predictors that play a role in just one response  
singles <- vi %>%  
 count(feature) %>%  
 filter(n == 1) %>%  
 .$feature  
  
vi %>%   
 filter(feature %in% singles) %>%  
 arrange(desc(abs(coefficient))) %>%  
 slice(1:10) %>%  
 mutate(prob = exp(coefficient) / (1 + exp(coefficient)))  
## response feature coefficient prob  
## 1 3 V170 0.10146013 0.5253433  
## 2 0 V60 0.08381852 0.5209424  
## 3 0 V225 0.07995005 0.5199769  
## 4 7 V780 0.05972853 0.5149277  
## 5 7 V534 0.03223708 0.5080586  
## 6 7 V778 0.02278233 0.5056953  
## 7 2 V392 0.02252209 0.5056303  
## 8 2 V82 0.02225574 0.5055637  
## 9 8 V335 0.01925485 0.5048136  
## 10 6 V88 0.01859690 0.5046491

**Notes** Remember that these linear models assume a monotonic linear relationship between the predictors and the response; meaning that for a given response category, the relationship represented by the coefficient is constant. Therefore, global and local model interpreation will be the same. See more in the machine learning interpretability chapter.

#### Predicting

Once you have identified your preferred model, you can simply use predict to predict the same model on a new data set. Similar to the binary classification problem:

1. You need to supply predict an s parameter with the preferred model’s value.
2. The default predicted values are the log odds. If you want the probability, include type = "response".
3. If you want to predict the categorical response, include type = "class".

Additionally, the predicted output is in the form of an array. Consequently, to assess the predicted values for the first five observations across all response categories, index with the following:

# predict and get log-odds  
pred\_log\_odds <- predict(elastic, s = elastic$lambda.min, test\_x)  
pred\_log\_odds[1:5, , 1]  
## 0 1 2 3 4 5 6 7 8 9  
## [1,] -5.432659 -1.339061 -2.333135 0.7305298 -3.606102 0.5498658 -6.628586 -6.8034856 6.3175820 0.353073  
## [2,] -8.302420 -5.464651 -3.410668 7.5492386 -4.806773 3.3335312 -4.941261 -0.1618101 0.7612956 2.373736  
## [3,] 7.266914 -8.224763 1.052633 1.6484586 -8.055438 -9.2497608 2.820177 -12.5006341 8.8967193 -10.473515  
## [4,] 8.415969 -10.700918 1.233072 -5.6888200 -2.364087 -0.6202491 5.928128 -7.3944607 0.2065292 -5.634506  
## [5,] -6.395731 7.228976 2.144250 -0.1299773 -4.381965 -2.0272380 -2.086397 -1.8958485 1.1838882 -2.138867  
  
# predict probability  
pred\_probs <- predict(elastic, s = elastic$lambda.min, test\_x, type = "response")  
pred\_probs[1:5, , 1]  
## 0 1 2 3 4 5 6 7 8 9  
## [1,] 7.808164e-06 4.681405e-04 0.0001732428 3.708412e-03 4.850791e-05 3.095470e-03 2.361373e-06 1.982470e-06 0.9899515761 2.542498e-03  
## [2,] 1.277183e-07 2.181128e-06 0.0000170104 9.784562e-01 4.211081e-06 1.444385e-02 3.681174e-06 4.382031e-04 0.0011029991 5.531581e-03  
## [3,] 1.633925e-01 3.056908e-08 0.0003268920 5.931549e-04 3.620925e-08 1.096811e-08 1.914428e-03 4.249083e-10 0.8337729101 3.225981e-09  
## [4,] 9.222861e-01 4.597347e-09 0.0007004437 6.906115e-07 1.919319e-05 1.097706e-04 7.663202e-02 1.254537e-07 0.0002509293 7.291585e-07  
## [5,] 1.198730e-06 9.905039e-01 0.0061317946 6.308170e-04 8.980257e-06 9.460923e-05 8.917459e-05 1.078935e-04 0.0023469721 8.461622e-05  
  
# predict and get predicted class  
pred\_class <- predict(elastic, s = elastic$lambda.min, test\_x, type = "class")  
head(pred\_class)  
## 1   
## [1,] "8"  
## [2,] "3"  
## [3,] "8"  
## [4,] "0"  
## [5,] "1"  
## [6,] "5"

Lastly, to assess various performance metrics on our test data we can use caret::confusionMatrix, which provides the majority of the performance measures we are typically concerned wit0.9281 and our model does well predicting “0”, “1”, and “6” but poorly predicting “2”, “3”, “5”,“8” and “9”.

caret::confusionMatrix(factor(pred\_class), factor(test\_y))  
## Confusion Matrix and Statistics  
##   
## Reference  
## Prediction 0 1 2 3 4 5 6 7 8 9  
## 0 959 0 7 3 1 8 10 2 6 10  
## 1 0 1114 8 1 1 2 3 9 7 8  
## 2 1 2 929 16 4 2 4 22 6 1  
## 3 1 2 15 921 1 33 2 5 21 10  
## 4 0 0 7 0 919 11 7 5 10 25  
## 5 7 1 3 29 0 779 14 0 24 5  
## 6 6 4 12 2 12 14 913 0 9 0  
## 7 5 2 11 10 6 8 3 954 10 22  
## 8 1 10 36 20 8 31 2 1 871 6  
## 9 0 0 4 8 30 4 0 30 10 922  
##   
## Overall Statistics  
##   
## Accuracy : 0.9281   
## 95% CI : (0.9229, 0.9331)  
## No Information Rate : 0.1135   
## P-Value [Acc > NIR] : < 2.2e-16   
##   
## Kappa : 0.9201   
## Mcnemar's Test P-Value : NA   
##   
## Statistics by Class:  
##   
## Class: 0 Class: 1 Class: 2 Class: 3 Class: 4 Class: 5 Class: 6 Class: 7 Class: 8 Class: 9  
## Sensitivity 0.9786 0.9815 0.9002 0.9119 0.9358 0.8733 0.9530 0.9280 0.8943 0.9138  
## Specificity 0.9948 0.9956 0.9935 0.9900 0.9928 0.9909 0.9935 0.9914 0.9873 0.9904  
## Pos Pred Value 0.9533 0.9662 0.9412 0.9110 0.9339 0.9037 0.9393 0.9253 0.8834 0.9147  
## Neg Pred Value 0.9977 0.9976 0.9886 0.9901 0.9930 0.9876 0.9950 0.9917 0.9886 0.9903  
## Prevalence 0.0980 0.1135 0.1032 0.1010 0.0982 0.0892 0.0958 0.1028 0.0974 0.1009  
## Detection Rate 0.0959 0.1114 0.0929 0.0921 0.0919 0.0779 0.0913 0.0954 0.0871 0.0922  
## Detection Prevalence 0.1006 0.1153 0.0987 0.1011 0.0984 0.0862 0.0972 0.1031 0.0986 0.1008  
## Balanced Accuracy 0.9867 0.9885 0.9469 0.9509 0.9643 0.9321 0.9733 0.9597 0.9408 0.9521

### h2o

To perform regularized multinomial logistic regression with **h2o**, we first need to initiate our **h2o** session.

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\Rtmpcru90T/h2o\_KojiKM\_Mizumura\_started\_from\_r.out  
## C:\Users\KOJIKM~1.MIZ\AppData\Local\Temp\Rtmpcru90T/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 235 milliseconds   
## H2O cluster timezone: Asia/Tokyo   
## H2O data parsing timezone: UTC   
## H2O cluster version: 3.20.0.8   
## H2O cluster version age: 2 months and 11 days   
## H2O cluster name: H2O\_started\_from\_R\_KojiKM.Mizumura\_wtv910   
## 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 3.5.0 (2018-04-23)

Since our data is already split into a training test set, to prepare for modeling, we just need to convert our training and test data to **h2o** objects and identify the response and predictor variables.

**Notes** One key difference compared to prior classification procedures, to perform a multinomial modeling with **h2o** we need to convert the response variable to a factor.

library(magrittr)  
library(tidyverse)  
  
# convert training data to h2o object  
train\_h2o <- train %>%   
 mutate(V785=factor(V785)) %>%   
 as.h2o()  
  
# convert test data to h2o object  
test\_h2o <- test %>%   
 mutate(V785=factor(V785)) %>%   
 as.h2o()  
  
# set the response column to V785  
response <- "V785"  
  
# set the predictor names  
predicotrs <- setdiff(colnames(train),reponse)

#### Basic implementation {“h2o-glm-classification”}

Similar to our regression and binary classification problem, we use h2o.glm to perform a regularized multinomial logistic regression model. The primary difference is that we need to set family=multinomial to signal a multinomial classification problem. As before, by default, h2o.glm performs an elastic net model with alpha = .5 and will perform an automated search across internally generated lambda values.

The following performs a default multinomial h2o.glm model with alpha = .5 and it performs a 5 fold cross validation (nfolds = 10).

**Notes** When you are working with large data sets, **h2o** provides a far more efficient approach for regularized models. Whereas **glmnet** took over an hour to perform a 5-fold cross validated regularized model on the **mnist** data, **h2o** took less than 2.5 minutes!

# train your model   
h2o\_fit1 <- h2o\_glm(  
 x=predictors,  
 y=response,  
 training\_frame=train\_h2o,  
 nfolds=5,  
 keep\_cross\_validation\_predictions=T,  
 alpha=.5,  
 family="multinomial"  
)

We can check out the cross-validated performance results of our model with h2o.performance. You can also get more results information using summary(h2o\_fit1). One unique output worth discussing is the “Top-10 Hit Ratios” table. The first line of this table (k = 1, hit\_ratio = 0.9213) represents the mean accuracy of our model across the 5-fold validated set. However, the second line tells us that when we take those missed predictions and use the second highest predicted probability we get a mean accuracy of 96.85% (we get an additional observations correct.) After only 4 reapplications we are able to achieve 99% accuracy.

h2o.performance(h2o\_fit1, xval=TRUE)  
## H2OMultinomialMetrics: glm  
## \*\* Reported on cross-validation data. \*\*  
## \*\* 5-fold cross-validation on training data (Metrics computed for combined holdout predictions) \*\*  
##   
## Cross-Validation Set Metrics:   
## =====================  
##   
## Extract cross-validation frame with `h2o.getFrame("filea18f736ccac9\_sid\_8725\_4")`  
## MSE: (Extract with `h2o.mse`) 0.07378137  
## RMSE: (Extract with `h2o.rmse`) 0.2716273  
## Logloss: (Extract with `h2o.logloss`) 0.2843358  
## Mean Per-Class Error: 0.07972688  
## Null Deviance: (Extract with `h2o.nulldeviance`) 276156.9  
## Residual Deviance: (Extract with `h2o.residual\_deviance`) 34193.91  
## R^2: (Extract with `h2o.r2`) 0.9911615  
## AIC: (Extract with `h2o.aic`) NaN  
## Hit Ratio Table: Extract with `h2o.hit\_ratio\_table(<model>,xval = TRUE)`  
## =======================================================================  
## Top-10 Hit Ratios:   
## k hit\_ratio  
## 1 1 0.921300  
## 2 2 0.968500  
## 3 3 0.984300  
## 4 4 0.991517  
## 5 5 0.995133  
## 6 6 0.997200  
## 7 7 0.998400  
## 8 8 0.999183  
## 9 9 0.999800  
## 10 10 1.000000

#### Tuning

Since **h2o** is much faster than **glmnet**, we can perform a grid search across a wider range of alpha parameters. Here, I assess alphas equal to o, 0.25, 0.5, 0.75 and 1 using h2o.grid to perform our grid search. The results show that performed best. The results also show that 4 out of 5 models have minor differences in the Log Loss loss function; however, (full ridge penalty) definitely performs the worst.

**Notes** This grid search took 20 minutes.

# create hyperparameter grid  
hyper\_params <- list(alpha=seq(0,1,by=.25))  
  
# perform grid search  
sorted\_grid <- h2o.getGrid("grid\_search\_glm\_multinomial",sort\_by="logloss", decreasing=FALSE)  
sorted\_grid  
## H2O Grid Details  
## ================  
##   
## Grid ID: grid\_search\_glm\_multinomial   
## Used hyper parameters:   
## - alpha   
## Number of models: 5   
## Number of failed models: 0   
##   
## Hyper-Parameter Search Summary: ordered by increasing logloss  
## alpha model\_ids logloss  
## 1 [0.25] grid\_search\_glm\_multinomial\_model\_1 0.2826176758945207  
## 2 [1.0] grid\_search\_glm\_multinomial\_model\_4 0.2833413952701668  
## 3 [0.75] grid\_search\_glm\_multinomial\_model\_3 0.2834683548256082  
## 4 [0.5] grid\_search\_glm\_multinomial\_model\_2 0.2855257887768779  
## 5 [0.0] grid\_search\_glm\_multinomial\_model\_0 0.31518212134903734

We can check out more details of the best performiing model. Our AUC (.84) is no better than our default cross-lvaidated model. We can also extract other model parameters, which show the optimal value for our model was 0.0006.

# grab top model id  
best\_h2o\_model <- sorted\_grid@model\_ids[[1]]  
best\_model <- h2o.getModel(best\_h2o\_model)  
  
# assess performance  
h2o.mse(best\_model, xval = TRUE)  
## [1] 0.0733229  
  
h2o.rmse(best\_model, xval = TRUE)  
## [1] 0.270782  
  
# get optimal parameters  
best\_model@parameters$lambda  
## [1] 0.000673668  
  
best\_model@parameters$alpha  
## [1] 0.25

We can also assess the confusion matrix for our optimal model. Our overall accuracy is 0.9146 (), which is slightly less than the mean 5-fold CV accuracy produced by the **glmnet** model (0.9344). Similar to our **glmnet** results, our optimal model is doing a good job predicting “0”, “1” and “6”, but is poorly predicting “2”, “3”,“5”, and “8”.

h2o.confusionMatrix(best\_model)  
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class  
## 0 1 2 3 4 5 6 7 8 9 Error Rate  
## 0 5776 1 14 9 10 28 36 6 38 5 0.0248 = 147 / 5,923  
## 1 1 6581 32 15 6 26 3 14 55 9 0.0239 = 161 / 6,742  
## 2 28 53 5449 85 64 16 57 67 121 18 0.0854 = 509 / 5,958  
## 3 18 31 119 5553 7 180 16 49 109 49 0.0943 = 578 / 6,131  
## 4 11 34 24 8 5504 7 44 12 30 168 0.0579 = 338 / 5,842  
## 5 50 32 31 146 52 4855 78 17 121 39 0.1044 = 566 / 5,421  
## 6 31 18 31 0 35 62 5710 4 25 2 0.0351 = 208 / 5,918  
## 7 11 35 58 16 46 8 4 5905 12 170 0.0575 = 360 / 6,265  
## 8 33 151 51 128 26 134 34 15 5211 68 0.1094 = 640 / 5,851  
## 9 25 27 11 76 142 30 3 156 38 5441 0.0854 = 508 / 5,949  
## Totals 5984 6963 5820 6036 5892 5346 5985 6245 5760 5969 0.0669 = 4,015 / 60,000

#### Feature interpretation

Similar to **glmnet**, we can extract useful information from our coefficients to interpret influential variables in our predictors. First, we need to do a little clean up of our coefficient table.

**Notes** The coefficient table provides both the coefficient estimate and the standard error; however, what follows only focuses on the coefficient estimate.

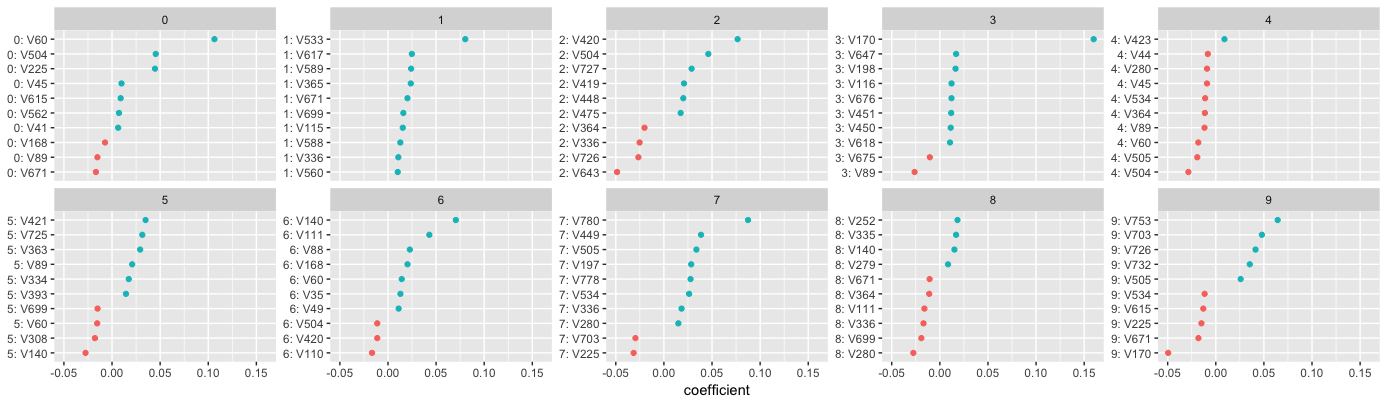
# clean up coefficient information  
vi <- best\_model@model$coefficients\_table %>%   
 as.data.frame() %>%  
 tidyr::gather(response, coefficient, -names) %>%  
 filter(  
 names != "Intercept",   
 !(stringr::str\_detect(response, "std"))) %>%  
 mutate(response = stringr::str\_remove(response, "coefs\_class\_"))  
  
head(vi)  
## names response coefficient  
## 1 V13 0 1.637796e-04  
## 2 V14 0 2.447977e-05  
## 3 V15 0 1.202897e-04  
## 4 V16 0 2.886954e-03  
## 5 V33 0 1.270802e-03  
## 6 V34 0 3.842376e-04

Unlike **glmnet**, **h2o** forces all predictors to be either in or out across all response categories (there are options to remove some of the collinear columns; however, this will still produce a consistent number of predictors across all response categories). Consequently, we see that all 717 features are present for each response.

count(vi, response)  
## # A tibble: 10 x 2  
## response n  
## <chr> <int>  
## 1 0 717  
## 2 1 717  
## 3 2 717  
## 4 3 717  
## 5 4 717  
## 6 5 717  
## 7 6 717  
## 8 7 717  
## 9 8 717  
## 10 9 717

Similar to **glmnet**, we can use this information to identify the top 10 predictors with the largest absolute coefficients for each response category. These predictors represent those that have the largest impact to the probability of that response. Comparing to the **glmnet** variable importance plots earlier, features V60, V225, and V504 are the top 3 features that have the largest increase in the probability that the response will be “0” (although their ordering differs slightly). However, the features that have the largest decrease in the probability of response “0” differ from before. This may be a result of the different penalty parameter that this optimal model has compared to what we used with **glmnet** ( versus ).

vi %>%  
 group\_by(response) %>%  
 top\_n(10, wt = abs(coefficient)) %>%  
 mutate(predictor = paste(response, names, sep = ": ")) %>%  
 ggplot(aes(coefficient, reorder(predictor, coefficient), color = coefficient > 0)) +  
 geom\_point(show.legend = FALSE) +  
 facet\_wrap(~ response, scales = "free\_y", ncol = 5) +  
 ylab(NULL)

 **Notes** Remember that these linear models assume a monotonic linear relationship between the predictors and the response; meaning that for a given response category, the relationship represented by the coefficient is constant. Therefore, global and local model interpreation will be the same. See more in the machine learning interpretability chapter.

#### Predicting

Lastly, we can use h2o.predict and h2o.performance to predict and evaluate our models performance on our hold out test data. Note how the h2o.predict function provides 11 columns - the predicted class and the probability of each class. We also produce our test set performance results with h2o.performance. If you compare the confusion matrix with the one produced by **glmnet** you will notice they produce very similar results. The **h2o** model has an overall accuracy of 0.9254 () whereas the **glmnet** model achieved an accuracy of 0.9281 on the test set. And both models have about the same level of accuracy across individual response categories.

# make predictions  
pred <- h2o.predict(object = best\_model, newdata = test\_h2o)  
head(pred)  
## predict p0 p1 p2 p3 p4 p5 p6 p7 p8 p9  
## 1 8 1.202141e-05 5.972906e-04 1.974938e-04 3.407596e-03 7.913304e-05 2.536697e-03 1.125671e-05 5.718130e-06 0.9905660926 2.586700e-03  
## 2 3 3.049209e-07 2.484003e-05 5.767582e-06 9.805161e-01 6.366074e-06 1.372732e-02 8.326104e-06 4.850814e-04 0.0009518292 4.274033e-03  
## 3 8 4.521685e-01 1.335448e-07 6.741594e-04 7.897412e-05 1.444127e-07 8.704872e-07 3.896986e-03 5.145982e-09 0.5431802637 3.238420e-09  
## 4 0 8.901589e-01 4.027749e-07 9.102679e-04 5.754058e-07 3.002048e-05 1.979771e-04 1.081991e-01 1.176814e-06 0.0005003709 1.131257e-06  
## 5 1 7.789492e-06 9.859295e-01 9.676005e-03 8.276910e-04 1.749899e-05 1.464323e-04 2.368448e-04 1.720208e-04 0.0029125347 7.370373e-05  
## 6 5 1.597256e-06 2.661786e-08 1.713028e-08 8.928673e-11 4.543192e-05 9.399342e-01 1.337817e-06 3.112696e-07 0.0600170128 2.307191e-08  
  
# assess performance  
h2o.performance(best\_model, newdata = test\_h2o)  
## H2OMultinomialMetrics: glm  
##   
## Test Set Metrics:   
## =====================  
##   
## MSE: (Extract with `h2o.mse`) 0.06887707  
## RMSE: (Extract with `h2o.rmse`) 0.2624444  
## Logloss: (Extract with `h2o.logloss`) 0.2682622  
## Mean Per-Class Error: 0.07570879  
## Null Deviance: (Extract with `h2o.nulldeviance`) 46020.38  
## Residual Deviance: (Extract with `h2o.residual\_deviance`) 5365.243  
## R^2: (Extract with `h2o.r2`) 0.9917859  
## AIC: (Extract with `h2o.aic`) NaN  
## Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, <data>)`)  
## =========================================================================  
## Confusion Matrix: Row labels: Actual class; Column labels: Predicted class  
## 0 1 2 3 4 5 6 7 8 9 Error Rate  
## 0 963 0 0 1 0 5 5 4 2 0 0.0173 = 17 / 980  
## 1 0 1112 2 2 0 1 4 2 12 0 0.0203 = 23 / 1,135  
## 2 7 9 926 17 9 3 13 9 36 3 0.1027 = 106 / 1,032  
## 3 4 1 17 921 1 24 2 11 23 6 0.0881 = 89 / 1,010  
## 4 1 3 3 2 919 0 14 3 7 30 0.0642 = 63 / 982  
## 5 8 2 1 33 10 774 17 12 31 4 0.1323 = 118 / 892  
## 6 11 3 6 1 7 15 910 3 2 0 0.0501 = 48 / 958  
## 7 2 9 22 6 6 0 0 951 1 31 0.0749 = 77 / 1,028  
## 8 7 12 6 19 11 24 11 12 861 11 0.1160 = 113 / 974  
## 9 10 8 1 10 28 5 0 24 6 917 0.0912 = 92 / 1,009  
## Totals 1013 1159 984 1012 991 851 976 1031 981 1002 0.0746 = 746 / 10,000  
##   
## Hit Ratio Table: Extract with `h2o.hit\_ratio\_table(<model>, <data>)`  
## =======================================================================  
## Top-10 Hit Ratios:   
## k hit\_ratio  
## 1 1 0.925400  
## 2 2 0.970200  
## 3 3 0.983900  
## 4 4 0.991300  
## 5 5 0.995600  
## 6 6 0.997400  
## 7 7 0.998600  
## 8 8 0.999300  
## 9 9 0.999700  
## 10 10 1.000000

# shutdown h2o  
h2o.removeAll()  
h2o.shutdown(prompt = FALSE)

## Learning More

This serves as an introduction to regularized regression; however, it just scrapes the surface. Regularized regression approaches have been extended to other parametric (i.e. Cox proportional hazard, poisson, support vector machines) and non-parametric (i.e. Least Angle Regression, the Bayesian Lasso, neural networks) models. The following are great resources to learn more (listed in order of complexity):

* [Applied Predictive Modeling](https://www.amazon.com/Applied-Predictive-Modeling-Max-Kuhn/dp/1461468485/ref=sr_1_1?ie=UTF8&qid=1522246635&sr=8-1&keywords=applied+predictive+modelling)
* [Practical Machine Learning with H2o](https://www.amazon.com/Practical-Machine-Learning-H2O-Techniques/dp/149196460X)
* [Introduction to Statistical Learning](https://www.amazon.com/Introduction-Statistical-Learning-Applications-Statistics/dp/1461471370/ref=sr_1_2?ie=UTF8&qid=1522246635&sr=8-2&keywords=applied+predictive+modelling)
* [The Elements of Statistical Learning](https://www.amazon.com/Elements-Statistical-Learning-Prediction-Statistics/dp/0387848576/ref=sr_1_3?ie=UTF8&qid=1522246635&sr=8-3&keywords=applied+predictive+modelling)
* [Statistical Learning with Sparsity](https://www.amazon.com/Statistical-Learning-Sparsity-Generalizations-Probability/dp/1498712169/ref=sr_1_1?ie=UTF8&qid=1522246685&sr=8-1&keywords=statistical+learning+with+sparsity)

1. Note that our pentalty is only applied to our feature coefficients () and not the intercept (). [↑](#footnote-ref-88)
2. The features highlighted for each package were originally identified by Erin LeDell in her [useR! 2016 tutorial](https://github.com/ledell/useR-machine-learning-tutorial). [↑](#footnote-ref-98)