UC business Analytics R programming Guide

Koji Mizumura

2019-02-21

Table of Contents

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

# Descrptive analytics

Descriptive methodologies focus on analyzing historic data of the purpose of identifying patterns or trends. Analytic techniques fall into this category are most often associated with exploratory data analysis which identifies central tendencies, variations, and distributional shapes. Descriptive methodologies can also search for underlying structures whithin data when no a *priori* knowledge about patterns and relationships are assumed.

This can include correlation analysis, exploratory factor analysis, principal component analysis, trend analyses, and cluster analysis. The following tutorial walk you through common forms of descriptive analytics.

1. Classical analyses

* [Numerical data descrptive statistics](http://uc-r.github.io/descriptives_numeric)
* Categorical data descriptive statistics
* Assumption of normality
* Assumption of homogeneity
* Assessing correlations
* Univariate statistical inference
* Multivariate statistical inference
* Bootstrapping for parameter estimates

1. Text mining

* Tidying text & word frequency
* Sentiment analysis
* Term vs Document frequency
* Word relationships
* Converting between tidy and non-tidy formats

1. Unsupervised learning

## Classical analyses

### [Numerical data descriptive statistics](http://uc-r.github.io/descriptives_numeric)

Descriptive statistics are the first piece of information used to understand and represent a dataset. There goal in essence, is to describe the main features of numerical and categorical information with simple summaries. These summaries can be presented with a single numeric measure, using summary tables, or via graphical representation. Here, I illustrate the most common forms of descriptive statistics for numerical data but keep in mind there are numerous ways to describe and illustrate key feature of data.

#### tl;dr

This tutorial covers the key features we are initially interested in understanding for numerical data, to include:

* central tendency: what are the most typical values?
* variability:How do the values vary?
* shape: Are the values symmetrically or asymetrically distributed?
* outliers: Are there values that represent abnormalities in the data?
* visualization: We should understand these features of the data through statistics and visualization.

#### Replication requirements

To illustrate ways to comute different summary statistics, and to visualize the data to provide understanding of these key features, I’ll demonstrate using this [data](https://github.com/bradleyboehmke/bradleyboehmke.github.io/blob/master/public/data/Baseball%20Salaries%202011.xlsx) which contains data on 843 MLB players in the 2011 season:

readxl::read\_xlsx("https://github.com/bradleyboehmke/bradleyboehmke.github.io/blob/master/public/data/Baseball%20Salaries%202011.xlsx")

In addition, the packages we will leverage include the following:

library(outliers) # identidying and extracting outliers  
library(ggplot2) # for generating visualization  
library(moments) # for calculating the skew and kurtosis

#### Central tendency {AA}

{AA}

# Predictive analytics

## Machine Learning

### Preparing for regression problems

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 parameters and assessing model performance.

Before introducing specific algorithms, this tutorial introduces concepts that are commonly required in the supervised machine learning process and that you will see briskly covered in tutorieals that follow. This tutorial will prepare you with the fundamentals needed prior to applying supervised machine learning algorithms.

#### tl;dr

Before introducing specific algorithms, this tutorial introduces concepts that you will see briskly covered in each chapter and are necessary for any type of supervised machine learning model:

1. Prerequisites: what you will need to reproduce the analysis in this tutorial

##### Prerequisites

This tutorial leverages the following packages.

library(rsample)  
library(caret)  
library(h2o)  
library(dplyr)  
  
# turn off progress bars  
h2o.no\_progress()  
  
# launch h2o  
h2o.init()  
## Connection successful!  
##   
## R is connected to the H2O cluster:   
## H2O cluster uptime: 23 minutes 25 seconds   
## H2O cluster timezone: Asia/Tokyo   
## H2O data parsing timezone: UTC   
## H2O cluster version: 3.22.1.1   
## H2O cluster version age: 1 month and 22 days   
## H2O cluster name: H2O\_started\_from\_R\_KojiKM.Mizumura\_hpt187   
## 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 concepts we will use the Ames Housing data that has been included in the AmesHousing package and the employee attrition data that has been included in the rsample package. The housing data represents a continuous response variable (Sale\_Price) along with 80 features (predictor variables) for 2930 homes in Ames, IA. Read more about this data [here](https://cran.r-project.org/web/packages/AmesHousing/AmesHousing.pdf). The attrition data represents a classification response variable (Attrition) with 30 features for 1470 employees. Read more about this data [here](https://www.ibm.com/communities/analytics/watson-analytics-blog/hr-employee-attrition/)

Throughout this tutorial, we will demonstrate approaches with the regular df data frame. However, since many of the supervised machine learning tutorials that we provide leverage h2o, we also show how to do some of the things with h2o. This requires your data to be in H2O object, which you can convert any data friame easily with as.h2o.

# ames data  
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 fina 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*

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.

Typically, we are lacking in the size of our data here, so a 70-30 split is often sufficinet. The two most common ways of splittin data include **simple random sampling** and **stratified sampling**.

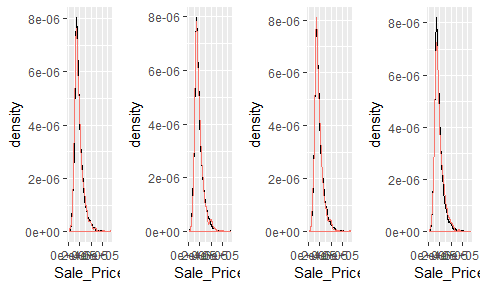
##### Simple random sampling

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 in the quantiles 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):

# base R  
df <- ames  
df.h2o <- ames.h2o  
  
set.seed(123)  
df  
## # A tibble: 2,930 x 81  
## MS\_SubClass MS\_Zoning Lot\_Frontage Lot\_Area Street Alley Lot\_Shape  
## <fct> <fct> <dbl> <int> <fct> <fct> <fct>   
## 1 One\_Story\_~ Resident~ 141 31770 Pave No\_A~ Slightly~  
## 2 One\_Story\_~ Resident~ 80 11622 Pave No\_A~ Regular   
## 3 One\_Story\_~ Resident~ 81 14267 Pave No\_A~ Slightly~  
## 4 One\_Story\_~ Resident~ 93 11160 Pave No\_A~ Regular   
## 5 Two\_Story\_~ Resident~ 74 13830 Pave No\_A~ Slightly~  
## 6 Two\_Story\_~ Resident~ 78 9978 Pave No\_A~ Slightly~  
## 7 One\_Story\_~ Resident~ 41 4920 Pave No\_A~ Regular   
## 8 One\_Story\_~ Resident~ 43 5005 Pave No\_A~ Slightly~  
## 9 One\_Story\_~ Resident~ 39 5389 Pave No\_A~ Slightly~  
## 10 Two\_Story\_~ Resident~ 60 7500 Pave No\_A~ Regular   
## # ... with 2,920 more rows, and 74 more variables: Land\_Contour <fct>,  
## # Utilities <fct>, Lot\_Config <fct>, Land\_Slope <fct>,  
## # Neighborhood <fct>, Condition\_1 <fct>, Condition\_2 <fct>,  
## # Bldg\_Type <fct>, House\_Style <fct>, Overall\_Qual <fct>,  
## # Overall\_Cond <fct>, Year\_Built <int>, Year\_Remod\_Add <int>,  
## # Roof\_Style <fct>, Roof\_Matl <fct>, Exterior\_1st <fct>,  
## # Exterior\_2nd <fct>, Mas\_Vnr\_Type <fct>, Mas\_Vnr\_Area <dbl>,  
## # Exter\_Qual <fct>, Exter\_Cond <fct>, Foundation <fct>, Bsmt\_Qual <fct>,  
## # Bsmt\_Cond <fct>, Bsmt\_Exposure <fct>, BsmtFin\_Type\_1 <fct>,  
## # BsmtFin\_SF\_1 <dbl>, BsmtFin\_Type\_2 <fct>, BsmtFin\_SF\_2 <dbl>,  
## # Bsmt\_Unf\_SF <dbl>, Total\_Bsmt\_SF <dbl>, Heating <fct>,  
## # Heating\_QC <fct>, Central\_Air <fct>, Electrical <fct>,  
## # First\_Flr\_SF <int>, Second\_Flr\_SF <int>, Low\_Qual\_Fin\_SF <int>,  
## # Gr\_Liv\_Area <int>, Bsmt\_Full\_Bath <dbl>, Bsmt\_Half\_Bath <dbl>,  
## # Full\_Bath <int>, Half\_Bath <int>, Bedroom\_AbvGr <int>,  
## # Kitchen\_AbvGr <int>, Kitchen\_Qual <fct>, TotRms\_AbvGrd <int>,  
## # Functional <fct>, Fireplaces <int>, Fireplace\_Qu <fct>,  
## # Garage\_Type <fct>, Garage\_Finish <fct>, Garage\_Cars <dbl>,  
## # Garage\_Area <dbl>, Garage\_Qual <fct>, Garage\_Cond <fct>,  
## # Paved\_Drive <fct>, Wood\_Deck\_SF <int>, Open\_Porch\_SF <int>,  
## # Enclosed\_Porch <int>, Three\_season\_porch <int>, Screen\_Porch <int>,  
## # Pool\_Area <int>, Pool\_QC <fct>, Fence <fct>, Misc\_Feature <fct>,  
## # Misc\_Val <int>, Mo\_Sold <int>, Year\_Sold <int>, Sale\_Type <fct>,  
## # Sale\_Condition <fct>, Sale\_Price <int>, Longitude <dbl>,  
## # Latitude <dbl>  
index <- sample(1:nrow(df), round(nrow(df) \* 0.7))  
train\_1 <- df[index, ]  
test\_1 <- df[-index, ]  
  
# caret package  
set.seed(123)  
index2 <- createDataPartition(df$Sale\_Price, p = 0.7, list = FALSE)  
train\_2 <- df[index2, ]  
test\_2 <- df[-index2, ]  
  
# rsample package  
set.seed(123)  
split\_1 <- initial\_split(df, prop = 0.7)  
train\_3 <- training(split\_1)  
test\_3 <- testing(split\_1)  
  
# h2o package  
split\_2 <- h2o.splitFrame(df.h2o, ratios = 0.7, seed = 123)  
train\_4 <- split\_2[[1]]  
test\_4 <- split\_2[[2]]

Since this sampling approach will randomly sample across the distribution of (Sale\_Price), you will typically result in a similar distribution betwen your training and test sets as iilustrated below.

# base R  
p1 <- ggplot()+  
 geom\_density(data=train\_1, aes(Sale\_Price), show.legend = FALSE)+  
 geom\_density(data=test\_1, aes(x=Sale\_Price, col="red"),show.legend = FALSE)  
   
# caret   
p2 <- ggplot()+  
 geom\_density(data=train\_2, aes(Sale\_Price),show.legend = FALSE)+  
 geom\_density(data=test\_2, aes(x=Sale\_Price, col="red"),show.legend = FALSE)  
  
# sample  
p3 <- ggplot()+  
 geom\_density(data=train\_3, aes(Sale\_Price),show.legend = FALSE)+  
 geom\_density(data=test\_3, aes(x=Sale\_Price, col="red"),show.legend = FALSE)  
  
# h2o  
  
p4 <- ggplot()+  
 geom\_density(data=as\_tibble(train\_4), aes(Sale\_Price),show.legend = FALSE)+  
 geom\_density(data=as\_tibble(test\_4), aes(x=Sale\_Price, col="red"),show.legend = FALSE)  
  
gridExtra::grid.arrange(p1,p2,p3,p4,  
 nrow=1)



##### Stratified sampling

However, if we want to explictly 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 also apply to regression problems for data sets that have a small sample size and where the response variable deviates strongly from *normality*.

With a continuous response variable, stratified sampling will break y down into quantiles and randomly sample from each quantile. Consequently, this will help ensure a balanced representation of the response distribution in both the 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 strata fy. 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 approximiately equal response distributions.

# original response distribution  
table(churn$Attrition) %>% prop.table()  
##   
## No Yes   
## 0.8387755 0.1612245  
  
# stratified sampling with the rsample package  
set.seed(123)  
split\_strat <- initial\_split(churn, prop=0.7, strata = "Attrition")  
train\_strat <- training(split\_strat)  
test\_strat <- testing(split\_strat)  
  
# 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 siginificant process and requires you to spend substantial time understanding your data… or as Leo Breiman said “live with your data before you plunge into modeling”

Although this guide is primarily concerned with 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.

1. One-hot encoding

Many models require all variables to be numeric. Consequently, we need to transform any categorical variables into numeric representation 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 representation (i.e., one-hot, orinal, binary, sum, Helmert).

The most common is refered 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:

sample <- tibble::tribble(  
 ~id, ~x,  
 1,"a",  
 2,"c",  
 3,"b",  
 4,"c",  
 5,"c",  
 6,"a",  
 7,"b",  
 8,"c"  
)  
  
sample %>%   
 mutate(x=as.factor(x))  
## # A tibble: 8 x 2  
## id x   
## <dbl> <fct>  
## 1 1 a   
## 2 2 c   
## 3 3 b   
## 4 4 c   
## 5 5 c   
## 6 6 a   
## 7 7 b   
## 8 8 c

results in the following representation:

If you need to manually implement one-hot encoding yourself, you can do that with caret::dummyVars. Sometimes you many 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.  
  
library(caret)  
full\_rank <- dummyVars(~., data = df, fullRank = TRUE)  
train\_oh <- predict(full\_rank, train\_1)  
test\_oh <- predict(full\_rank, test\_1)  
  
train\_1 %>% head()  
## # A tibble: 6 x 81  
## MS\_SubClass MS\_Zoning Lot\_Frontage Lot\_Area Street Alley Lot\_Shape  
## <fct> <fct> <dbl> <int> <fct> <fct> <fct>   
## 1 Two\_Story\_~ Resident~ 80 10400 Pave No\_A~ Regular   
## 2 One\_Story\_~ Resident~ 0 39384 Pave No\_A~ Slightly~  
## 3 Duplex\_All~ Resident~ 87 9246 Pave No\_A~ Slightly~  
## 4 One\_Story\_~ Resident~ 66 12778 Pave No\_A~ Regular   
## 5 One\_Story\_~ Resident~ 64 7488 Pave No\_A~ Slightly~  
## 6 One\_Story\_~ Resident~ 78 7800 Pave No\_A~ Regular   
## # ... with 74 more variables: Land\_Contour <fct>, Utilities <fct>,  
## # Lot\_Config <fct>, Land\_Slope <fct>, Neighborhood <fct>,  
## # Condition\_1 <fct>, Condition\_2 <fct>, Bldg\_Type <fct>,  
## # House\_Style <fct>, Overall\_Qual <fct>, Overall\_Cond <fct>,  
## # Year\_Built <int>, Year\_Remod\_Add <int>, Roof\_Style <fct>,  
## # Roof\_Matl <fct>, Exterior\_1st <fct>, Exterior\_2nd <fct>,  
## # Mas\_Vnr\_Type <fct>, Mas\_Vnr\_Area <dbl>, Exter\_Qual <fct>,  
## # Exter\_Cond <fct>, Foundation <fct>, Bsmt\_Qual <fct>, Bsmt\_Cond <fct>,  
## # Bsmt\_Exposure <fct>, BsmtFin\_Type\_1 <fct>, BsmtFin\_SF\_1 <dbl>,  
## # BsmtFin\_Type\_2 <fct>, BsmtFin\_SF\_2 <dbl>, Bsmt\_Unf\_SF <dbl>,  
## # Total\_Bsmt\_SF <dbl>, Heating <fct>, Heating\_QC <fct>,  
## # Central\_Air <fct>, Electrical <fct>, First\_Flr\_SF <int>,  
## # Second\_Flr\_SF <int>, Low\_Qual\_Fin\_SF <int>, Gr\_Liv\_Area <int>,  
## # Bsmt\_Full\_Bath <dbl>, Bsmt\_Half\_Bath <dbl>, Full\_Bath <int>,  
## # Half\_Bath <int>, Bedroom\_AbvGr <int>, Kitchen\_AbvGr <int>,  
## # Kitchen\_Qual <fct>, TotRms\_AbvGrd <int>, Functional <fct>,  
## # Fireplaces <int>, Fireplace\_Qu <fct>, Garage\_Type <fct>,  
## # Garage\_Finish <fct>, Garage\_Cars <dbl>, Garage\_Area <dbl>,  
## # Garage\_Qual <fct>, Garage\_Cond <fct>, Paved\_Drive <fct>,  
## # Wood\_Deck\_SF <int>, Open\_Porch\_SF <int>, Enclosed\_Porch <int>,  
## # Three\_season\_porch <int>, Screen\_Porch <int>, Pool\_Area <int>,  
## # Pool\_QC <fct>, Fence <fct>, Misc\_Feature <fct>, Misc\_Val <int>,  
## # Mo\_Sold <int>, Year\_Sold <int>, Sale\_Type <fct>, Sale\_Condition <fct>,  
## # Sale\_Price <int>, Longitude <dbl>, Latitude <dbl>  
train\_oh %>% as\_tibble() %>% head()  
## # A tibble: 6 x 308  
## MS\_SubClass.One~ MS\_SubClass.One~ MS\_SubClass.One~ MS\_SubClass.One~  
## <dbl> <dbl> <dbl> <dbl>  
## 1 0 0 0 0  
## 2 0 0 0 0  
## 3 0 0 0 0  
## 4 0 0 0 0  
## 5 0 0 0 0  
## 6 0 0 0 0  
## # ... with 304 more variables: MS\_SubClass.Two\_Story\_1946\_and\_Newer <dbl>,  
## # MS\_SubClass.Two\_Story\_1945\_and\_Older <dbl>,  
## # MS\_SubClass.Two\_and\_Half\_Story\_All\_Ages <dbl>,  
## # MS\_SubClass.Split\_or\_Multilevel <dbl>, MS\_SubClass.Split\_Foyer <dbl>,  
## # MS\_SubClass.Duplex\_All\_Styles\_and\_Ages <dbl>,  
## # MS\_SubClass.One\_Story\_PUD\_1946\_and\_Newer <dbl>,  
## # MS\_SubClass.One\_and\_Half\_Story\_PUD\_All\_Ages <dbl>,  
## # MS\_SubClass.Two\_Story\_PUD\_1946\_and\_Newer <dbl>,  
## # MS\_SubClass.PUD\_Multilevel\_Split\_Level\_Foyer <dbl>,  
## # MS\_SubClass.Two\_Family\_conversion\_All\_Styles\_and\_Ages <dbl>,  
## # MS\_Zoning.Residential\_High\_Density <dbl>,  
## # MS\_Zoning.Residential\_Low\_Density <dbl>,  
## # MS\_Zoning.Residential\_Medium\_Density <dbl>, MS\_Zoning.A\_agr <dbl>,  
## # MS\_Zoning.C\_all <dbl>, MS\_Zoning.I\_all <dbl>, Lot\_Frontage <dbl>,  
## # Lot\_Area <dbl>, Street.Pave <dbl>, Alley.No\_Alley\_Access <dbl>,  
## # Alley.Paved <dbl>, Lot\_Shape.Slightly\_Irregular <dbl>,  
## # Lot\_Shape.Moderately\_Irregular <dbl>, Lot\_Shape.Irregular <dbl>,  
## # Land\_Contour.HLS <dbl>, Land\_Contour.Low <dbl>,  
## # Land\_Contour.Lvl <dbl>, Utilities.NoSeWa <dbl>,  
## # Utilities.NoSewr <dbl>, Lot\_Config.CulDSac <dbl>,  
## # Lot\_Config.FR2 <dbl>, Lot\_Config.FR3 <dbl>, Lot\_Config.Inside <dbl>,  
## # Land\_Slope.Mod <dbl>, Land\_Slope.Sev <dbl>,  
## # Neighborhood.College\_Creek <dbl>, Neighborhood.Old\_Town <dbl>,  
## # Neighborhood.Edwards <dbl>, Neighborhood.Somerset <dbl>,  
## # Neighborhood.Northridge\_Heights <dbl>, Neighborhood.Gilbert <dbl>,  
## # Neighborhood.Sawyer <dbl>, Neighborhood.Northwest\_Ames <dbl>,  
## # Neighborhood.Sawyer\_West <dbl>, Neighborhood.Mitchell <dbl>,  
## # Neighborhood.Brookside <dbl>, Neighborhood.Crawford <dbl>,  
## # Neighborhood.Iowa\_DOT\_and\_Rail\_Road <dbl>,  
## # Neighborhood.Timberland <dbl>, Neighborhood.Northridge <dbl>,  
## # Neighborhood.Stone\_Brook <dbl>,  
## # Neighborhood.South\_and\_West\_of\_Iowa\_State\_University <dbl>,  
## # Neighborhood.Clear\_Creek <dbl>, Neighborhood.Meadow\_Village <dbl>,  
## # Neighborhood.Briardale <dbl>, Neighborhood.Bloomington\_Heights <dbl>,  
## # Neighborhood.Veenker <dbl>, Neighborhood.Northpark\_Villa <dbl>,  
## # Neighborhood.Blueste <dbl>, Neighborhood.Greens <dbl>,  
## # Neighborhood.Green\_Hills <dbl>, Neighborhood.Landmark <dbl>,  
## # Condition\_1.Feedr <dbl>, Condition\_1.Norm <dbl>,  
## # Condition\_1.PosA <dbl>, Condition\_1.PosN <dbl>,  
## # Condition\_1.RRAe <dbl>, Condition\_1.RRAn <dbl>,  
## # Condition\_1.RRNe <dbl>, Condition\_1.RRNn <dbl>,  
## # Condition\_2.Feedr <dbl>, Condition\_2.Norm <dbl>,  
## # Condition\_2.PosA <dbl>, Condition\_2.PosN <dbl>,  
## # Condition\_2.RRAe <dbl>, Condition\_2.RRAn <dbl>,  
## # Condition\_2.RRNn <dbl>, Bldg\_Type.TwoFmCon <dbl>,  
## # Bldg\_Type.Duplex <dbl>, Bldg\_Type.Twnhs <dbl>, Bldg\_Type.TwnhsE <dbl>,  
## # House\_Style.One\_and\_Half\_Unf <dbl>, House\_Style.One\_Story <dbl>,  
## # House\_Style.SFoyer <dbl>, House\_Style.SLvl <dbl>,  
## # House\_Style.Two\_and\_Half\_Fin <dbl>,  
## # House\_Style.Two\_and\_Half\_Unf <dbl>, House\_Style.Two\_Story <dbl>,  
## # Overall\_Qual.Poor <dbl>, Overall\_Qual.Fair <dbl>,  
## # Overall\_Qual.Below\_Average <dbl>, Overall\_Qual.Average <dbl>,  
## # Overall\_Qual.Above\_Average <dbl>, Overall\_Qual.Good <dbl>,  
## # Overall\_Qual.Very\_Good <dbl>, Overall\_Qual.Excellent <dbl>,  
## # Overall\_Qual.Very\_Excellent <dbl>, Overall\_Cond.Poor <dbl>,  
## # Overall\_Cond.Fair <dbl>, ...  
  
# less than full rank  
dummy <- dummyVars(~., data = df, fullRank=FALSE)  
train\_oh <- predict(dummy, train\_1)  
test\_oh <- predict(dummy, test\_1)

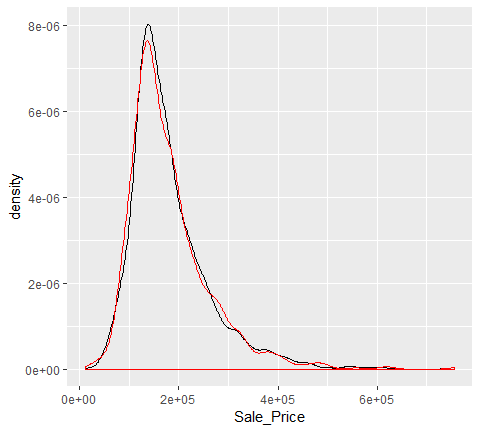
Two things to note:

* Since one-hot encoding adds new features it can significantly increase the dimensionality of our data. If you have a dataset 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 explore ordinal encoding of your data.
* if using h2o you do not need to explictly encode your categorical variables but you can override the default encoding. This can be considered a tuning parameter as some encoding 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).

##### 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.

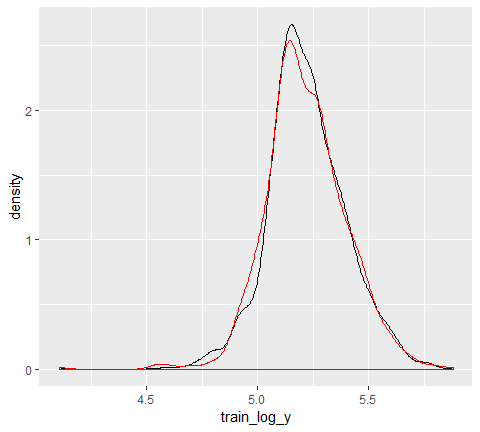
ggplot(train\_1,aes(x=Sale\_Price))+  
 geom\_density(trim=TRUE)+  
 geom\_density(data=test\_1, trim=T, col="red")



To normalize, we have two options:

**Option 1**: normalize with a \_\_log transformation\_.\_ This will transform most right skewed distributions to be approximiately normal.

# log transformation  
train\_log\_y <- log10(train\_1$Sale\_Price)  
test\_log\_y <- log10(test\_1$Sale\_Price)  
  
log\_transform <- ggplot(data = tibble(train\_log\_y), aes(train\_log\_y))+  
 geom\_density(trim=TRUE)+  
 geom\_density(data=tibble(test\_log\_y),aes(test\_log\_y), trim=T, col="red")  
log\_transform



**Option 2**: use a **Box Cox transformation**.

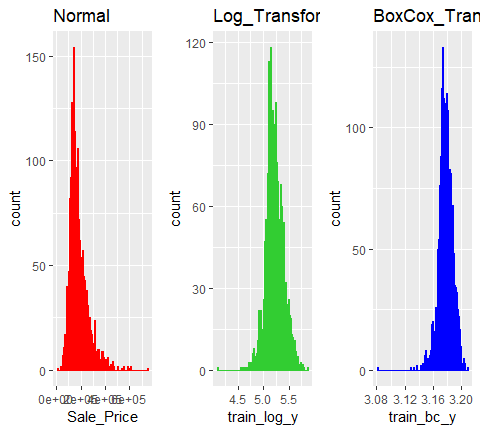
A Box Cox transformation is more flexible and will find the transformation from a family of power transformations that will transform the variable as close as possible to a normal distribution.

**Important notes**: be sure to compute the lambda on the training set and apply that same lambda to both the training and test set to minimize data leakage.

# Box cox transformation  
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 transformation both do about equally well in transforming our response variable to be normally distributed.

p1 <- ggplot(data=train\_1, aes(Sale\_Price))+  
 geom\_histogram(bins=100, col="red")+  
 ggtitle("Normal")  
  
p2 <- ggplot(data = tibble(train\_log\_y), aes(train\_log\_y))+  
 geom\_histogram(bins=100, col="lime green")+  
 ggtitle("Log\_Transform")  
  
p3 <- ggplot(data=tibble(train\_bc\_y), aes(train\_bc\_y))+  
 geom\_histogram(bins=100,col="blue")+  
 ggtitle("BoxCox\_Transform")  
  
gridExtra::grid.arrange(p1,p2,p3, nrow=1)



Note that when you 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 back to their normal state so that decision-makers can interpret the results. The following code can do this for you:

# 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)  
y  
## [1] 1.651249  
## attr(,"lambda")  
## [1] -0.3067918  
# forecast::BoxCox  
  
# inverse Box Cox function  
inv\_box\_cox <- function(x,lambda){  
 if (lambda==0) exp(x) else (lambda\*x+1)^(1/lambda)  
}  
  
# re-transfrming the Box Cox transform value  
inv\_box\_cox(y, lambda)  
## [1] 10  
## attr(,"lambda")  
## [1] -0.3067918

##### Predictor transformation

Some models such as K-NN, SVMs, PLS, neural networks require that the features have the same units. **Centering** and **scaling** can be used for this purpose and is often refered 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., h20, caret) to standardize and some do not (ie., 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 predictor variables. Note, it is important you standardize the test data based on the training mean and variance values of each feature. This minimizes data leakage.

# identify only the predictor variables  
features <- setdiff(names(train\_1), "Sale\_Price")  
  
# pre-process estimation based on training features   
pre\_process <- caret::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 transformations that you can perform:

* Normalizing the predictor variables with a *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 generalize linear models. However, this reduces the amount of information at your disposal and future tutorials show you how to use regularization as a better alternative to PCA.
* Removing *near-zero* or *zero variance variables*. Variables with vary little variance tend to not improve model performance and can be removed.
* preProcess provides other options which you can read more about [here](https://topepo.github.io/caret/pre-processing.html).

# 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("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 mddel formulation

There are **many** packages to perform machine learning and there are always more than one to perform each algorithm (i.e., there are over 20 packages to perform random forests). There are pros/cons to each package; some nay be more computationally efficient while other may have more hyperparameter tuning options.

Future tutorials will expose you to several packages; some that have become “the standard” and others that are new and may be considered “maturing”. Just realize there are more ways than one to skin.

For example, these three functions will all produce the samme linear regression model output:

lm.lm <- lm(Sale\_Price~., data = train\_1)   
lm.glm <- glm(Sale\_Price~., data=train\_1, family=gaussian)  
lm.caret <- caret::train(Sale\_Price ~., data=train\_1, method = "lm")  
  
lm\_multiple\_package = tibble(  
 model = c("lm", "glm", "caret"),  
 outcome = list(lm.lm, lm.glm, lm.caret)  
)  
  
# test <- "abcde"  
# stringr::str\_sub(test, 1,3)  
  
lm\_multiple\_package$outcome[[1]] %>%   
 broom::tidy() %>%   
 ggplot(aes(stringr::str\_sub(term, 1, 3), estimate, col=p.value<0.01))+  
 geom\_point()+  
 coord\_flip()  
  
# lm\_multiple\_package %>%   
# mutate( outcome\_tidy = purrr::map(outcome, broom::tidy))

One thing you will notice throughout future tutotiral s is that we can 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 include 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 tutorial we will use glmnet which requires our features x and response y to be specified separately:

# 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  
library(glmnet)  
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  
h2o.init()  
## Connection successful!  
##   
## R is connected to the H2O cluster:   
## H2O cluster uptime: 23 minutes 39 seconds   
## H2O cluster timezone: Asia/Tokyo   
## H2O data parsing timezone: UTC   
## H2O cluster version: 3.22.1.1   
## H2O cluster version age: 1 month and 22 days   
## H2O cluster name: H2O\_started\_from\_R\_KojiKM.Mizumura\_hpt187   
## 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)  
y <- "Sale\_Price"  
x <- setdiff(names(train\_1), y)  
train.h2o <- as.h2o(train\_1)  
  
# example of variable name specification  
h20.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 pattersn within our data For example the simple illustration below shows how the more flexible model aligns more closely to the data than fixed linear model.

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.

Throughout the future tutorial, we will demonstrate how to tune the different parameters for each model. However, we bring up this point because it feeds into the next section nicely.

##### Cross validation for generalization

Our goal is to 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, this may not generalize well to future unseen data. Consequently, our goal is to find a model and its hyperparameters that will minimize error on hold-out data.

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

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.

knitr::include\_graphics("")

Fig 8: Bias-variance tradeoff

Fig 8: Bias-variance tradeoff

To find the model that balances the *bias-variance tradeoff*, we search for a model that minimizes a *k*-fold cross-validation error metric (you will also be introduced to what’s called an *out of bag error* which provides a similar form of evaluation). *k*-fold cross-validation is a resampling method that randomly divides the training data into *k* groups (aka folds) of approximately same size. The model is fit on folds and then held-out validation fods is used to compute the error.

This process results in *k* estimates of the test error (). Thus, the *k*-fold CV estimate is computed by averagin these values, which provides us with an approximation of the error to expect on unseen data.

knitr::include\_graphics("")

Fig 9: Illustration of the k-fold cross validation process

Fig 9: Illustration of the k-fold cross validation process

Most algorithms and packages we cover in future tutorials have built-in cross-validation capabilities. One typically uses a 5 or 10 fold CV ( *k*=5 or *k*=10 ). For example, h2o implements CV with the nfolds argument:

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

#### Model evaulation

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 eorr (). 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 Root mean squared error. This simply takes the square root of the MSE metric (RMSE=$ $) so that your error is in the same units as your response variable. If your response variable units are dollars, the units of MSE are dollars-squared, but the RMSE will be in dollars. Objective: **minimize**
* **Deviance**:Short for mean residual deviance. In essence, it provides a measure of goodness-of-fit of the model being evaluated when compared to the null model (intercept only). If the response variable distribution is gaussian, then it is equal to MSE. When not, it usually gives a more useful estimate of error. **Objective: minimize**
* **MAE**:ean absolute error. Similar to MSE but rather than squaring, it just takes the mean absolute difference between the actual and predicted values () **Objective: minimize**
* **RMSLE**:Root mean squared logarithmic error. Similiar to RMSE but it performs a log() on the actual and predicted values prior to computing the difference ($ RMSLE = \sqrt{\_{i=1}^n (log(y\_i+1)-log())$ ) When your response variable has a wide range of values, large response 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.

Most models we assess in future tutorials will report most, if not all, of these metrics. We will often emphasize 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).

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 clasifier classify actual events? This metric is concerned with maximizing the true positives to false negative ratio. In other words, for the events that occured, how many did we predict? Example: . **Objective: maximize**
* **Specificity**: 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**
* **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

### Linear regression

Linear regression is a very simple approach for supervised learning.In particular, linear regression is a useful tool for predicting a quantitative response. Linear regression has been around for a long time and is the topic of innumerable textbooks. Though it may seem somewhat dull compared to some of the more modern statistical learning approaches described in later tutorials, linear regression is still a useful and widely used statistical learning method. Moreover, it serves as a good jumping-off point for newer approaches: as we will see in later tutorials, many fancy statistical learning approaches can be seen as generalizations or extensions of linear regression. Consequently, the importance of having a good understanding of linear regression before studying more complex learning methods cannot be overstated.

#### tl;dr

This tutorial serves asa an introduction to linear regression 1. [Replication requirements](#RR): What you will need to reproduce 2. Preparing our data: Prepare our data for modeling 3. Simple linear regression: Predicting a quantitative response with a single predictor variable 4. Multiple linear regression: Predicting a quantitative response with multiple predictor variables 5. Incorporating interactions: Removing the additive assumption 6. Additional considerations: A few other considerations to know about

##### Replication requirements

This tutorial primarily leverages this [advertising data](http://www-bcf.usc.edu/~gareth/ISL/Advertising.csv) provided by the authors of [an Introduction to Statistical Learning](http://www-bcf.usc.edu/~gareth/ISL/index.html). This is a simple data set that contains, in thousands of dolloars, TV, Radio, and Newspaper budgets for 200 different markets along with the Sales, in thousands of