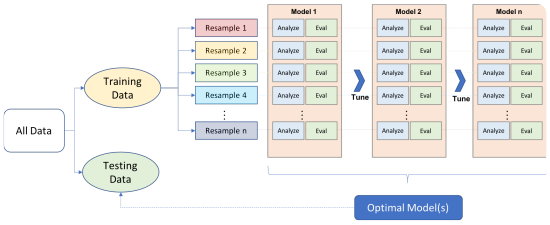
↩

Preparing for Regression Problems

When the objective of a supervised model is to predict a continuous numeric output, we refer to this as a ***regression model***. Applying statistical and machine learning algorithms for regression problems 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 this process 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. This tutorial will prepare you with the fundamentals needed prior to applying regression machine learning algorithms.



tl;dr

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

1. Prerequisites: What you'll need to reproduce the analysis in this tutorial.
2. Data splitting: The need for training and test sets.
3. Feature engineering: A few variable pre-processing steps you'll see throughout this guide.
4. Basic model formulation: Understand the different ways to specify models.
5. Model tuning: Knobs to twiddle.
6. Model evaluation: Gauging model performance for regression problems.

Prerequisites

This tutorial leverages the following packages.

```
library(rsample)
library(caret)
library(h2o)

# turn off progress bars
h2o.no_progress()

# launch h2o
h2o.init()
##  Connection successful!
##
## R is connected to the H2O cluster:
##     H2O cluster uptime:         1 hours 9 minutes
##     H2O cluster timezone:       America/New_York
##     H2O data parsing timezone:  UTC
##     H2O cluster version:        3.18.0.4
##     H2O cluster version age:    2 months
##     H2O cluster name:           H2O_started_from_
##     H2O cluster total nodes:    1
##     H2O cluster total memory:   1.61 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:         XGBoost, Algos, A
##     R Version:                  R version 3.4.4 (
```

To illustrate some of the concepts we will use the Ames Housing data that has been included in the `AmesHousing package` . This 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. Throughout this tutorial, we'll demonstrate approaches with the regular `df` data frame. However, since many of the supervised regression tutorials that we provide leverage `h2o` , we'll also show how to do some of the things with `h2o` . This requires your data to be in an H2O object, which you can convert any data frame easily with `as.h2o` .

```
# import data
df <- AmesHousing::make_ames()

# convert to h2o object
```

```
df.h2o <- as.h2o(df)

# dimensions
dim(df)
## [1] 2930   81
```

Data splitting

Spending our data wisely

A major goal of the machine learning process is to find an algorithm $f(x)$ that most accurately predicts future values (y) based on a set of inputs (x). 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!*

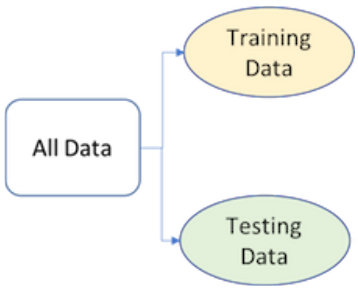


Fig 1: Splitting data into training and test sets.

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 ($> 80\%$) 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 ($> 40\%$) won't allow us to get a good assessment of model parameters

Typically, we are not lacking in the size of our data here, 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 in the quantiles in your response variable (y). 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
set.seed(123)
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.
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
train_4 <- split_2[[1]]
test_4  <- split_2[[2]]
```

Since this sampling approach will randomly sample across the distribution of y (`Sale_Price` in our example), you will typically result in a similar distribution between your training and test sets as illustrated below.

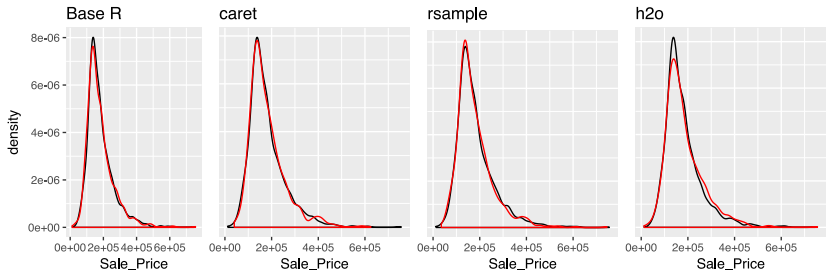


Fig 2: Training (black) vs. test (red) distribution.

Stratified sampling

However, if we want to explicitly control our sampling so that our training and test sets have similar y distributions, we can use stratified sampling. This is more common with classification problems but we also apply within regression problems as our data sets have less observations and strongly deviate 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 continuous response variable is to use the `rsample` package:

```
# stratified sampling with the rsample package
set.seed(123)
split_strat  <- initial_split(df, prop = 0.7, strata
train_strat  <- training(split_strat)
test_strat   <- testing(split_strat)
```

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 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.

One-hot encoding

Many models require all 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:

```
##   id x
## 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:

```
##   id x.a x.b x.c
## 1  1   1   0   0
## 2  2   0   0   1
## 3  3   0   1   0
## 4  4   0   0   1
## 5  5   0   0   1
## 6  6   1   0   0
```

```
## 7  7   0   1   0
## 8  8   0   0   1
```

This is called *less than full rank* (aka one-to-one) one-hot 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  1   0   0
## 2  2   0   1
## 3  3   1   0
## 4  4   0   1
## 5  5   0   1
## 6  6   0   0
## 7  7   1   0
## 8  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 general
# neural networks
full_rank  <- dummyVars( ~ ., data = df, fullRank = '
train_oh   <- predict(full_rank, train_1)
test_oh    <- predict(full_rank, test_1)

# less than full rank --> dummy encoding
dummy    <- dummyVars( ~ ., data = df, fullRank = FA
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 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 explore ordinal encoding of your data.

- if using `h2o` you do not need to explicity encode your categorical variables but you can override 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.

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 = TRUE, col = "re
```

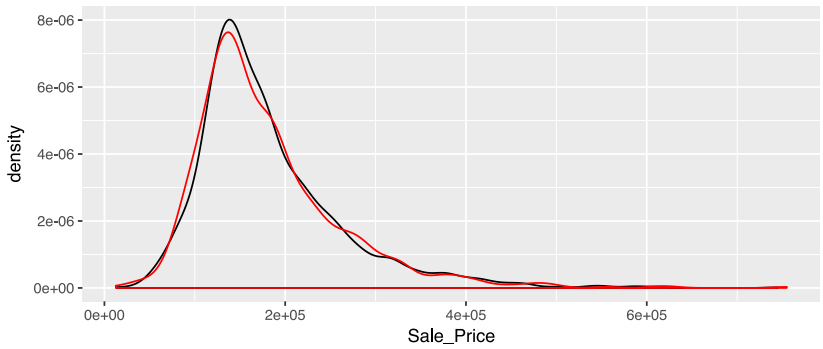


Fig 3: Right skewed response variable.

To normalize, we have two options:

Option 1: normalize with a log 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)
```

Option 2: use a Box Cox transformation. A Box Cox transformation is more flexible and will find the transformation from a family of power transforms that will transform the variabe as close as possible to a normal distribution. **Important note**: 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$Sale_Pric
train_bc_y <- forecast::BoxCox(train_1$Sale_Price, l
test_bc_y  <- forecast::BoxCox(test_1$Sale_Price, la
```

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

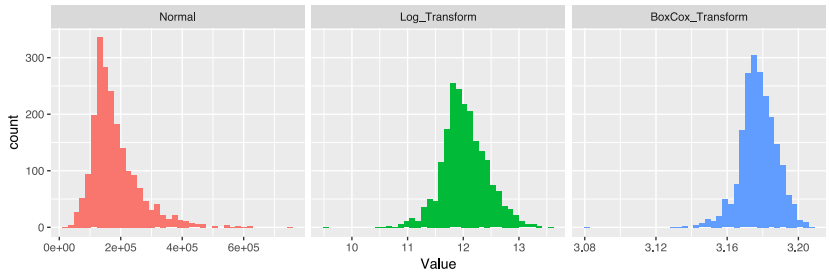


Fig 4: Response variable transformations.

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)

# Inverse Box Cox function
inv_box_cox <- function(x, lambda) {
```

```
  if (lambda == 0) exp(x) else (lambda*x + 1)^(1/lam
}

# re-transforming the Box Cox-transformed value
inv_box_cox(y, lambda)
## [1] 10
```

Predictor Transformation

Some models (*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 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 predictor variables. Note, it is important that 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 <- 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.

```
# 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", "nz
  )

# 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 forests). There are pros and cons to each package; some may be more computationally efficient while others 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 a* 🐱.

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, fami
lm.caret <- train(Sale_Price ~ ., data = train_1, me
```

One thing you will notice throughout future tutorials 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'll use `glmnet` which requires our features (`x`) and response (`y`) variable 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
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 = tra
```

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.

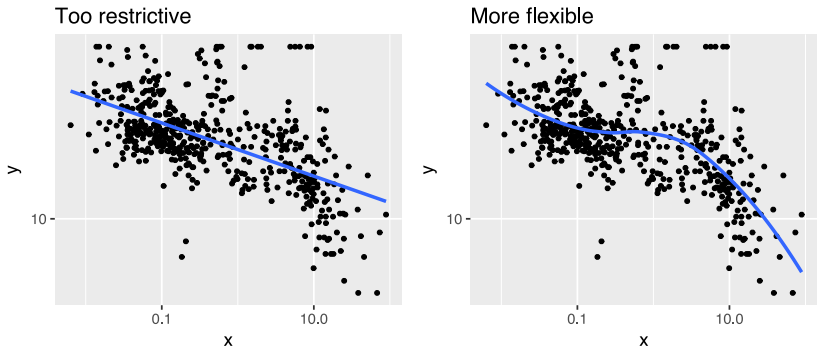


Fig 5: Tuning allows for more flexible patterns to be fit.

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.

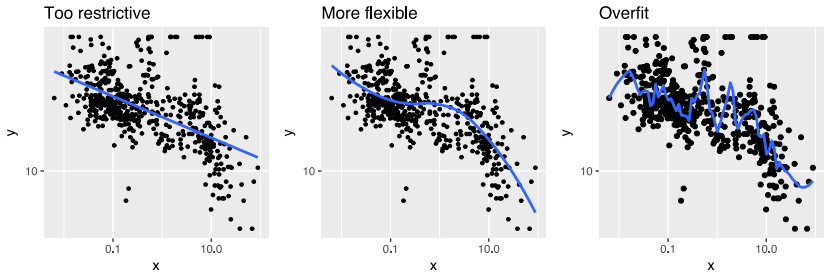


Fig 6: Highly tunable models can overfit if we are not careful.

Throughout future tutorials 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 held-out data.

Let's go back to this image...

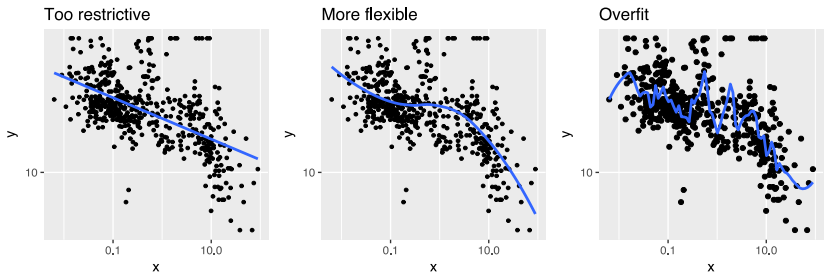


Fig 7: Bias versus variance.

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 model 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.

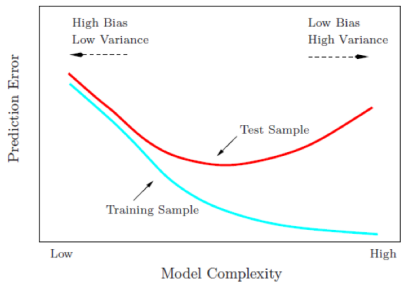


Fig 8: Bias-variance tradeoff.

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 equal size. The model is fit on $k-1$ folds and then the held-out validation fold is used to compute the error. This procedure 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 ($\epsilon_1, \epsilon_2, \ldots, \epsilon_k$). 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.

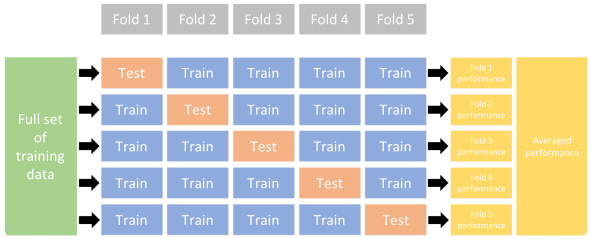


Fig 9: Illustration of the k-fold cross validation process.

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.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 regression model. The most common include:

- **MSE**: Mean squared error is the average of the squared error ($MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$). 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 ($RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$) 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 ($MAE = \frac{1}{n}\sum_{i=1}^{n}(|y_i - \hat{y}_i|)$). **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 ($RMSLE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(log(y_i + 1) - log(\hat{y}_i + 1))^2}$). 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**

- R^2: 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 R^2 than the other. You should not place

too much emphasis on this metric. **Objective: maximize**

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