UC business Analytics R programming Guide 1(2)

Koji Mizumura

2019-09-06

Table of Contents

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

# Predictive analytics

## Machine Learning

### Regularized regression

As discussed, linear regression is a simple and fundamental approach for supervised learning. Moreover, when the assumption required by OLS are met, the coefficients produced by OLS are unbiased and, of all unbiased linear techniques, have the lowest variance. However, in today’s world, data sets being analyzed typically have a large amount of features. As the number of features grow, our OLS assumptions typically break down 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 regression coefficients, which can reduce the variance and decrease our of sample error.

#### tl;dr

This tutorial serves as an introduction to regularized and covers:

1. [Replication requirements](#RR_RR): What you’ll need to reproduce the analysis in this tutorial.
2. [Why regularize](#RR_Reg): A closer look at why regularization can improve upon ordinary least squares regression.
3. [Ridge regression](#RR_Ridge): Regularizing coefficients but keeping all features.
4. [Lasso regression](#RR_LASSO): Regularizing coefficients to perform feature selection.
5. [Elastic nets](#RR_EN): Combining Ridge and Lasso regularization. Predicting: Once you’ve found your optimal model, predict on a new data set.
6. [Other package implementations](#RR_pck_imp): Implementing regularization with other popular packages.
7. [Learning more](#RR_Learn): Where to go from here.

#### Replication Requirements:

This tutorial leverages the following packags. Most of these packages are playing a supporting role while the main emphasis will be on the glmnet package.

library(tidyverse)  
library(rsample) # data splitting   
library(glmnet) # implementing regularized regression approaches  
library(dplyr) # basic data manipulation procedures  
library(ggplot2) # plotting

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

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

#### Why regularize

The objective of ordinary least squares regression is to find the plane that minimizes the sum of squared errors (SSE) between the observed and predicted response. In Figure 1, this means identifying the plane that minimizes the grey lines, which measure the distance between the observed (red dots) and predicted response (blue plane).

More formally, this objective function is written as:

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

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 outcome. 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. Multicollinearty**

As *p* increases we are more likely to capture multiple features that have some multicollinearity. When multicollinearity exists, we often see high variability in our coefficient terms. For example, in our Ames data, Gr\_Liv\_Area and TotRms\_AbvGrd are two variables that have a correlation of 0.801 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.

# fit with two strongly correlated variables  
lm(Sale\_Price ~ Gr\_Liv\_Area + TotRms\_AbvGrd, data = ames\_train)  
##   
## Call:  
## lm(formula = Sale\_Price ~ Gr\_Liv\_Area + TotRms\_AbvGrd, data = ames\_train)  
##   
## Coefficients:  
## (Intercept) Gr\_Liv\_Area TotRms\_AbvGrd   
## 38807.0 145.9 -11844.3  
##   
## Call:  
## lm(formula = Sale\_Price ~ Gr\_Liv\_Area + TotRms\_AbvGrd, data = ames\_train)  
##   
## Coefficients:  
## (Intercept) Gr\_Liv\_Area TotRms\_AbvGrd   
## 49953.6 137.3 -11788.2

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  
lm(Sale\_Price ~ Gr\_Liv\_Area, data = ames\_train)  
##   
## Call:  
## lm(formula = Sale\_Price ~ Gr\_Liv\_Area, data = ames\_train)  
##   
## Coefficients:  
## (Intercept) Gr\_Liv\_Area   
## 7989.4 115.6  
##   
## Call:  
## lm(formula = Sale\_Price ~ Gr\_Liv\_Area, data = ames\_train)  
##   
## Coefficients:  
## (Intercept) Gr\_Liv\_Area   
## 17797 108  
  
# fit with just TotRms\_Area  
lm(Sale\_Price ~ TotRms\_AbvGrd, data = ames\_train)  
##   
## Call:  
## lm(formula = Sale\_Price ~ TotRms\_AbvGrd, data = ames\_train)  
##   
## Coefficients:  
## (Intercept) TotRms\_AbvGrd   
## 15177 25788  
##   
## Call:  
## lm(formula = Sale\_Price ~ TotRms\_AbvGrd, data = ames\_train)  
##   
## Coefficients:  
## (Intercept) TotRms\_AbvGrd   
## 26820 23731

This is a common result when collinearity exists. Coefficients for correlated features become over-inflated and can fluctuate significantly. 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 (Myers, 1994) 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), 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 prefer techniques that provide feature selection. One approach to this is called *hard threshholding* feature selection, which can be performed with linear 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.

**Regulaized regression**

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 regulaized regression model is very similar to OLS regression; however, we add a penalty paramter (P)

There are two main penalty parameters, which we will see shortly, but they both have a similarr effect. They constrain 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). Next, we’ll explore the most common approaches to incorporate regularization.

#### Ridge

Ridge regression [(Hoerl, 1970](https://www.tandfonline.com/doi/abs/10.1080/00401706.1970.10488634)) controls the coefficients by adding to the objective function. This penalty parameter is also referred to as ???" as it signifies a second-order penlty being used on the coefficients.

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 PLS regression objective function of simply minimizing SSE.

However, as $\lambda -> \infit$, the penalty becomes large and forces our coefficients to zero. This is illustrated in Figure 2 where exemplar coefficients have been regularized with $ranging from 0 to over 8,000 (log(8103)=9).

Aothough these coefficients were scaled and centered prior to the analysis, you will notice that some are extremely large when . Furthermore, you will notice the large shrinks to zero. his is indicitive of multicollinearity and likely illustrates that constraining our coefficients with whre its then continuously shrinks to zero. This is indicative of multicollinearity and likely illustrates that constraining our coefficients with may reduce the variance, and therefore the error in our model. However, the question remains - how do we find the amount that minimizes our error? We will answer this shortly.

##### Implementation

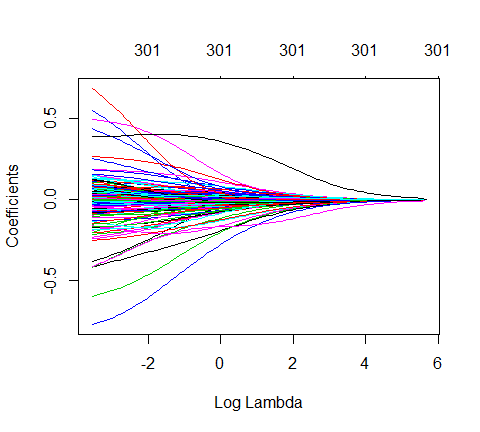
To implement Ridge regression, we will focus on the glmnet package (implementation in other packages are illustrated [below](http://uc-r.github.io/regularized_regression#other)). glmnet does not use the formula method (y ~ x) so prior to modeling we need to create our feature and target set. Furthermore, we use the model.matrix function on our feature set, which will automatically dummy encode qualitative variables (see Matrix::sparse.model.matrix for increased efficiency on large dimension data). We also log transform our response variable due to its skeweness.

# Create training and testing feature model matrices and response vectors.  
# we use model.matrix(...)[, -1] to discard the intercept  
ames\_train\_x <- model.matrix(Sale\_Price ~ ., ames\_train)[, -1]  
ames\_train\_y <- log(ames\_train$Sale\_Price)  
  
ames\_test\_x <- model.matrix(Sale\_Price ~ ., ames\_test)[, -1]  
ames\_test\_y <- log(ames\_test$Sale\_Price)  
  
# What is the dimension of of your feature matrix?  
dim(ames\_train\_x)  
## [1] 2054 307  
## [1] 2054 307

To apply a ridge 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 awre 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 ames data  
library(glmnet)  
ames\_ridge <- glmnet(  
 x = ames\_train\_x,  
 y = ames\_train\_y,  
 alpha = 0  
)  
  
plot(ames\_ridge, xvar="lambda")



In fact, we can see the exact values applied with ames\_ridge$lambda. Although you can specify your own values, by default glmnet applies 100 values that are derived. 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 thebcoefficients for the Gr\_Liv\_Area and TotRms\_abvGrd features for the largest (279.1035) and smallest (0.02791035). You can see how the largest value has pushed these coefficients to nearly 0.

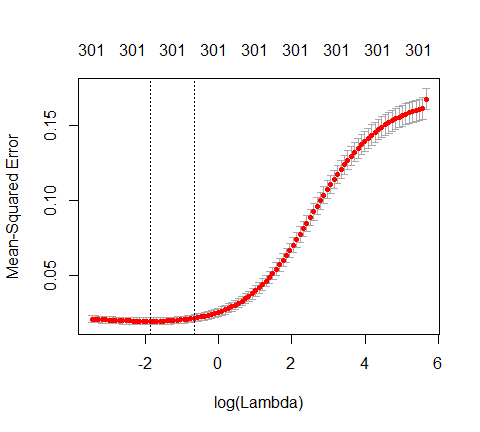
# lambdas applied to penalty parameter  
ames\_ridge$lambda %>% head()  
## [1] 289.0010 263.3270 239.9337 218.6187 199.1972 181.5011  
## [1] 279.1035 254.3087 231.7166 211.1316 192.3752 175.2851  
  
# coefficients for the largest and smallest lambda parameters  
coef(ames\_ridge)[c("Gr\_Liv\_Area", "TotRms\_AbvGrd"), 100]  
## Gr\_Liv\_Area TotRms\_AbvGrd   
## 0.0001108687 0.0083032186  
## Gr\_Liv\_Area TotRms\_AbvGrd   
## 0.0001004011 0.0096383231  
coef(ames\_ridge)[c("Gr\_Liv\_Area", "TotRms\_AbvGrd"), 1]   
## Gr\_Liv\_Area TotRms\_AbvGrd   
## 5.848028e-40 1.341550e-37  
## Gr\_Liv\_Area TotRms\_AbvGrd   
## 5.551202e-40 1.236184e-37

However, at this point, we do not understand how much improvement we are experiencing in our model.

**Tuning**

Recall tht is a tuning parameter that helps to control our model frok overfitting to the training data. However, to identify the optimal value we need to perform [cross validation](http://uc-r.github.io/resampling_methods) (CV). cv.glmnet provides a built-in option to perform k-fold CV, and by default, performs 10-fold CV.

# Apply CV Ridge regression to ames data  
ames\_ridge <- cv.glmnet(  
 x = ames\_train\_x,  
 y = ames\_train\_y,  
 alpha = 0  
)  
  
# plot results  
plot(ames\_ridge)



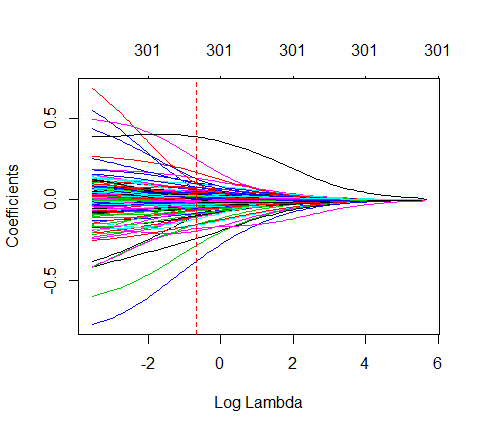
Our plot outputs above illustrates the 10-fold CV mean squared error(MSE) across the values. It illustrates that we do not see substantial improvement; however, as we constrain our coefficients with penalty, the MSE rises considerably.

The numbers at the top of the plot (299) just refer to the number of variables in the model. Ridge regression does not force any variable exactly zero so all features will remain in the model (we will see this change with lasso and elastic nets).

ames\_ridge$cvm %>% min() #minimum MSE  
## [1] 0.01955871  
ames\_ridge$lambda.min # lambda for this min MSE  
## [1] 0.1542312  
  
ames\_ridge$cvm[ames\_ridge$lambda == ames\_ridge$lambda.1se] # 1st st.error  
## [1] 0.02160821  
ames\_ridge$lambda.1se # lambda for this MSE  
## [1] 0.5169216

The advantage of identifying the with an MSE within one standard error becomes more obvious with the lasso and elastic net models. However, for now we can assess this visually. Here we plot the coefficients across the values and the dashed red line represents the largest that falls within oe standard error of the minimum MSE. This shows you how much we can constrain the coefficients while still maximizing predictive accuracy.

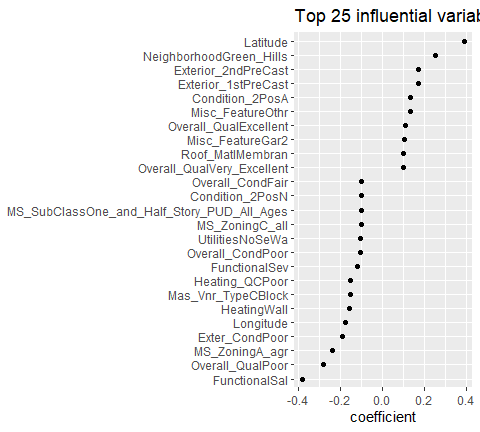
ames\_ridge\_min <- glmnet(  
 x = ames\_train\_x,  
 y = ames\_train\_y,  
 alpha = 0  
)  
  
plot(ames\_ridge\_min, xvar = "lambda")  
abline(v = log(ames\_ridge$lambda.1se), col = "red", lty = "dashed")



**Advantage & Disdvantage**

In essence, the ridge regression model has pushed 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 have been pushed closer to zero. This means we have reduced the noise in our data, which provides us more clarity in identifying the true signals in our model.

coef(ames\_ridge, s = "lambda.1se") %>%   
 broom::tidy() %>%   
 filter(row != "(Intercept)") %>%   
 top\_n(25, wt = abs(value)) %>%   
 ggplot(aes(value, reorder(row, value)))+  
 geom\_point()+  
 ggtitle("Top 25 influential variables")+  
 xlab("coefficient")+  
 ylab(NULL)



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 model may be preferable.

#### LASSO

The *least absolute shrinkage and selection operator (lasso)* model ([Tibshirani, 1996](http://www.jstor.org/stable/2346178?seq=1#page_scan_tab_contents)) is an alternative to ridge regression 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

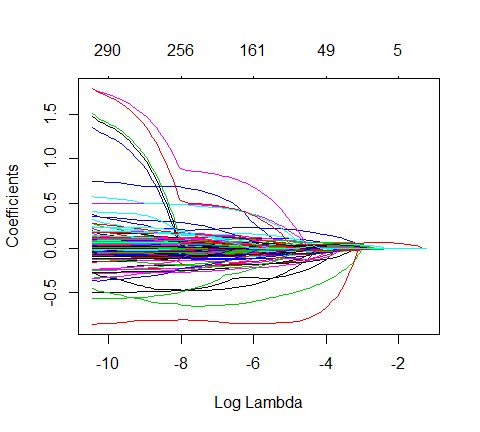
Whereas the ridge regression approach pushes variables to *approximately but not equal to zero*, the lasso penalty will actually push coefficients to zero as illustrated with Fig. 3. Thus the lasso model not only improves the model with regularization but it also conducts automated feature selection.

In Fig.3we see that 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.

**Implementation**

Implementing lasso follows the same logic as implementing the ridge model, we just need to switch alpha=1 within glmnet.

## apply lasso regression to ames data  
  
ames\_lasso <- glmnet(  
 x = ames\_train\_x,  
 y = ames\_train\_y,  
 alpha = 1  
)  
  
plot(ames\_lasso, xvar = "lambda")



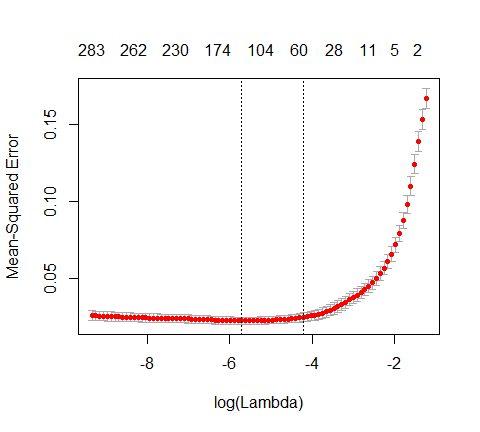
Our output illustrates a quick drop in the number of features retained in the lasso model as -> -6. In fact, we see several features that had very large coefficients for the OLS model (when . As before, these features are likely highly correlated with other features in the data, causing their coefficients to be excessively large. As we constrain our model, these noisy features are pushed to zero.

However, similar to the Ridge regression section, we need to perform CV to determine when the right value is for .

**Tuning**

To perform CV we use the same approach as we did in the ridge regression tuning section, but change our alpha = 1. We see that we can minimize our MSE by applying approximately Not only does this minimize our MSE but it also reduces the number of features to

# Apply CV Ridge regression to ames data  
ames\_lasso <- cv.glmnet(  
 x = ames\_train\_x,  
 y = ames\_train\_y,  
 alpha = 1  
)  
# plot results  
plot(ames\_lasso)

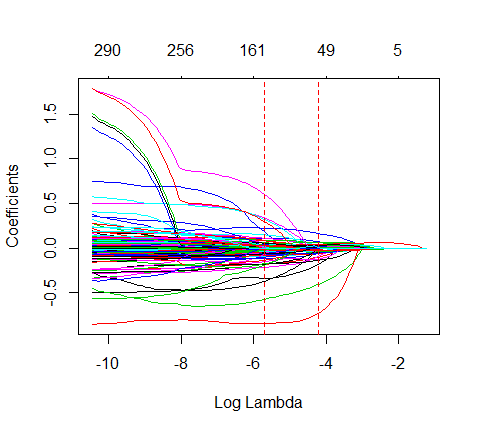


As before, we can extract our minimum and one standard error MSE and $\λ$ values.

min(ames\_lasso$cvm) # minimum MSE  
## [1] 0.02246344  
## [1] 0.02275227  
ames\_lasso$lambda.min # lambda for this min MSE  
## [1] 0.00332281  
## [1] 0.003521887  
  
ames\_lasso$cvm[ames\_lasso$lambda == ames\_lasso$lambda.1se] # 1 st.error of min MSE  
## [1] 0.02482119  
## [1] 0.02562055  
ames\_lasso$lambda.1se # lambda for this MSE  
## [1] 0.01472211  
## [1] 0.01180396

Now the advantage of identifying the with an MSE within one standard error becomes more obvious. If we use the that derives the minimum MSE we can reduce our feature set from 307 down to less than 160. 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 63 features. If describing and interpreting the predictors is an important outcome of your analysis, this may significantly aid your endeavor.

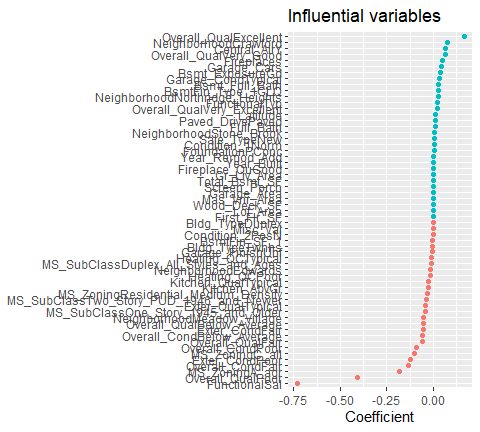
ames\_lasso\_min <- glmnet(  
 x = ames\_train\_x,  
 y = ames\_train\_y,  
 alpha = 1  
)  
  
plot(ames\_lasso\_min, xvar = "lambda")  
abline(v = log(ames\_lasso$lambda.min), col = "red", lty = "dashed")  
abline(v = log(ames\_lasso$lambda.1se), col = "red", lty = "dashed")



**Advantage & Disadvantage**

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 feature most influential to predictive accuracy.

coef(ames\_lasso, s = "lambda.1se") %>%   
 broom::tidy() %>%   
 filter(row != "(Intercept)") %>%   
 ggplot(aes(value, reorder(row, value), color = value >0))+  
 geom\_point(show.legend = FALSE)+  
 ggtitle("Influential variables")+  
 xlab("Coefficient")+  
 ylab(NULL)



However, often when we remove features we sacrifice accuracy. Consequently, to gain the refined clarity and simplicity that lasso provides, we sometimes reduce the level of accuracy. Typically we do not see large differences in the minimum errors between the two. So practically, this may not be significant but if you are purely competing on minimizing error (i.e. Kaggle competitions) this may make all the difference!

# minimum Ridge MSE  
min(ames\_ridge$cvm)  
## [1] 0.01955871  
## [1] 0.02147691  
  
# minimum Lasso MSE  
min(ames\_lasso$cvm)  
## [1] 0.02246344  
## [1] 0.02275227

#### Elastic Nets

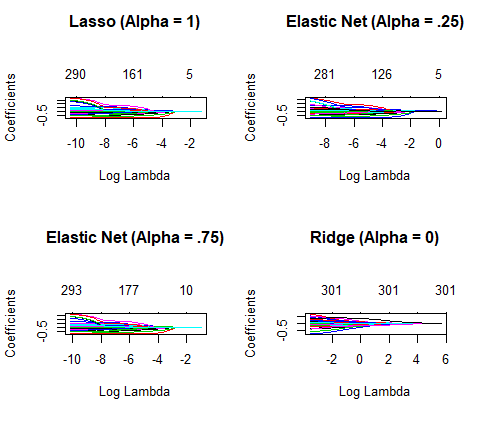
A generalization of the ridge and lasso models is the elastic net (Zou and Hastie, 2005), which combines the two penalties.

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 model is that it enables effective regularization via the ridge penalty with the feature selection characteristics of the lasso penalty.

**Implementation**

We implement an elastic net the same way as the ridge and lasso models, which are controlled by 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(ames\_train\_x, ames\_train\_y, alpha = 1.0)   
elastic1 <- glmnet(ames\_train\_x, ames\_train\_y, alpha = 0.25)   
elastic2 <- glmnet(ames\_train\_x, ames\_train\_y, alpha = 0.75)   
ridge <- glmnet(ames\_train\_x, ames\_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")



**Tuning**

In ridge and lasso models is our primary tuning parameter. However, with elastic nets, we want to tune 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 alphas from 0-1, and empty columns where we will dump our model results into.

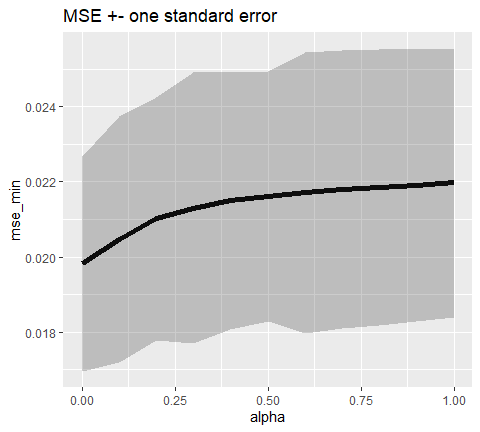
# maintain the same folds across all models  
fold\_id <- sample(1:10, size = length(ames\_train\_y), replace=TRUE)  
  
# search across a range of alphas  
tuning\_grid <- tibble::tibble(  
 alpha = seq(0, 1, by = .1),  
 mse\_min = NA,  
 mse\_1se = NA,  
 lambda\_min = NA,  
 lambda\_1se = NA  
)

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

for(i in seq\_along(tuning\_grid$alpha)) {  
   
 # fit CV model for each alpha value  
 fit <- cv.glmnet(ames\_train\_x, ames\_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 0.0198 0.0227 0.141 0.623   
## 2 0.1 0.0205 0.0237 0.0365 0.134   
## 3 0.2 0.0210 0.0243 0.0182 0.0736  
## 4 0.3 0.0213 0.0249 0.0122 0.0539  
## 5 0.4 0.0215 0.0249 0.00912 0.0404  
## 6 0.5 0.0216 0.0250 0.00729 0.0323  
## 7 0.6 0.0217 0.0255 0.00608 0.0296  
## 8 0.7 0.0218 0.0255 0.00521 0.0253  
## 9 0.8 0.0219 0.0255 0.00456 0.0222  
## 10 0.9 0.0219 0.0255 0.00405 0.0197  
## 11 1 0.0220 0.0256 0.00365 0.0177  
## # A tibble: 11 x 5  
## alpha mse\_min mse\_1se lambda\_min lambda\_1se  
## <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 0 0.0217 0.0241 0.136 0.548   
## 2 0.100 0.0215 0.0239 0.0352 0.0980  
## 3 0.200 0.0217 0.0243 0.0193 0.0538  
## 4 0.300 0.0218 0.0243 0.0129 0.0359  
## 5 0.400 0.0219 0.0244 0.0106 0.0269  
## 6 0.500 0.0220 0.0250 0.00848 0.0236  
## 7 0.600 0.0220 0.0250 0.00707 0.0197  
## 8 0.700 0.0221 0.0250 0.00606 0.0169  
## 9 0.800 0.0221 0.0251 0.00530 0.0148  
## 10 0.900 0.0221 0.0251 0.00471 0.0131  
## 11 1.00 0.0223 0.0254 0.00424 0.0118

If we plot the MSE +- one standard erro 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 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)+  
 ggtitle("MSE +- one standard error")



**Advantage & Disadvantage**

As previously stated, the advantage of the elastic net model is that it enables effective regularization via the ridge penalty with the feature selection characteristics of the lasso penalty. Effectively, elastic nets allow us to control multicollinearity concerns, perform regression when , and reduce excessive noise in our data so that we can isolate the most influential variables while balancing prediction accuracy.

However, elastic nets, and regularization models in general, still assume linear relationships between the features and the target variable. And although we can incorporate non-additive models with interactions, doing this when the number of features is large is extremely tedious and difficult. When non-linear relationships exist, its beneficial to start exploring non-linear regression

#### Predicting

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 0.022. I use the minimum value to predict on the unseen test set and obtain a slightly lower MSE of 0.015.

# some best model  
cv\_lasso <- cv.glmnet(ames\_train\_x, ames\_train\_y, alpha = 1.0)  
min(cv\_lasso$cvm)  
## [1] 0.02036225  
## [1] 0.02279668  
  
# predict  
pred <- predict(cv\_lasso, s = cv\_lasso$lambda.min, ames\_test\_x)  
mean((ames\_test\_y - pred)^2)  
## [1] 0.02040651  
## [1] 0.01488605

#### Other package implementation {##RR\_pck\_imp}

glmnet is not the only package that can perform regularized regression. The following also shows how to implement with the popular caret and h2o packages. For brevity, I show the code but not the output.

## caret package  
library(caret)  
  
train\_control <- trainControl(method = "cv", number = 10)  
  
caret\_mod <- train(  
 x = ames\_train\_x,  
 y = ames\_train\_y,  
 method = "glmnet",  
 preProc = c("center", "scale", "zv", "nzv"),  
 trControl = train\_control,  
 tuneLength = 10  
)  
  
caret\_mod  
## glmnet   
##   
## 2054 samples  
## 307 predictor  
##   
## Pre-processing: centered (113), scaled (113), remove (194)   
## Resampling: Cross-Validated (10 fold)   
## Summary of sample sizes: 1849, 1848, 1847, 1850, 1849, 1849, ...   
## Resampling results across tuning parameters:  
##   
## alpha lambda RMSE Rsquared MAE   
## 0.1 0.0001335259 0.1529759 0.8626587 0.09899496  
## 0.1 0.0003084622 0.1529697 0.8626702 0.09899085  
## 0.1 0.0007125878 0.1527312 0.8630945 0.09882473  
## 0.1 0.0016461703 0.1523040 0.8638461 0.09858775  
## 0.1 0.0038028670 0.1518043 0.8647353 0.09829651  
## 0.1 0.0087851163 0.1511589 0.8658996 0.09792048  
## 0.1 0.0202947586 0.1512482 0.8658861 0.09846301  
## 0.1 0.0468835259 0.1542133 0.8616162 0.10095388  
## 0.1 0.1083070283 0.1613716 0.8525304 0.10562397  
## 0.1 0.2502032890 0.1783894 0.8378278 0.11798124  
## 0.2 0.0001335259 0.1530144 0.8625906 0.09901019  
## 0.2 0.0003084622 0.1529054 0.8627844 0.09893138  
## 0.2 0.0007125878 0.1526093 0.8633042 0.09873037  
## 0.2 0.0016461703 0.1520597 0.8642754 0.09843475  
## 0.2 0.0038028670 0.1515362 0.8652246 0.09801179  
## 0.2 0.0087851163 0.1511130 0.8660407 0.09800962  
## 0.2 0.0202947586 0.1531822 0.8629272 0.10015289  
## 0.2 0.0468835259 0.1583264 0.8557192 0.10357302  
## 0.2 0.1083070283 0.1713989 0.8406963 0.11319420  
## 0.2 0.2502032890 0.2015887 0.8196545 0.13607965  
## 0.3 0.0001335259 0.1530175 0.8625852 0.09901246  
## 0.3 0.0003084622 0.1528417 0.8628943 0.09887464  
## 0.3 0.0007125878 0.1524303 0.8636143 0.09862411  
## 0.3 0.0016461703 0.1519261 0.8645281 0.09829674  
## 0.3 0.0038028670 0.1513782 0.8655399 0.09792851  
## 0.3 0.0087851163 0.1517901 0.8649707 0.09874489  
## 0.3 0.0202947586 0.1548532 0.8604377 0.10124984  
## 0.3 0.0468835259 0.1631931 0.8487612 0.10712019  
## 0.3 0.1083070283 0.1804627 0.8317568 0.12000176  
## 0.3 0.2502032890 0.2242488 0.8048855 0.15383093  
## 0.4 0.0001335259 0.1530032 0.8626107 0.09900104  
## 0.4 0.0003084622 0.1527835 0.8629949 0.09883071  
## 0.4 0.0007125878 0.1522777 0.8638796 0.09853749  
## 0.4 0.0016461703 0.1517960 0.8647653 0.09816823  
## 0.4 0.0038028670 0.1512813 0.8657246 0.09790412  
## 0.4 0.0087851163 0.1527729 0.8634153 0.09973034  
## 0.4 0.0202947586 0.1567219 0.8575524 0.10244652  
## 0.4 0.0468835259 0.1674084 0.8431901 0.11030906  
## 0.4 0.1083070283 0.1900641 0.8220999 0.12713979  
## 0.4 0.2502032890 0.2456477 0.7968665 0.17219318  
## 0.5 0.0001335259 0.1529694 0.8626695 0.09897170  
## 0.5 0.0003084622 0.1527206 0.8631031 0.09877948  
## 0.5 0.0007125878 0.1521743 0.8640690 0.09847427  
## 0.5 0.0016461703 0.1517015 0.8649410 0.09808645  
## 0.5 0.0038028670 0.1514410 0.8654740 0.09816784  
## 0.5 0.0087851163 0.1535069 0.8622717 0.10039370  
## 0.5 0.0202947586 0.1585892 0.8546972 0.10366586  
## 0.5 0.0468835259 0.1711919 0.8386196 0.11310210  
## 0.5 0.1083070283 0.1988715 0.8144491 0.13394394  
## 0.5 0.2502032890 0.2677720 0.7874492 0.19208881  
## 0.6 0.0001335259 0.1529457 0.8627110 0.09894655  
## 0.6 0.0003084622 0.1526476 0.8632329 0.09872871  
## 0.6 0.0007125878 0.1521007 0.8642136 0.09841560  
## 0.6 0.0016461703 0.1516403 0.8650668 0.09805262  
## 0.6 0.0038028670 0.1516698 0.8651021 0.09848230  
## 0.6 0.0087851163 0.1541771 0.8612517 0.10084110  
## 0.6 0.0202947586 0.1606510 0.8515691 0.10511838  
## 0.6 0.0468835259 0.1751425 0.8337697 0.11597593  
## 0.6 0.1083070283 0.2072075 0.8085757 0.14014121  
## 0.6 0.2502032890 0.2895134 0.7807597 0.21111363  
## 0.7 0.0001335259 0.1529182 0.8627588 0.09892152  
## 0.7 0.0003084622 0.1525582 0.8633865 0.09868078  
## 0.7 0.0007125878 0.1520463 0.8643179 0.09835732  
## 0.7 0.0016461703 0.1515912 0.8651754 0.09802856  
## 0.7 0.0038028670 0.1519555 0.8646544 0.09881899  
## 0.7 0.0087851163 0.1548004 0.8603025 0.10119692  
## 0.7 0.0202947586 0.1627999 0.8483414 0.10680467  
## 0.7 0.0468835259 0.1792009 0.8286142 0.11899322  
## 0.7 0.1083070283 0.2158269 0.8020943 0.14685245  
## 0.7 0.2502032890 0.3123587 0.7639857 0.23129960  
## 0.8 0.0001335259 0.1528934 0.8628024 0.09889940  
## 0.8 0.0003084622 0.1524796 0.8635221 0.09864034  
## 0.8 0.0007125878 0.1519877 0.8644285 0.09830023  
## 0.8 0.0016461703 0.1515406 0.8652735 0.09802800  
## 0.8 0.0038028670 0.1522942 0.8641256 0.09918411  
## 0.8 0.0087851163 0.1554853 0.8592392 0.10161487  
## 0.8 0.0202947586 0.1649098 0.8451450 0.10842937  
## 0.8 0.0468835259 0.1832344 0.8233507 0.12199727  
## 0.8 0.1083070283 0.2242570 0.7962765 0.15379654  
## 0.8 0.2502032890 0.3360986 0.7248864 0.25185135  
## 0.9 0.0001335259 0.1528683 0.8628464 0.09887948  
## 0.9 0.0003084622 0.1524011 0.8636585 0.09859931  
## 0.9 0.0007125878 0.1519283 0.8645367 0.09825245  
## 0.9 0.0016461703 0.1514541 0.8654270 0.09800839  
## 0.9 0.0038028670 0.1526743 0.8635214 0.09958496  
## 0.9 0.0087851163 0.1562250 0.8580869 0.10208126  
## 0.9 0.0202947586 0.1667279 0.8425633 0.10988380  
## 0.9 0.0468835259 0.1869510 0.8187765 0.12486550  
## 0.9 0.1083070283 0.2328192 0.7897299 0.16118585  
## 0.9 0.2502032890 0.3598416 0.6471565 0.27230177  
## 1.0 0.0001335259 0.1528421 0.8628890 0.09885896  
## 1.0 0.0003084622 0.1523370 0.8637698 0.09856552  
## 1.0 0.0007125878 0.1518838 0.8646164 0.09822006  
## 1.0 0.0016461703 0.1514135 0.8655058 0.09802221  
## 1.0 0.0038028670 0.1530411 0.8629353 0.09994305  
## 1.0 0.0087851163 0.1570340 0.8568153 0.10261590  
## 1.0 0.0202947586 0.1683894 0.8402729 0.11115096  
## 1.0 0.0468835259 0.1904441 0.8148559 0.12743546  
## 1.0 0.1083070283 0.2409139 0.7855310 0.16832017  
## 1.0 0.2502032890 0.3805446 0.5646772 0.29026265  
##   
## RMSE was used to select the optimal model using the smallest value.  
## The final values used for the model were alpha = 0.2 and lambda  
## = 0.008785116.

## h2o package  
library(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\RtmpmC07pv/h2o\_kojikm\_mizumura\_started\_from\_r.out  
## C:\Users\KOJIKM~1.MIZ\AppData\Local\Temp\RtmpmC07pv/h2o\_kojikm\_mizumura\_started\_from\_r.err  
##   
##   
## Starting H2O JVM and connecting: . Connection successful!  
##   
## R is connected to the H2O cluster:   
## H2O cluster uptime: 5 seconds 819 milliseconds   
## H2O cluster timezone: Asia/Tokyo   
## H2O data parsing timezone: UTC   
## H2O cluster version: 3.22.1.1   
## H2O cluster version age: 8 months and 9 days !!!   
## H2O cluster name: H2O\_started\_from\_R\_kojikm.mizumura\_qba461   
## 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 3.6.1 (2019-07-05)  
  
# convert data to h2o object  
ames\_h2o <- ames\_train %>%  
 mutate(Sale\_Price\_log = log(Sale\_Price)) %>%  
 as.h2o()  
##   
 |   
 | | 0%  
 |   
 |=================================================================| 100%  
  
# set the response column to Sale\_Price\_log  
response <- "Sale\_Price\_log"  
  
# set the predictor names  
predictors <- setdiff(colnames(ames\_train), "Sale\_Price")  
  
  
# try using the `alpha` parameter:  
# train your model, where you specify alpha  
ames\_glm <- h2o.glm(  
 x = predictors,   
 y = response,   
 training\_frame = ames\_h2o,  
 nfolds = 10,  
 keep\_cross\_validation\_predictions = TRUE,  
 alpha = .25  
 )  
##   
 |   
 | | 0%  
 |   
 |=========================================================== | 91%  
 |   
 |=================================================================| 100%  
  
# print the mse for the validation data  
print(h2o.mse(ames\_glm, xval = TRUE))  
## [1] 0.03605061  
  
# grid over `alpha`  
# select the values for `alpha` to grid over  
hyper\_params <- list(  
 alpha = seq(0, 1, by = .1),  
 lambda = seq(0.0001, 10, length.out = 10)  
 )  
  
# this example uses cartesian grid search because the search space is small  
# and we want to see the performance of all models. For a larger search space use  
# random grid search instead: {'strategy': "RandomDiscrete"}  
  
# build grid search with previously selected hyperparameters  
grid <- h2o.grid(  
 x = predictors,   
 y = response,   
 training\_frame = ames\_h2o,   
 nfolds = 10,  
 keep\_cross\_validation\_predictions = TRUE,  
 algorithm = "glm",  
 grid\_id = "ames\_grid",   
 hyper\_params = hyper\_params,  
 search\_criteria = list(strategy = "Cartesian")  
 )  
##   
 |   
 | | 0%  
 |   
 | | 1%  
 |   
 |= | 1%  
 |   
 |= | 2%  
 |   
 |== | 3%  
 |   
 |== | 4%  
 |   
 |=== | 5%  
 |   
 |==== | 6%  
 |   
 |==== | 7%  
 |   
 |===== | 7%  
 |   
 |===== | 8%  
 |   
 |====== | 8%  
 |   
 |====== | 9%  
 |   
 |====== | 10%  
 |   
 |======= | 10%  
 |   
 |======== | 13%  
 |   
 |========= | 13%  
 |   
 |========= | 14%  
 |   
 |========== | 16%  
 |   
 |============ | 18%  
 |   
 |============ | 19%  
 |   
 |============= | 21%  
 |   
 |============== | 22%  
 |   
 |================= | 26%  
 |   
 |==================== | 31%  
 |   
 |======================= | 35%  
 |   
 |========================= | 38%  
 |   
 |============================ | 42%  
 |   
 |============================== | 47%  
 |   
 |================================= | 51%  
 |   
 |=================================== | 53%  
 |   
 |===================================== | 57%  
 |   
 |======================================== | 61%  
 |   
 |========================================== | 64%  
 |   
 |=========================================== | 66%  
 |   
 |============================================= | 69%  
 |   
 |=============================================== | 73%  
 |   
 |================================================== | 77%  
 |   
 |===================================================== | 81%  
 |   
 |====================================================== | 84%  
 |   
 |========================================================= | 88%  
 |   
 |=========================================================== | 92%  
 |   
 |============================================================ | 93%  
 |   
 |=============================================================== | 96%  
 |   
 |=================================================================| 100%  
  
# Sort the grid models by mse  
sorted\_grid <- h2o.getGrid("ames\_grid", sort\_by = "mse", decreasing = FALSE)  
sorted\_grid  
## H2O Grid Details  
## ================  
##   
## Grid ID: ames\_grid   
## Used hyper parameters:   
## - alpha   
## - lambda   
## Number of models: 110   
## Number of failed models: 0   
##   
## Hyper-Parameter Search Summary: ordered by increasing mse  
## alpha lambda model\_ids mse  
## 1 [1.0] [1.0E-4] ames\_grid\_model\_11 0.01879637819918772  
## 2 [0.6] [1.0E-4] ames\_grid\_model\_7 0.019149259948806063  
## 3 [0.2] [1.0E-4] ames\_grid\_model\_3 0.01916913620380096  
## 4 [0.3] [1.0E-4] ames\_grid\_model\_4 0.01930059026345251  
## 5 [0.8] [1.0E-4] ames\_grid\_model\_9 0.019363870416558602  
##   
## ---  
## alpha lambda model\_ids mse  
## 105 [0.7] [4.4445] ames\_grid\_model\_52 0.16776797237612928  
## 106 [0.5] [4.4445] ames\_grid\_model\_50 0.16776812574044836  
## 107 [0.8] [6.6667] ames\_grid\_model\_75 0.16777928688692229  
## 108 [0.9] [1.1112] ames\_grid\_model\_21 0.16778577469946573  
## 109 [0.2] [2.2223] ames\_grid\_model\_25 0.16790752750688803  
## 110 [0.9] [10.0] ames\_grid\_model\_109 0.1679420341143368  
  
# grab top model id  
best\_h2o\_model <- sorted\_grid@model\_ids[[1]]  
best\_model <- h2o.getModel(best\_h2o\_model)

#### 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 generalized linear models (i.e. logistic regression, multinomial, poisson, support vector machines). Moreover, alternative approaches to regularization exist such as Least Angle Regression and The Bayesian Lasso. The following are great resources to learn more (listed in order of complexity):

* Applied Predictive Modeling
* Introduction to Statistical Learning
* The Elements of Statistical Learning
* Statistical Learning with Sparsity

### Multivariate Adaptive regression splines (MARS)

Several previous tutorials (i.e., ) discussed algorithms that are intrinsically linear. Many of these models can be adapted to nonlinear patterns in the data by manually adding model terms (i.e. squared terms, interaction effects); however, to do so you must know the specific nature of the nonlinearity a priori. Alternatively, there are numerous algorithms that are inherently nonlinear. When using these models, the exact form of the nonlinearity does not need to be known explicitly or specified prior to model training. Rather, these algorithms will search for, and discover, nonlinearities in the data that help maximize predictive accuracy.

This tutorial discusses multivariate adaptive regression splines (MARS), an algorithm that essentially creates a piecewise linear model which provides an intuitive stepping block into nonlinearity after grasping the concept of linear regression and other intrinsically linear models.

* [Prerequisites](#MARS_PR)
* [Basic Idea](#MARS_BI)
* [Multivariage regression splines](#MARS)
* [Fitting a basic MARS model](#MARS_Fit)
* [Tuning](#MARS_TUNE)
* [Feature interpretation](#MARS_FI)
* [Final thoughts](#MARS_SUM)
* [Learning more](#MARS_LM)

#### Prerequisites

For this tutorial we will use the following packages:

library(rsample) # data splitting   
library(ggplot2) # plotting  
library(earth) # fit MARS models  
library(caret) # automating the tuning process  
library(vip) # variable importance  
library(pdp) # variable relationships

To illustrate various MARS modeling concepts we will use Ames Housing data, which is available via the AmesHousing package.

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

#### The basic idea

Some previous tutorials have focused on linear models. In those tutorials we illustrated some of the advantages of linear models such as their ease and speed of computation and also the intuitive nature of interpreting their coefficients. However, linear models make a strong assumption about linearity, and this assumption is often a poor one, which can affect predictive accuracy.

We can extend linear models to capture non-linear relationships. Typically, this is done by explicitly including polynomial parameters or step functions. Polynomial regression is a form of regression in which the relationship between the independent variable and the dependent variable y is modeled as an degree polynomial of . For example, Equation 1 represents a polynomial regression function where y is modeled as a function of with degrees. Generally speaking, it is unusual to use d greater than 3 or 4 as the larger d becomes, the easier the function fit becomes overly flexible and oddly shapened…especially near the boundaries of the range of x values

An alternative to polynomial regression. is step function regression. Whereas polynomial functions impose a global non-linear relationship, step functions break the range of into bins, and fit a different constant for each bin. This amounts to converting a continuous variable into an ordered categorical variable such that our linear regression function is converted to Equation 2

where represents value ranging from , represents value ranging from , …, represents values ranging from . Figure 1 illustrate polynomial and step function fits for Sale\_Price asa a function of Year\_Build in our ames data.

# polynomial degree of 1  
p1 <- ames\_train %>%   
 ggplot(aes(Year\_Built, Sale\_Price))+  
 geom\_point()+  
 geom\_smooth(method = "gam", formula = y ~ x)  
  
# polynomial degree of 2  
p2 <- ames\_train %>%   
 ggplot(aes(Year\_Built, Sale\_Price))+  
 geom\_point()+  
 geom\_smooth(method = "gam", formula = y ~ poly(x,2))  
  
# polynomial degree of 3  
p3 <- ames\_train %>%   
 ggplot(aes(Year\_Built, Sale\_Price))+  
 geom\_point()+  
 geom\_smooth(method = "gam", formula = y ~ poly(x,3))  
  
# step function  
  
gridExtra::grid.arrange(p1, p2, p3, ncol=3)

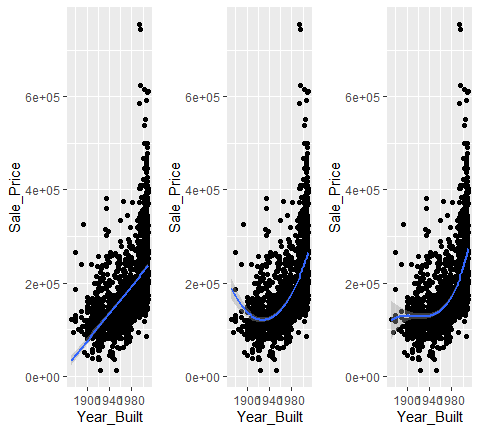


Figure 1: Blue line represents predicted Sale\_Price values as a function of Year\_Built for alternative approaches to modeling explicit nonlinear regression patterns. (A) Traditional nonlinear regression approach does not capture any nonlinearity unless the predictor or response is transformed (i.e. log transformation). (B) Degree-2 polynomial, (C) Degree-3 polynomial, (D) Step function fitting cutting Year\_Built into three categorical levels.

Although useful, the typical implementation of polynomial regression and step functions require the user to explicitly identify and incorporate which variables should have what specific degree of interaction or at points of a variable should cut points to be made for the step functions. Considering many data sets today can easily contain 50, 100, or more features, this would require an enormous and unncessary time commitment from an analyst to determine these explicit non-linear settings.

### Multivariage regression splines

Multivariate adaptive regression splines (MARS) provide a convenient approach to capture the nonlinearity aspect of polynomial regression by assessing cutpoints ( *knots*) similar to step functions. The procedure assesses each data point for each predictor as a knot and creates a linear regression model with the candidate feature(s). For example, consider our simple model of Sale\_Price ~ Year\_Built. The MARS procedure will first look for the single point across the range of Year\_Built values where two different linear relationships between Sale\_Price and Year\_Built achieve the smallest error.

What results is known as hinge function ( where is the cutpoint value). For a single knot, our hinge function is such that our two linear models for Sale\_Price are

Once the first knot has benn found, the search continues for a second knot which is found at 2006 (Figure 2 (B)). This results in three linear models for Sale\_Price:

mars0 <- ames\_train %>%   
 earth::earth(Sale\_Price ~ Year\_Built, data=.)   
  
mars0 %>% print()

This procedure can continue until many knots are found, producing a highly non-linear pattern. Although including many knots may allow us to fit a really good relationship with our training data, it may not generalize very well to new, unseen data. For example, Figure 3 includes nine knots but this likley will not generalize very well to our test data.

Consequently, once the full set of knots have been created, we can sequentially remove knots that do not contribute significantly to predictive accuracy. This process is known as “pruning??? and we can use cross-validation, as we have with the previous models, to find the optimal number of knot

#### Fitting a basic MARS model

We can fit a MARS model with the **earth** package. By default, earth::earth() will assess all potential knots across all supplied features and then will prune to the optimal number of knots on an expected change in (for the training data) of less than 0.001. This calculation is performed by the Generalized cross-validation procedure (GCV statistic), which is a computational shortcut for linear models that produces an error value that approximates leave-one-out cross-validation (see [here](http://w3.atomki.hu/~efo/hornyak/Tikhonov_references/Technometrics_Golub_Heath_Wahba) for technical details).

The following applies a basic MARS model to our ames data and performs a search for required knots across all features. The results show us the final models GCV statistic, generalized (GRSq) and more.

# Fit a basic MARS model  
mars1 <- earth::earth(  
 Sale\_Price ~.,  
 data = ames\_train  
)  
  
# Print model summary  
print(mars1)  
## Selected 36 of 41 terms, and 24 of 307 predictors  
## Termination condition: RSq changed by less than 0.001 at 41 terms  
## Importance: First\_Flr\_SF, Second\_Flr\_SF, Year\_Built, ...  
## Number of terms at each degree of interaction: 1 35 (additive model)  
## GCV 511235214 RSS 978736420657 GRSq 0.9206251 RSq 0.9259456  
## Selected 37 of 45 terms, and 26 of 307 predictors  
## Termination condition: RSq changed by less than 0.001 at 45 terms  
## Importance: Gr\_Liv\_Area, Year\_Built, Total\_Bsmt\_SF, ...  
## Number of terms at each degree of interaction: 1 36 (additive model)  
## GCV 521186626 RSS 995776275391 GRSq 0.9165386 RSq 0.92229

It also shows us that 38 of 41 terms were used from 27 of the 307 original predictors. But what does this mean? If we were to look at all the coefficients, we would see that there are 38 terms in our model (including the intercept). These terms include hinge functions produced from the original 307 predictors (307 predictors because the model automatically dummy encodes our categorical variables). Looking at the first 10 terms in our model, we see that Gr\_Liv\_Area is included with a knot at 2945 (the coefficients for is ), Year\_Built is included with a knot at 2003, etc.

mars1 %>% .$coefficients %>% head()  
## Sale\_Price  
## (Intercept) 289316.21612  
## h(Gr\_Liv\_Area-2790) -55.26061  
## h(Year\_Built-2002) 3040.55619  
## h(2002-Year\_Built) -410.85708  
## h(2220-Total\_Bsmt\_SF) -30.70632  
## h(Bsmt\_Unf\_SF-543) -25.37750  
  
## Sale\_Price  
## (Intercept) 301399.98345  
## h(2945-Gr\_Liv\_Area) -49.84518  
## h(Year\_Built-2003) 2698.39864  
## h(2003-Year\_Built) -357.11319  
## h(Total\_Bsmt\_SF-2171) -265.30709  
## h(2171-Total\_Bsmt\_SF) -29.77024  
## Overall\_QualExcellent 88345.90068  
## Overall\_QualVery\_Excellent 116330.48509  
## Overall\_QualVery\_Good 36646.55568  
## h(Bsmt\_Unf\_SF-278) -21.15661

The plot method for MARS model objects provide convenient performance and residual plots.Figure 4 illustrates the model selection plot that graphs the GCV (left-hand y-axis and solid black line) based on the number of terms retained in the model (x-axis) which are constructed from a certain number of original predictors (right-hand y-axis). The vertical dashed lined at 37 tells us the optimal number of non-intercept terms retained where marginal increases in GCV

plot(mars1, which =1)

In addition to pruning the number of knots, earth::earth() allows us to also assess potential interactions between different hinge functions. The following illustrates by including a degree = 2 argument. You can see that now our model includes interaction terms between multiple hinge functions (i.e. h(Year\_Built-2003)\*h(Gr\_Liv\_Area-2274)) is an interaction effect for those houses built prior to 2003 and have less than 2,274 square feet of living space above ground).

# Fit a basic MARS model  
library(earth)  
mars2 <- earth(  
 Sale\_Price ~ .,   
 data = ames\_train,  
 degree = 2  
)  
  
# check out the first 10 coefficient terms  
summary(mars2) %>% .$coefficients %>% head(10)  
## Sale\_Price  
## (Intercept) 230781.444604  
## h(Gr\_Liv\_Area-2790) 94.795150  
## h(2790-Gr\_Liv\_Area) -50.976089  
## h(Year\_Built-2002) 9112.206563  
## h(2002-Year\_Built) -689.257584  
## h(Year\_Built-2002)\*h(2362-Gr\_Liv\_Area) -7.717206  
## h(Total\_Bsmt\_SF-1136) 63.681023  
## h(1136-Total\_Bsmt\_SF) -32.703993  
## h(Bsmt\_Unf\_SF-504) -25.086196  
## h(504-Bsmt\_Unf\_SF) 12.956904  
## Sale\_Price  
## (Intercept) 242611.63686  
## h(Gr\_Liv\_Area-2945) 144.39175  
## h(2945-Gr\_Liv\_Area) -57.71864  
## h(Year\_Built-2003) 10909.70322  
## h(2003-Year\_Built) -780.24246  
## h(Year\_Built-2003)\*h(Gr\_Liv\_Area-2274) 18.54860  
## h(Year\_Built-2003)\*h(2274-Gr\_Liv\_Area) -10.30826  
## h(Total\_Bsmt\_SF-1035) 62.11901  
## h(1035-Total\_Bsmt\_SF) -33.03537  
## h(Total\_Bsmt\_SF-1035)\*Kitchen\_QualTypical -32.75942

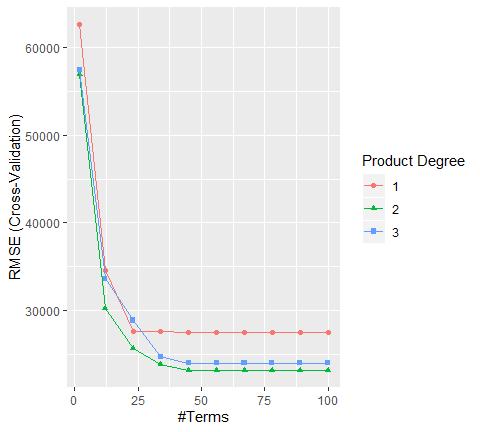
#### Tuning

Since there are two tuning parameters associated with our MARS model: the degree of interactions and the number of retained terms, we need to perform a grid search to identify the optimal combination of these hyperparameters that minimize prediction error (the above pruning process was based only on an approximation of cross-validated performance on the training data rather than an actual k-fold cross validation process). As in previous tutorials, we will perform a cross-validated grid search to identify the optimal mix. Here, we set up a search grid that assesses 30 different combinations of interaction effects (degree) and the number of terms to retain (nprune).

# create a tuning grid  
hyper\_grid <- expand.grid(  
 degree = 1:3,  
 nprune = seq(2, 100, length.out = 10) %>% floor()  
)

We can use **caret** to perform a grid search using 10-fold cross-validation. The model that provides the optimal combination includes second degree interactions and retain 34 terms. The cross-validated RMSE for these models are illustrated in Figure 5 and the optimal model’s cross validated RMSE is

# for reproducibility  
set.seed(123)  
  
# cross validation model  
library(caret)  
tuned\_mars <- train(  
 x = subset(ames\_train, select= -Sale\_Price),  
 y = ames\_train$Sale\_Price,  
 method = "earth",  
 metric = "RMSE",  
 trControl = trainControl(method = "cv", number = 10),  
 tuneGrid = hyper\_grid  
)  
  
# best model  
tuned\_mars$bestTune  
## nprune degree  
## 15 45 2  
# nprune degree  
# 14 34 2  
  
# plot residuals  
ggplot(tuned\_mars)



The above grid search helps to focus where we can further refine our model tuning. As a next step, we could perform a grid search that focuses in on a refined grid space for nprune (i.e. comparing 25-40 terms retained). However, for brevity we will leave this as an exercise for the reader.

So how does this compare to some other linear models for the Ames housing data? The following table compares the cross-validated RMSE for our tuned MARS model to a regular multiple regression model along with tuned principal component regression (PCR), partial least squares (PLS), and regularized regression (elastic net) models. By incorporating non-linear relationships and interaction effects, the MARS model provides a substantial improvement over the previous linear models that we have explored.

Notes: Notice that we standardize the features for the linear model but we did not for the MARS model. Whereas linear models tend to be sensitive to the scale of the features. MARS models are not.

# multiple regression  
set.seed(123)  
cv\_model1 <- train(  
 Sale\_Price ~.,  
 data = ames\_train,  
 method = "lm",  
 metric = "RMSE",  
 trControl = trainControl(method ="cv", number =10),  
 preProcess = c("zv", "center", "scale")  
)  
  
# prncipal component regression  
set.seed(123)  
cv\_model2 <- train(  
 Sale\_Price ~ .,   
 data = ames\_train,   
 method = "pcr",  
 trControl = trainControl(method = "cv", number = 10),  
 metric = "RMSE",  
 preProcess = c("zv", "center", "scale"),  
 tuneLength = 20  
 )  
  
# partial least squares regression  
set.seed(123)  
cv\_model3 <- train(  
 Sale\_Price ~ .,   
 data = ames\_train,   
 method = "pls",  
 trControl = trainControl(method = "cv", number = 10),  
 metric = "RMSE",  
 preProcess = c("zv", "center", "scale"),  
 tuneLength = 20  
 )  
  
# regularized regression  
set.seed(123)  
cv\_model4 <- train(  
 Sale\_Price ~ .,   
 data = ames\_train,  
 method = "glmnet",  
 trControl = trainControl(method = "cv", number = 10),  
 metric = "RMSE",  
 preProcess = c("zv", "center", "scale"),  
 tuneLength = 10  
)  
  
# extract out of sample perfromance measrues  
  
library(kableExtra)  
summary(resamples(list(  
 Multiple\_regression = cv\_model1,   
 PCR = cv\_model2,   
 PLS = cv\_model3,  
 Elastic\_net = cv\_model4,  
 MARS = tuned\_mars  
 )))$statistics$RMSE %>%  
 kableExtra::kable() %>%  
 kableExtra::kable\_styling(bootstrap\_options = c("striped", "hover"))

Min.

1st Qu.

Median

Mean

3rd Qu.

Max.

NA’s

Multiple\_regression

20945.23

25674.46

33769.49

37304.33

42966.80

80338.82

0

PCR

26924.92

29987.32

33890.70

33659.29

37180.37

40713.34

0

PLS

21147.31

25950.93

28902.72

29969.99

35260.78

41413.51

0

Elastic\_net

19938.23

24061.57

25260.68

29784.99

33829.27

53332.81

0

MARS

19657.89

21380.19

22678.92

23198.78

24192.78

30446.08

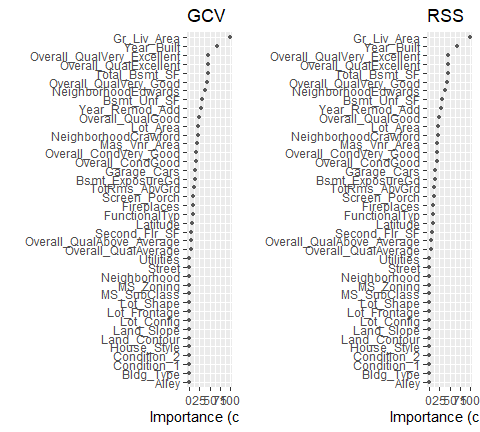
0

#### Feature interpretation

MARS models via earth::earth() include a backwards elimination feature selection routine that looks at reductions in the GCV estimate of error as each predictor is added to the model. This total reduction is used as the variable importance measure (value = "gcv). Since MARS will automatically include and exclude terms during the pruning process, , it essentially performs automated feature selection. If a predictor was never used in any of the MARS basis functions in the final model (after pruning), it has an importance value of zero. This is illustrated in Figure 6 where 27 features have importance values while the rest of the feature have an importance value of zero since they were not included in the final model.

Alternatively, you can also monitor the change in the residual sum of squares (RSS) as terms are added (value = "rss); however, you will see very little difference between these methods.

# variable importance plots  
library(vip)  
p1 <- vip(tuned\_mars, num\_features = 40, bar = FALSE, value ="gcv")+ ggtitle("GCV")  
p2 <- vip(tuned\_mars, num\_features = 40, bar = FALSE, value ="rss")+ ggtitle("RSS")  
  
gridExtra::grid.arrange(p1, p2, ncol = 2)



It is important to realize that variable importance will only measure the impact of the prediction error as features are included; however, it does not measure the impact for particular hinge functions created for a given feature. For example, in Figure 6, we see that Gr\_Liv\_Area and Year\_Built are the two most influential variables; however, variable importance does not tell us how our model is treating the non-linear patterns for each feature.

Also, if we look at the interaction terms, our model retained, we see interactions between different hinge functions for Gr\_Liv\_Area and Year\_Built.

coef(tuned\_mars$finalModel) %>%   
 broom::tidy() %>%   
 filter(str\_detect(names, "\\\*"))  
## # A tibble: 18 x 2  
## names x  
## <chr> <dbl>  
## 1 h(Year\_Built-2002) \* h(2362-Gr\_Liv\_Area) -7.72   
## 2 h(Year\_Remod\_Add-2007) \* h(Total\_Bsmt\_SF-1136) 9.59   
## 3 h(2007-Year\_Remod\_Add) \* h(Total\_Bsmt\_SF-1136) -1.47   
## 4 NeighborhoodEdwards \* h(Gr\_Liv\_Area-2790) -418.   
## 5 h(Lot\_Area-3874) \* h(3-Garage\_Cars) -0.722   
## 6 h(2002-Year\_Built) \* h(Year\_Remod\_Add-1971) 7.98   
## 7 h(2002-Year\_Built) \* h(1971-Year\_Remod\_Add) 5.12   
## 8 NeighborhoodCrawford \* h(2002-Year\_Built) 471.   
## 9 Bsmt\_ExposureGd \* h(Total\_Bsmt\_SF-1136) 32.9   
## 10 h(Total\_Bsmt\_SF-1136) \* h(Second\_Flr\_SF-144) 0.0732  
## 11 h(Total\_Bsmt\_SF-1136) \* h(144-Second\_Flr\_SF) 0.227   
## 12 h(Total\_Bsmt\_SF-1136) \* h(116-Screen\_Porch) -0.334   
## 13 Overall\_QualAverage \* h(2790-Gr\_Liv\_Area) 5.83   
## 14 h(2002-Year\_Built) \* h(TotRms\_AbvGrd-10) -247.   
## 15 h(Lot\_Area-3874) \* h(Latitude-42.0014) 9.69   
## 16 h(Lot\_Area-3874) \* h(42.0014-Latitude) -93.0   
## 17 Overall\_CondGood \* h(2002-Year\_Built) 212.   
## 18 Overall\_CondVery\_Good \* h(2002-Year\_Built) 222.  
## # A tibble: 16 x 2  
## names x  
## <chr> <dbl>  
## 1 h(Year\_Built-2003) \* h(Gr\_Liv\_Area-2274) 18.7   
## 2 h(Year\_Built-2003) \* h(2274-Gr\_Liv\_Area) -10.9   
## 3 h(Total\_Bsmt\_SF-1035) \* Kitchen\_QualTypical -33.1   
## 4 NeighborhoodEdwards \* h(Gr\_Liv\_Area-2945) -507.   
## 5 h(Lot\_Area-4058) \* h(3-Garage\_Cars) -0.791  
## 6 h(2003-Year\_Built) \* h(Year\_Remod\_Add-1974) 7.00   
## 7 Overall\_QualExcellent \* h(Total\_Bsmt\_SF-1035) 104.   
## 8 NeighborhoodCrawford \* h(2003-Year\_Built) 424.   
## 9 h(Lot\_Area-4058) \* Overall\_CondFair -3.29   
## 10 Overall\_QualAbove\_Average \* h(2003-Year\_Built) 136.   
## 11 h(Lot\_Area-4058) \* Overall\_CondGood 1.35   
## 12 Bsmt\_ExposureNo \* h(Total\_Bsmt\_SF-1035) -22.5   
## 13 NeighborhoodGreen\_Hills \* h(5-Bedroom\_AbvGr) 27362.   
## 14 Overall\_QualVery\_Good \* Bsmt\_QualGood -18641.   
## 15 h(2003-Year\_Built) \* Sale\_ConditionNormal 192.   
## 16 h(Lot\_Area-4058) \* h(Full\_Bath-2) 1.61

To better understand the relationship between these features and Sale\_Price, we can create partial dependence plot (PDPs) for each feature individually and also an interaction PDF. The individual PDPs illustrate that our model found that one knot in each feature provides the best fit.

For Gr\_Liv\_Area, as homes exceed 2,945 square feet, each additional square foot demands a higher marginal increase in sale price than homes with less that 2,945 square feet. Similarly, for homes built after 2003, there is a greter marginal effect on sales price based on the age of the home than for homes built priot to 2003.

The interaction plot (far right plot) illustrates the strong effect these two features have when combined.

p1 <- partial(tuned\_mars, pred.var = "Gr\_Liv\_Area", grid.resolution = 10) %>% ggplot2::autoplot()  
p2 <- partial(tuned\_mars, pred.var = "Year\_Built", grid.resolution = 10) %>% ggplot2::autoplot()  
  
# p3 <- partial(tuned\_mars, pred.var = c("Gr\_Liv\_Area", "Year\_Built"), grid.resolution = 10) %>%   
# plotPartial(levelplot = FALSE, zlab = "yhat", drape = TRUE, colorkey = TRUE, screen = list(z = -20, x = -60))  
  
gridExtra::grid.arrange(p1, p2, ncol = 2)

#### Final thoughts

MARS provides a great stepping stone into nonlinear modeling and tends to be fairly intuitive due to being closely related to multiple regression techniques. They are also easy to train and tune. This tutorial illustrated how incorporating non-linear relationships via MARS modeling greatly improved predictive accuracy on our Ames housing data. The following summarizes some of the advantages and disadvantages discussed regarding MARS modeling:

**Advantages**:

* Accurate if the local linear relationships are correct.
* Quick computation.
* Can work well even with large and small data sets.
* Provides automated feature selection.
* The non-linear relationship between the features and response are fairly intuitive.
* Can be used for both regression and classification problems.
* Does not require feature standardization.

**Disadvantages**: - Not accurate if the local linear relationships are incorrect. - Typically not as accurate as more advanced non-linear algorithms (random forests, gradient boosting machines). - The earth package does not incorporate more advanced spline features (i.e. Piecewise cubic models). - Missing values must be pre-processed.

#### Learning more

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

* An Introduction to Statistical Learning, Ch. 7
* Applied Predictive Modeling, Ch. 7
* Elements of Statistical Learning, Ch. 5
* [Notes on the earth package by Stephen Milborrow](http://www.milbo.org/doc/earth-notes.pdf)

### Regression trees & bagging

Basic regression trees partition a dataset into small groups and then fit a simple model for each subgroup. Unfortunately, a single tree model tends to be highly unstable and a poor predictor. However, by bootstrap aggregating ( **bagging**) regression trees, this technique can become quite powerful and effective. Moreover, this provides the fundamental basis of more complex tree-based models such as random forests and *gradient boosting machines*. This tutorial will get you started with regression trees and bagging.

#### tl;dr

This tutorial services as an introduction to the Regression Decision Trees. This tutorail will cover the following material:

* [Replication Requirements](#RT_RR): What you’ll need to reproduce the analysis in this tutorial.
* The idea: A quick overview of how regression trees work.
* [Basic implementation](#RT_IMP): Implementing regression trees in R.
* [Tuning](#RT_TUNE): Understanding the hyperparameters we can tune.
* [Bagging](#RT_BAGG): Improving performance by fitting many trees.
* [Learning more](#RT_MORE): Where you can learn more.

#### Replication Requirements

This tutorial leverages the following packages. Most of these packages are playing a supportive role while the main emphasis will be

library(rsample) # data splitting   
library(dplyr) # data wrangling  
library(rpart) # performing regression trees  
library(rpart.plot) # plotting regression trees  
library(ipred) # bagging  
library(caret) # bagging

To illustrate various regularization concepts we will use the Ames Housing data that has been included in the AmesHousing package.

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

#### Basic Implementation

**The Idea**

There are many methodologies for constructing regression trees but one of the oldest is known as the classification and regression tree (CART) approach developed by Breiman et al. (1984). This tutorial focuses on the regression part of CART. Basic regression trees partition a data set into smaller subgroups and then fit a simple constant for each observation in the subgroup. The partitioning is achieved by successive binary partitions (aka *recursive partitioning*) based on the different predictors. The constant to predict is based on the average response values for all observations that fall in that subgroup.

or example, consider we want to predict the miles per gallon a car will average based on cylinders (cyl) and horsepower (hp). All observations go through this tree, are assessed at a particular node, and proceed to the left if the answer is ?gyes?h or proceed to the right if the answer is ?gno?h. So, first, all observations that have 6 or 8 cylinders go to the left branch, all other observations proceed to the right branch. Next, the left branch is further partitioned by horsepower. Those 6 or 8 cylinder observations with horsepower equal to or greater than 192 proceed to the left branch; those with less than 192 hp proceed to the right. These branches lead to terminal nodes or leafs which contain our predicted response value. Basically, all observations (cars in this example) that do not have 6 or 8 cylinders (far right branch) average 27 mpg. All observations that have 6 or 8 cylinders and have more than 192 hp (far left branch) average 13 mpg.

This simple example can be generalized to state we have a continuous response variable and two inputs and . The recursive partitioning results in three regions () where the model predicts with a constanct for region :

$$ (X) = \_{m=1}^3 I(X\_1, X\_2) R\_m…(1)

$$

Howeer, an important question remains of how to grow a regression tree.

**Deciding on splits**

First, it is important to realize the partitioning of variables are done in a top-down, *greedy* fashion. This just means that a partition performed earlier in the tree will not change based on later partitions. But how are these partions made? The model begins with the entire data set, S, and searches every distinct value of every input variable to find the predictor and split value that partitions the data into two regions ($R\_1 and ) such that the overall sums of squares error are minimized:

Having found the best split, we partition the data into the two resulting regions and repeat the splitting process on each of the two regions. This process is continued until some stopping criterion is reached. What results is, typically, a very deep, complex tree that may produce good predictions on the training set, but is likely to overfit the data, leading to poor performance on unseen data.

For example, using the well-known Boston housing data set, I create three decision trees based on three different samples of the data. You can see that the first few partitions are fairly similar at the top of each tree; however, they tend to differ substantially closer to the terminal nodes. These deeper nodes tend to overfit to specific attributes of the sample data; consequently, slightly different samples will result in highly variable estimate/predicted values in the terminal nodes. By pruning these lower level decision nodes, we can introduce a little bit of bias in our model that help to stabilize predictions and will tend to generalize better to new, unseen data.

\_**Cost complexity criterion**

There is often a balance to be achieved in the depth and complexity of the tree to optimize predictive performance on some unseen data. To find this balance, we typically grow a very large tree as defined in the previous section and then prune back to find an *optimal subtree*.

We find the optimal subtree by using a *cost complexity parameter* () that penalizes our objective function in Eq.2 for the number of terminal nodes of the tree (T) as Eq.3.

For a given value of , we find the smallest pruned tree that has the lowest penalized error. If you are familiar with [regularized regression](http://uc-r.github.io/regularized_regression), you will realize the close association to the [lasso norm penalty](http://uc-r.github.io/regularized_regression#lasso).

As with these regularization methods, smaller penalties tend to produce more complex models, which result in larger trees. Whereas larger penalties result in much smaller trees.

consequently, as a tree grows larger, the reduction in the SSE must be greater than the cost compexity penalty. Typically, we evaluate multiple models across a spectrum of and use cross-validation to identify the optimal and, therefore, the optimal subtree.

**Strenghts and weakness**

There are several **advantages** to regression trees:

* They are very interpretable.
* Making prediction is fast (no compicated calculations, just looking up constants in the trees)
* It?fs easy to understand what variables are important in making the prediction. The internal nodes (splits) are those variables that most largely reduced the SSE.
* If some data is missing, we might not be able to go all the way down the tree to a leaf, but we can still make a prediction by averaging all the leaves in the sub-tree we do reach.
* The model provides a non-linear ?gjagged?h response, so it can work when the true regression surface is not smooth. If it is smooth, though, the piecewise-constant surface can approximate it arbitrarily closely (with enough leaves).
* There are fast, reliable algorithms to learn these trees.

But there are also some significant **weaknesses**:

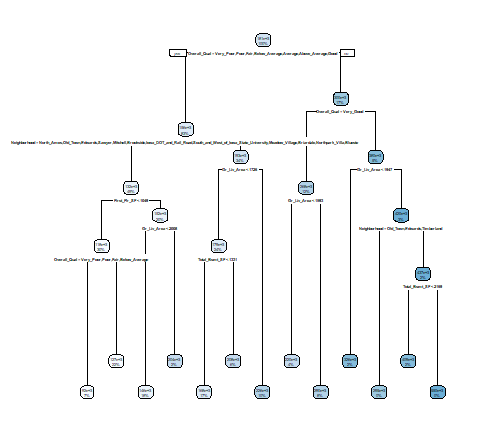
* Single regression trees have high variance, resulting in unstable predictions (an alternative subsample of traiinng data can significantly change the terminal nodes)
* Due to high variance, single regreesion trees have poor predictive accuracy

**Implementation details**

We can fit a regression tree using rpart and then visualize it using rpart.plot. The fitting process and the visual output of regression trees and classification trees are very similar. Both use the formula method for expressing the model (similar to lm).

However, when fitting a regression tree, we need to set method = "anova". By default, rpart will make an intelligent guess as to what the method value should be based on the data type of your response column, but it is recommended that you explicitly set the method for reproducibility reasons (since the auto-guess may change in the future).

m1 <- rpart::rpart(  
 formula = Sale\_Price ~ .,  
 data = ames\_train,  
 method = "anova"  
)  
  
m1 %>% rpart.plot::rpart.plot()

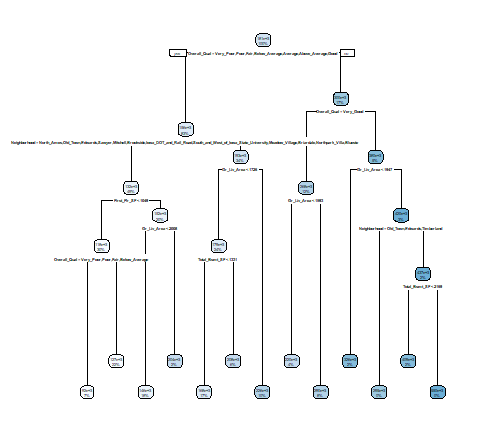


Once we have fit our we can take a peak at the m1 output. This just explains steps of the splits. For example, we start with 2051 observations at the root node (very beginning) and the first variables we split on (the first variable that optimizes a reduction in SSE) is Overall\_Qual. We see at the first node all observations with Overall\_Qual=Very\_Poor,Poor,Fair,Below\_Average,Average,Above\_Average,Good go to the 2nd (2)) branch. The total number of observations that follow this branch (), their average sales price () and SSE () are listed. If you look for the 3rd branch (3)) you will see that observations with Overall\_Qual=Very\_Good,Excellent,Very\_Excellent follow this branch and their average sales prices is 304571.10 and the SEE in this region is 2.874510e+12. Basically, this is telling us the most important variable that has the largest reduction in SEE initially is Overall\_Qual with those homes on the upper end of the quality spectrum having almost double the average sales price.

m1  
## n= 2051   
##   
## node), split, n, deviance, yval  
## \* denotes terminal node  
##   
## 1) root 2051 1.273987e+13 180775.50   
## 2) Overall\_Qual=Very\_Poor,Poor,Fair,Below\_Average,Average,Above\_Average,Good 1703 4.032269e+12 156431.40   
## 4) Neighborhood=North\_Ames,Old\_Town,Edwards,Sawyer,Mitchell,Brookside,Iowa\_DOT\_and\_Rail\_Road,South\_and\_West\_of\_Iowa\_State\_University,Meadow\_Village,Briardale,Northpark\_Villa,Blueste 1015 1.360332e+12 131803.50   
## 8) First\_Flr\_SF< 1048.5 611 4.924281e+11 118301.50   
## 16) Overall\_Qual=Very\_Poor,Poor,Fair,Below\_Average 152 1.053743e+11 91652.57 \*  
## 17) Overall\_Qual=Average,Above\_Average,Good 459 2.433622e+11 127126.40 \*  
## 9) First\_Flr\_SF>=1048.5 404 5.880574e+11 152223.50   
## 18) Gr\_Liv\_Area< 2007.5 359 2.957141e+11 145749.50 \*  
## 19) Gr\_Liv\_Area>=2007.5 45 1.572566e+11 203871.90 \*  
## 5) Neighborhood=College\_Creek,Somerset,Northridge\_Heights,Gilbert,Northwest\_Ames,Sawyer\_West,Crawford,Timberland,Northridge,Stone\_Brook,Clear\_Creek,Bloomington\_Heights,Veenker,Green\_Hills 688 1.148069e+12 192764.70   
## 10) Gr\_Liv\_Area< 1725.5 482 5.162415e+11 178531.00   
## 20) Total\_Bsmt\_SF< 1331 352 2.315412e+11 167759.00 \*  
## 21) Total\_Bsmt\_SF>=1331 130 1.332603e+11 207698.30 \*  
## 11) Gr\_Liv\_Area>=1725.5 206 3.056877e+11 226068.80 \*  
## 3) Overall\_Qual=Very\_Good,Excellent,Very\_Excellent 348 2.759339e+12 299907.90   
## 6) Overall\_Qual=Very\_Good 249 9.159879e+11 268089.10   
## 12) Gr\_Liv\_Area< 1592.5 78 1.339905e+11 220448.90 \*  
## 13) Gr\_Liv\_Area>=1592.5 171 5.242201e+11 289819.70 \*  
## 7) Overall\_Qual=Excellent,Very\_Excellent 99 9.571896e+11 379937.20   
## 14) Gr\_Liv\_Area< 1947 42 7.265064e+10 325865.10 \*  
## 15) Gr\_Liv\_Area>=1947 57 6.712559e+11 419779.80   
## 30) Neighborhood=Old\_Town,Edwards,Timberland 7 8.073100e+10 295300.00 \*  
## 31) Neighborhood=College\_Creek,Somerset,Northridge\_Heights,Northridge,Stone\_Brook 50 4.668730e+11 437207.00   
## 62) Total\_Bsmt\_SF< 2168.5 40 1.923959e+11 408996.90 \*  
## 63) Total\_Bsmt\_SF>=2168.5 10 1.153154e+11 550047.30 \*

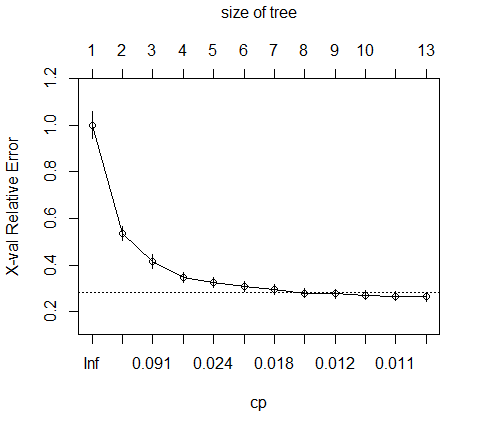
We can visualize our model with rpart.plot. rpart.plot has many plotting options, which we will leave to the reader to explorer. However, in the default print, it will show the percentage of data that fall to that node and the average sales price for that branch. One thing you may notice is that this tree contains 11 internal nodes resulting in 12 terminal nodes. Basically, this tree is partitioning on 11 variables to produce its model. However, there are 80 variables in ames\_train. So what happened?

library(rpart.plot)  
rpart.plot(m1)



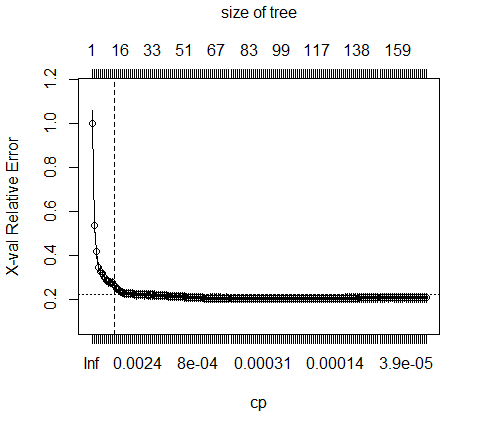
Behind the scenes, rpart is automatically applying a range of cost compexity ( values to prune the tree). To compare the error for each value, rpart performs a 10-fold cross validation so that the error associated with a given value is computed on the hold-out validation data. In this example we find diminishing returns after 12 terminal nodes as illustrated below (y-axis is cross validation error, lower x-axis is cost complexity value, upeer x-axis is the number of terminal nodes (tree size =). You may also notice the dashed line which goes through the point . Breiman et al. (1984) suggested that in actual practice, it is common to instead use the smallest tree within 1 standard deviation of the minimum cross validation error (aka the 1-SE rule). Thus, we could use a tree with 9 terminal nodes and reasonably expect to experience similar results within a small margin of error.

rpart::plotcp(m1)



To illustrate the point of selecting a tree with 12 terminal nodes (or 9 if you go by the 1-SE rule), we can force rpart to generate a full tree by using cp=0 (no penalty results in a fully grown tree). we can see that after 12 terminal nodes, we see diminishing returns in error reduction as the tree grows deeper. Thus, we can significantly prune our tree and still achieve minimal expected error.

library(rpart)  
  
m2 <- rpart(  
 formula = Sale\_Price ~.,  
 data = ames\_train,  
 method = "anova",  
 control = list(cp = 0, xval = 10)  
)  
  
plotcp(m2)  
abline(v = 12, lty = "dashed")



So, by default, rpart is performing some automated tuning, with an optimal subtree of 11 splits, 12 terminal nodes, and a cross-validated error of 0.272 (note that this error is equivalent to the PRESS statistic but not the MSE). However, we can perform additional tuning to try improve model performance.

m1$cptable %>% broom::tidy()   
## # A tibble: 12 x 5  
## CP nsplit rel.error xerror xstd  
## <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 0.467 0 1 1.00 0.0586  
## 2 0.120 1 0.533 0.535 0.0312  
## 3 0.0696 2 0.413 0.415 0.0306  
## 4 0.0256 3 0.344 0.346 0.0221  
## 5 0.0220 4 0.318 0.324 0.0218  
## 6 0.0202 5 0.296 0.307 0.0213  
## 7 0.0167 6 0.276 0.296 0.0211  
## 8 0.0119 7 0.259 0.280 0.0190  
## 9 0.0113 8 0.247 0.276 0.0194  
## 10 0.0111 9 0.236 0.270 0.0190  
## 11 0.0106 11 0.214 0.267 0.0188  
## 12 0.01 12 0.203 0.264 0.0188

#### Tuning

In addition to the cost complexity paramter(), it is also common to tune:

* minsplit: the minimum nunber of data points required to attempt a split before it is forced to create a terminal node. The default is 20. Making this smaller allows for terminal nodes that may contain only a handful of observations to create the predicted value.
* maxdepth: the maximum number of internal nodes between the root node and the terminal nodes. The default is 30, which is quite liberal and allows for fairly large trees to be built.

rpart uses special control argument where we provide a list of hyperparameter values. For example, if we wanted to assess a model with minsplit=10 and maxdepth=12, we could execute the following:

m3 <- rpart(  
 formula = Sale\_Price ~.,  
 data = ames\_train,  
 method = "anova",  
 control = list(minsplit=10, maxdepth=12, xval=10)  
)  
  
m3$cptable %>% broom::tidy()  
## # A tibble: 12 x 5  
## CP nsplit rel.error xerror xstd  
## <dbl> <dbl> <dbl> <dbl> <dbl>  
## 1 0.467 0 1 1.00 0.0585  
## 2 0.120 1 0.533 0.534 0.0309  
## 3 0.0696 2 0.413 0.415 0.0304  
## 4 0.0256 3 0.344 0.346 0.0219  
## 5 0.0220 4 0.318 0.326 0.0217  
## 6 0.0202 5 0.296 0.306 0.0211  
## 7 0.0167 6 0.276 0.306 0.0218  
## 8 0.0119 7 0.259 0.292 0.0206  
## 9 0.0113 8 0.247 0.287 0.0244  
## 10 0.0111 9 0.236 0.285 0.0244  
## 11 0.0106 11 0.214 0.283 0.0233  
## 12 0.01 12 0.203 0.274 0.0226

Although useful, this approach requires you to manually assess multiple models. Rather, we can perform a grid search to automatically search across a range of differently tuned models to identify the optimal hyperparameter setting.

To perform a grid search we first create our hyperparameter grid. In this example, I search a range of minsplit from 5-20 and vary maxdepth from 8-15 (since our original model found an optimal depth of 12). What results is 128 different combinations, which requires 128 different models.

hyper\_grid <- expand.grid(  
 minsplit = seq(5, 20, 1),  
 maxdepth = seq(8, 15, 1)  
)  
  
head(hyper\_grid)  
## minsplit maxdepth  
## 1 5 8  
## 2 6 8  
## 3 7 8  
## 4 8 8  
## 5 9 8  
## 6 10 8

To automate the modeling we simply set up a for loop and iterate through each minsplit and maxdepth combination. We save each model into its own list item.

models <- list()  
  
for (i in 1:nrow(hyper\_grid)){  
   
 # get minsplit, maxdepth values at row i  
 minsplit <- hyper\_grid$minsplit[i]  
 maxdepth <- hyper\_grid$maxdepth[i]  
   
 # train a model and store in the list  
 models[[i]] <- rpart(  
 formula = Sale\_Price~.,  
 data = ames\_train,  
 method = "anova",  
 control = list(minsplit = minsplit,  
 maxdepth = maxdepth)  
 )  
}  
  
# option - purrr::map2()  
  
h <- hyper\_grid %>%   
 mutate(models=map2(minsplit, maxdepth,  
 ~rpart(  
 formula = Sale\_Price~.,  
 data = ames\_train,  
 method = "anova",  
 control = list(minsplit = .x,  
 maxdepth = .y)  
 ),  
 tidy = broom::tidy(models)))  
  
h$models[[1]]$cptable[,"xerror"]  
## 1 2 3 4 5 6 7   
## 1.0007823 0.5345050 0.4149125 0.3470926 0.3300224 0.3113462 0.3044571   
## 8 9 10 11 12   
## 0.2927465 0.2889267 0.2808702 0.2779345 0.2864204

We can now create a function to extract the minimum error associated with the optimal cost complexity value for each model. After a little data wrangling to extract the optimal value and its respective error, adding it back to our grid, and filter for top 5 minimal error values we see that the optimal model makes a slight improvement over our ealier model (xerror of 0.242 versus 0.272).

# function to get optimal cp  
get\_cp <- function(x) {  
 min <- which.min(x$cptable[, "xerror"])  
 cp <- x$cptable[min, "CP"]   
}  
  
# function to get minimum error  
get\_min\_error <- function(x) {  
 min <- which.min(x$cptable[, "xerror"])  
 xerror <- x$cptable[min, "xerror"]}  
  
  
h %>%  
 mutate(  
 cp = purrr::map\_dbl(models, get\_cp),  
 error = purrr::map\_dbl(models, get\_min\_error)) %>%   
 arrange(error) %>%  
 top\_n(-5, wt = error)  
## minsplit maxdepth cp error  
## 1 15 12 0.0100000 0.2419963  
## 2 5 13 0.0100000 0.2422198  
## 3 7 10 0.0100000 0.2438687  
## 4 17 13 0.0108982 0.2468053  
## 5 19 13 0.0100000 0.2475141

If we were satisfied with these results, we could apply this final optimal model and predict on our test set. The final RMSE is which suggests that, on average, our predicted sales prices are about $39,145 off from the actual sales price.

optimal\_tree <- rpart(  
 formula = Sale\_Price ~.,  
 data = ames\_train,  
 method = "anova",  
 control = list(minsplit = 11,  
 maxdepth = 8,  
 cp = 0.01)  
)  
  
pred <- predict(optimal\_tree, newdata = ames\_test)  
RMSE(pred = pred, obs = ames\_test$Sale\_Price)  
## [1] 39852.01

#### Bagging

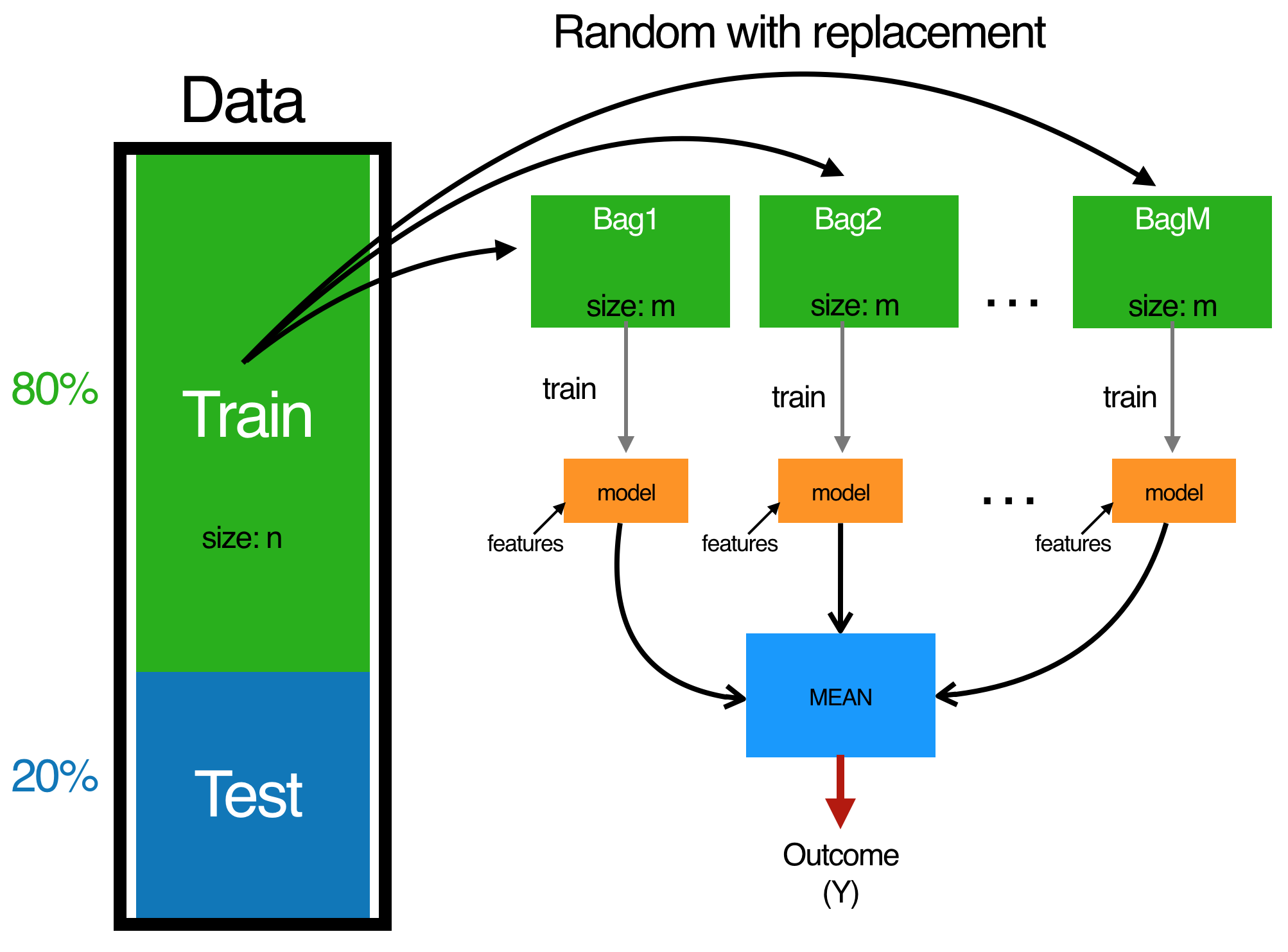
**The Idea**

As previously mentioned, single tree models suffer from high varianace. Although pruning the tree helps reduce this variance, there are alternative methods that actually exploit the variability of single trees in a way that can significantly improve performance over and above that of *Bootstrap Aggregate (Bagging)* is one such approach (oroginally proposed by [Breiman, 1996](https://link.springer.com/article/10.1023%2FA%3A1018054314350))

Bagging combines and averages multiple models. Averaging across multiple trees reduces the variability of any tone tree and reduces overfitting, which imrpoves predictive performance. Bagging follows three simple steps:

1. Create m [bootstrap samples](http://uc-r.github.io/bootstrapping) from the training data.Boostrapped samples allow us to create many slightly different data sets but with the same distribution as the overall training set.
2. For each boostrap sample train a single, unpruned regression tree.
3. Average individual predictions from each tree to create an overall average predicted value.

knitr::include\_graphics("C:/Protected/Data Science/UC Business Analytics/image/bagging3.png")



This process can actually be applied to any regression or classification model; however, it provides the greatest improvement for models that have high variance. For example, more stable parametric models such as linear regression and multi-adaptive regression splines tend to experience less improvement in predictive performance.

One benefit of bagging is that, on average, a bootstrap sample will contain 63% of the training data. This leaves about 33% of the data out of the bootstrapped sample. We call this is the **out-of-bag (OOB)** sample. We can use the OOB observations to estimate the model’s accuracy, creating a natural cross-validation process.

**Bagging with ipred**

Fitting a bagged tree model is quite simple. Instead of using rpart we use opred::bagging. We use coob=TRUE to use the OOB sample to estimate the test error. We see that our initial estimate error is close to $3K less than the test error we achieved with our singple optimal tree (36543 vs 39145).

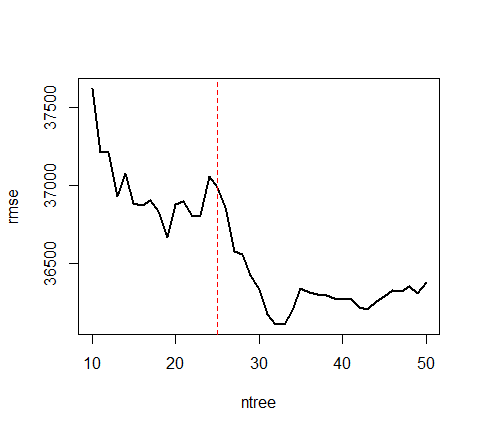
# make boostrapping reproducible  
set.seed(123)  
  
# train bagged model  
bagged\_m1 <- bagging(  
 formula = Sale\_Price ~ .,  
 data = ames\_train,  
 coob = TRUE  
)  
  
bagged\_m1  
##   
## Bagging regression trees with 25 bootstrap replications   
##   
## Call: bagging.data.frame(formula = Sale\_Price ~ ., data = ames\_train,   
## coob = TRUE)  
##   
## Out-of-bag estimate of root mean squared error: 36991.67  
##   
## Bagging regression trees with 25 bootstrap replications   
##   
## Call: bagging.data.frame(formula = Sale\_Price ~ ., data = ames\_train,   
## coob = TRUE)  
##   
## Out-of-bag estimate of root mean squared error: 36543.37

One thing to note is that typically, the more trees the better. As we add more trees we are averaging over more high variance single trees. What result is that early on, we see a dramatic reduction in variance (and hence our error) and eventually the reduction in error will flatline signaling an appropriate number of trees to create a stable model.

Rarely you need more than 50 trees to stabilize the error.

By default, bagging performs 25 booststrap samples and trees but we may require more. We can assess the error versus number of trees as below. We see that the error is stabilizing at about 25 trees so we will likely not gain much improvement by simply bagging more trees.

# assess 10-50 bagged trees  
ntree <- 10:50  
  
# create empty vector to store OOB RMSE values  
rmse <- vector(mode = "numeric", length=length(ntree))  
  
for (i in seq\_along(ntree)){  
 # reproducibility  
 set.seed(123)  
   
 # perform bagged model  
 model <- bagging(  
 formula = Sale\_Price ~.,  
 data = ames\_train,  
 coob = TRUE,  
 nbagg = ntree[i]  
 )  
   
 # get OOB error  
 rmse[i] <- model$err  
}  
  
plot(ntree, rmse, type = 'l', lwd = 2)  
abline(v = 25, col = "red", lty = "dashed")



**Bagging with caret**

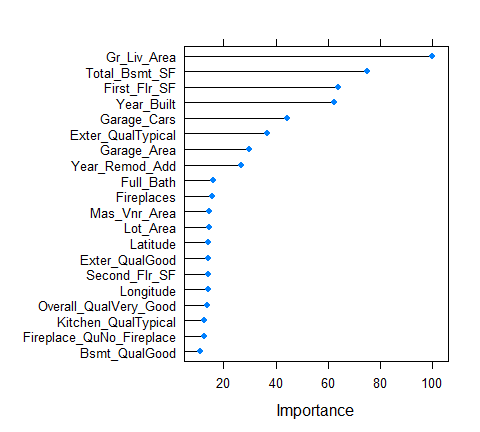
Bagging with ipred is quite simple; however, there are some additional benefits of bagging with caret.

1. It is easier to perform cross-validation. Although we can use the OOBE error, performing cross validation will provide a more robust understanding of the true expected test error.
2. We can assess variable important across the bagged trees.

Here, we performa a 10-fold corss-validated model. We see that the cross-validated RMSE is . We also assess the top 20 variables from our model.

Variable importance for regression trees is measured by assessing the total amount SSE is decreased by splits over a given predictor, averaged over all trees. The predictors with the largest average impact to SSE are considered most important. The importance value is simple the relative mean decrease in SSE compared to the most important variables.

# specify 10-fold cross validation  
library(caret)  
ctrl <- trainControl(method = "cv", number = 10)  
  
# CV bagged model  
bagged\_cv <- train(  
 Sale\_Price ~.,  
 data = ames\_train,  
 method = "treebag",  
 trControl = ctrl,  
 importance = TRUE  
)  
  
# assess results  
bagged\_cv  
## Bagged CART   
##   
## 2051 samples  
## 80 predictor  
##   
## No pre-processing  
## Resampling: Cross-Validated (10 fold)   
## Summary of sample sizes: 1846, 1845, 1846, 1845, 1847, 1847, ...   
## Resampling results:  
##   
## RMSE Rsquared MAE   
## 35854.02 0.8009063 23785.85  
  
## Bagged CART   
##   
## 2051 samples  
## 80 predictor  
##   
## No pre-processing  
## Resampling: Cross-Validated (10 fold)   
## Summary of sample sizes: 1846, 1845, 1847, 1845, 1846, 1847, ...   
## Resampling results:  
##   
## RMSE Rsquared MAE   
## 36477.25 0.8001783 24059.85  
  
# plot most important variables  
plot(varImp(bagged\_cv), 20)



#### Learning more

Decision trees provide a very intuitive modeling approach that have several, flexible benefits. Unfortunately, they suffer from high variance; however, when you combine them with bagging you can minimize this drawback. Moreover, bagged trees provides the fundamental basis for more complex and powerful algorithms such as random forests and gradient boosting machines. To learn more I would start with the following resources listed in order of complexity:

* An Introduction to Statistical Learning
* Applied Predictive Modeling
* [Classification and Regression Trees](https://www.amazon.com/Classification-Regression-Wadsworth-Statistics-Probability/dp/0412048418)
* The Elements of Statistical Learning
* [Bagging Predictors](https://link.springer.com/article/10.1023%2FA%3A1018054314350)

### Random forests

[Bagging (bootstrap aggregating) regression trees](http://uc-r.github.io/regression_trees) is a technique that can turn a single tree model with high variance and poor preditive power into a fairly accurate prediction function.

However, bagging trees typically suffers from tree correlation, which reduces the overall performance of the model.

Random forests are a modification of bagging that builds a large collection of *de-correlated* trees and have become a very popular “out-of-the-box” learning algorithm that enjoys good predictive peformance. This tutotiral will cover the fundamentals of random forests.

#### tl;dr

This tutorial serves as an introduction to the random forests. This tutorial will cover the following material:

* [Replication Requirements](#RF_RR): What you?fll need to reproduce the analysis in this tutorial.
* [The idea](#RF_Idea): A quick overview of how random forests work.
* [Basic implementation](#RF_BI): Implementing regression trees in R.
* [Tuning](#RF_Tune): Understanding the hyperparameters we can tune and performing grid search with ranger & h2o.
* [Predicting](#RF_Pred): Apply your final model to a new data set to make predictions.
* [Learning more](#RF_LM): Where you can learn more

#### Replication Requirements

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

library(rsample) # data splitting   
library(randomForest) # basic implementation  
library(ranger) # a faster implementation of randomForest  
library(caret) # an aggregator package for performing many machine learning models  
library(h2o) # an extremely fast java-based platform

To illustrate various regularization concepts we will use the Ames Housing data that has been included in the AmesHousing package.

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

#### The idea

Random forests are built on the same fundamental principles as decision trees and bagging (check out this [tutorial](http://uc-r.github.io/regression_trees) if you need a refresher on these techniques).

Bagging trees introduces a random component in to the tree building process that reduces the variance of a single tree’s prediction and improve predictive performance. However, the trees in bagging are not completely independent of each other since all the original predictors are considered at every split of every tree. Rather, trees from different bootstrap samples typically have similar structure to each other (especially at the top of the tree) due to underlying relationships.

For example, if we create six decision trees with different bootstrapped samples of the Boston housing data, we see that the top of the trees all have a very similar structure. Although there are 15 predictor variables to split on, all six trees have both lstat and rm variables driving the first few splits.

This characteristic is known as *tree correlation* and prevents bagging from optimally reducing variance of the predictive values. In order to reduce variance further, we need to minimize the amount of correlation between the trees.

This can be achieved by injecting more randomness into the tree-growing process. random forecast achieve this in two ways:

1. **Bootstrap**: similar to bagging, each tree is grown to a bootstrap resampled data set, which makes them different and somewhat decorrelates them.
2. **Split-variable randomization**: each time a split is to be performed, the search for the split variable is limited to a random subset of *m* of the *p* variables.

For regression trees, typical default values are , but this should be considere a tuning parameter. When , the randomization amounts to using only step 1 and is the same as **bagging**.

The basic algorthim for a regression random forest can be generalized to the following:

1. Given training data set  
2. Select number of trees to build (ntrees)  
3. for i = 1 to ntrees do  
4. | Generate a bootstrap sample of the original data  
5. | Grow a regression tree to the bootstrapped data  
6. | for each split do  
7. | | Select m variables at random from all p variables  
8. | | Pick the best variable/split-point among the m  
9. | | Split the node into two child nodes  
10. | end  
11. | Use typical tree model stopping criteria to determine when a tree is complete (but do not prune)  
12. end

since the algorithm randomly selects a bootstrap sample to traini on **and** predictors to use at each split, tree correlation will be lessened beyond bagged trees.

**OOB error vs test set error**

Similar to bagging, a natural benefit of the bootstrap resampling process is that random forests have an out-of-bag (OOB) sample that provides an eifficient and reasonable approximation of the test error. This provides a built-in validation set without an extra work on your part, and you do not need to sacrifice any of your training data to use for validation.

This makes identifying the number of trees required to stabilize the error rate during tuning more efficient; however, as illustrated below, some difference between the OOB erro and test error are expected.

Furthermore, many packages do not keep track of which observations were part of the OOB sample for a given tree and which were not. If you are comparing multiple models to one-another, you?fd want to score each on the same validation set to compare performance. Also, although technically it is possible to compute certain metrics such as root mean squared logarithmic error (RMSLE) on the OOB sample, it is not built in to all packages. So if you are looking to compare multiple models or use a slightly less traditional loss function you will likely want to still perform cross validation.

**Advantages & Disadvantages** *Advantages:*

* Typically have very good performance
* Remarkably good ?gout-of-the box?h - very little tuning required
* Built-in validation set - don?ft need to sacrifice data for extra validation
* No pre-processing required
* Robust to outliers

*Disdvantages:*

* Can become slow on large data sets
* Although accurate, often cannot compete with advanced boosting algorithms
* Less interpretable

#### Basic implementation

There are over 20 random forest packages in R.1 To demonstrate the basic implementation we illustrate the use of the randomForest package, the oldest and most well known implementation of the Random Forest algorithm in R. However, as your data set grows in size randomForest does not scale well (although you can parallelize with foreach). Moreover, to explore and compare a variety of tuning parameters we can also find more effective packages. Consequently, in the [Tuning](#RF_Tune) section we illustrate how to use the ranger and h2o packages for more efficient random forest modeling.

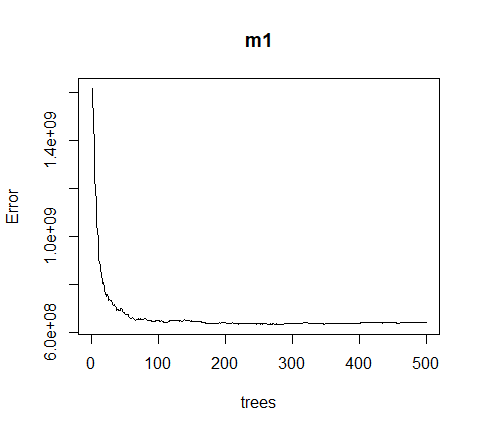
randomForest::randomForest can use the formula or separate x, y matrix notation for specifying our model. Below we apply the default randomForest model using the formulaic specification.

The default random forest performs 500 trees and randomly selected predictor variables at each split. Averaging across all 500 trees provides an OOB

# for reproducibility  
set.seed(123)  
  
# default RF model  
m1 <- randomForest(  
 formula = Sale\_Price ~.,  
 data = ames\_train  
)  
  
# m1.caret <- caret::train(  
# Sale\_Price ~.,   
# data=ames\_train,   
# method = "rf",  
# preProcess = c("center", "scale", "zv"),  
# trcontrol = trainControl(method="cv"))  
# https://www.rdocumentation.org/packages/caret/versions/6.0-82/topics/preProcess  
  
# predicted value computation  
# predRF <- predict(m1.caret, ames\_test)  
  
# RMSE computation  
# sqrt(sum((ames\_train$Sale\_Price - predRF)^2))

Plotting the model will illustrate the error rate as we average across more trees and shows that our error rate stabilizies with around 100 trees but continues to decrease slowly around 300 or so trees.

plot(m1)



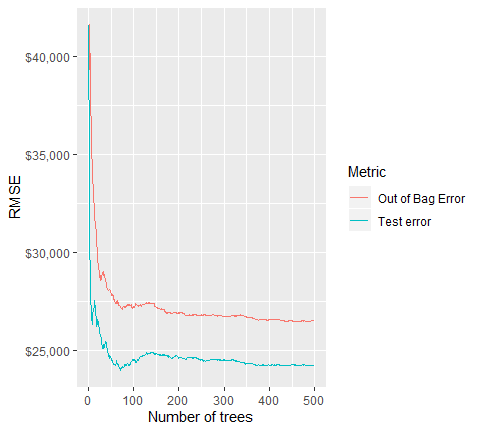
The plotted error rate above is based on the OOB sample error and can be accessed directly at m1$mse. Thus, we can find which number of trees providing the lowest error rate, which is 344 trees providing an average home sales price error of

# number of trees with lowest MSE  
which.min(m1$mse)  
## [1] 280  
  
# RMSE of this optimal random forest  
sqrt(m1$mse[which.min(m1$mse)])  
## [1] 25135.88

randomForest also allows us to use a validatio set to measure predictive accuracy if we did not want to use the OOB samples. Here we split our training set further to create a training and validation set.

We then supply the validation data in the xtest and ytest arguments.

# create training and validation data  
set.seed(123)  
valid\_split <- initial\_split(ames\_train, .8)  
  
# training data  
ames\_train\_v2 <- analysis(valid\_split)  
  
# validation data  
ames\_valid <- assessment(valid\_split)  
x\_test <- ames\_valid[setdiff(names(ames\_valid), "Sale\_Price")]  
y\_test <- ames\_valid$Sale\_Price  
  
rf\_oob\_comp <- randomForest(  
 formula = Sale\_Price ~ .,  
 data = ames\_train\_v2,  
 xtest = x\_test,  
 ytest = y\_test  
)  
  
# extract OOB & validation errors  
oob <- sqrt(rf\_oob\_comp$mse)  
validation <- sqrt(rf\_oob\_comp$test$mse)  
  
# compare errror rates  
tibble::tibble(  
 `Out of Bag Error` = oob,  
 `Test error` = validation,  
 ntrees = 1:rf\_oob\_comp$ntree  
) %>%  
 gather(Metric, RMSE, -ntrees) %>%  
 ggplot(aes(ntrees, RMSE, color = Metric)) +  
 geom\_line() +  
 scale\_y\_continuous(labels = scales::dollar) +  
 xlab("Number of trees")



Random forests are one of the best “out-of-the-box” machine learning algorithms. They typically perform remarkably well with very little tuning required. For example, as we saw above, we were able to get an RMSE of less than $30K without any tuning which is over a $6K reduction to the RMSE achieved with a fully-tuned [bagging model](http://uc-r.github.io/regression_trees#bag) and $4K reduction to to a fully-tuned [elastic net model](http://uc-r.github.io/regularized_regression#elastic). However, we can still seek improvement by tuning our random forest model.

#### Tuning

Random forests are fairly easy to tune since there are only a handful of tuning parameters. Typically, the primary concern when starting out is tuning the number of candidate variables to select from each split. However, there are a few additional hyperparameters that we should be aware of. Although the argument names may differ across packages, these hyperparameters should be present:

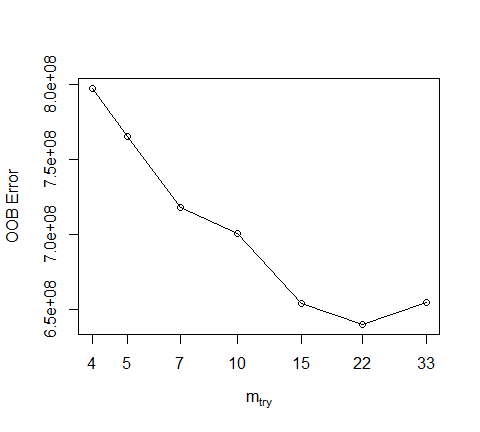
* ntree: number of trees. We want enough trees to stabalize the error but using too many trees is unncessarily inefficient, especially when using large data sets.
* mtry: the number of variables to randomly sample as candiates at each split. When mtry=p, the model equates to bagging. When mtry=1, the split variable is completely random, so all variable get a chance btu can lead to overly biased results. A common suggestion is to start with 5 values evenly spaced across the range from 2 to .
* sampsize: the number of samples to train on. The default value is 63.25% of the training set since this is the expected value of unique observations in the bootstrap sample. Lower sample sizes can reduce the training time but may introduce more bias than necessary. Increasing the sample size can increase performance but at the risk of overfitting because it introduces more variance. Typically, when tuning this parameter we stay near the 60-80% range.
* nodesize: minimum number of samples within the terminal nodes. Controls the complexity of the trees. Smaller node size allows for deeper, more complex trees and smaller node results in shallower trees. This is another bias-variance tradeoff where deeper trees introduce more variance (risk of overfitting) and shallower trees introduce more bias (risk of not fully capturing unique patters and relatonships in the data).
* maxnodes: maximum number of terminal nodes. Another way to control the complexity of the trees. More nodes equates to deeper, more complex trees and less nodes result in shallower trees.

**Initial tuning with randomForest**

If we are interested with just starting out and tuning the mtry parameter we can use randomForest::tuneRF for a quick and easy tuning assessment.

tuneRf will start at a value of mtry that you supply and increase by a certain step factor until the OOB error stops improving be a specified amount. For example, the below starts with mtry=5 and increases by a factor of 1.5 until the OOB error stops improving by 1%. Note that tuneRF requires a separate xy specification. We see that the optimal mtry value in this sequence is very close to the default mtry value of .

# names of features  
features <- setdiff(names(ames\_train), "Sale\_Price")  
  
set.seed(123)  
  
m2 <- tuneRF(  
 x = ames\_train[features],  
 y = ames\_train$Sale\_Price,  
 ntreeTry = 500,  
 mtryStart = 5,  
 stepFactor = 1.5,  
 improve = 0.01,  
 trace = FALSE # to not show real-time progress  
)  
## -0.04236505 0.01   
## 0.0614441 0.01   
## 0.02425961 0.01   
## 0.06634214 0.01   
## 0.02149491 0.01   
## -0.02257957 0.01



## -0.02973818 0.01   
## 0.0607281 0.01   
## 0.01912042 0.01   
## 0.02776082 0.01   
## 0.01091969 0.01   
## -0.01001876 0.01

**Full grid search with ranger**

To perfrom a larger grid search across several hyperparameters, we will need to create a grid and loop through each hyperparameter combination and evaluate the model. Unfortunately, this is where randomForest becomes quite inefficient since it does not scale well. Instead, we can use ranger wihch is a C++ implementation of Brieman’s random forest algorithm, and as the folliwing illustrates, is over 6 times faster than randomForest.

#randomForest speed  
system.time(  
 ames\_randomForest <- randomForest(  
 formula = Sale\_Price ~.,  
 data = ames\_train,  
 ntree = 500,  
 mtry = floor(length(features)/3)  
 )  
)  
## user system elapsed   
## 176.88 0.51 194.12  
## user system elapsed   
## 55.371 0.590 57.364  
  
# ranger speed  
system.time(  
 ames\_ranger <- ranger(  
 formula = Sale\_Price ~.,  
 data = ames\_train,  
 num.tree = 500,  
 mtry = floor(length(features)/3)  
 )  
)  
## user system elapsed   
## 14.94 0.12 5.64  
## user system elapsed   
## 9.267 0.215 2.997

To perform the grid search, first we want to construct our grid of hyperparameters.We are going to search across 96 different models with varying mtry, minimum node size and sample size.

# hyperparameter grid srearch  
  
hyper\_grid <- expand.grid(  
 mtry = seq(20, 30, by = 2),  
 node\_size = seq(3, 9, by = 2),  
 sample\_size = c(.55, .632, .7, .8),  
 OOB\_RMSE = 0  
)  
   
# total number of combinations  
nrow(hyper\_grid)  
## [1] 96

We look through each yperparameter combination and apply 500 trees since our previous examples illustrated that 500 was plenty to achieve a stable error rate. Also note that we set the random number generator seed. This allows us to consistently sample the same observations for each sample size and make it more clear the impact that each change makes. Our OOB RMSE ranges between ~26,000-27,000. Our top 10 performing models all have RMSE values right around 26,000 and the results show that models with slighly larger sample sizes (70-80%) and deeper trees (3-5 observations in an terminal node) perform best. We get a full range of mtry values showing up in our top 10 so is does not look like that is over influential.

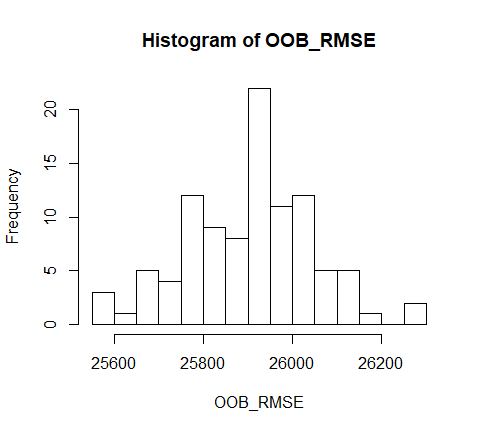
for (i in 1:nrow(hyper\_grid)){  
 # train model  
 model <- ranger(  
 formula = Sale\_Price ~.,  
 data = ames\_train,  
 num.trees = 500,  
 mtry = hyper\_grid$mtry[i],  
 min.node.size = hyper\_grid$node\_size[i],  
 sample.fraction = hyper\_grid$sample\_size[i],  
 seed = 123)  
   
 # add OOB error to grid  
 hyper\_grid$OOB\_RMSE[i] <- sqrt(model$prediction.error)  
}  
  
hyper\_grid %>%   
 dplyr::arrange(OOB\_RMSE) %>%   
 head()  
## mtry node\_size sample\_size OOB\_RMSE  
## 1 28 3 0.8 25477.32  
## 2 28 5 0.8 25543.14  
## 3 28 7 0.8 25689.05  
## 4 28 9 0.8 25780.86  
## 5 30 3 0.8 25818.27  
## 6 24 3 0.8 25838.55

Although, random forests typically perform quite well with categorical variables in their original columnar form, it is worth checking to see if alternative encodings can increase performance. For example, the following one-hot encodes our categorical variables which produces 353 predictor variables versus the 80 we were using above. We adjust our mtry parameter to search from 50-200 random predictor variables at each split and re-perform our grid search. The results suggest that one-hot encoding does not improve performance.

# one-hot encode our categorical variables  
# caret package dummyVars()  
one\_hot <- dummyVars(~., ames\_train, fullRank=FALSE)  
ames\_train\_hot <- predict(one\_hot, ames\_train) %>% as.data.frame()  
  
# makr ranger compatible names  
names(ames\_train\_hot) <- make.names(names(ames\_train\_hot), allow\_=FALSE)  
  
#hyperparameter grid search --> same as above but with increased mtry values  
hyper\_grid\_2 <- expand.grid(  
 mtry = seq(50, 200, by = 25),  
 node\_size = seq(3, 9, by = 2),  
 sampe\_size = c(.55, .632, .70, .80),  
 OOB\_RMSE = 0  
)  
  
# perform grid search  
for(i in 1:nrow(hyper\_grid\_2)) {  
   
 # train model  
 model <- ranger(  
 formula = Sale.Price ~ .,   
 data = ames\_train\_hot,   
 num.trees = 500,  
 mtry = hyper\_grid\_2$mtry[i],  
 min.node.size = hyper\_grid\_2$node\_size[i],  
 sample.fraction = hyper\_grid\_2$sampe\_size[i],  
 seed = 123  
 )  
   
 # add OOB error to grid  
 hyper\_grid\_2$OOB\_RMSE[i] <- sqrt(model$prediction.error)  
}  
  
hyper\_grid\_2 %>%   
 dplyr::arrange(OOB\_RMSE) %>%  
 head(10)  
## mtry node\_size sampe\_size OOB\_RMSE  
## 1 100 3 0.8 26758.10  
## 2 100 9 0.8 26831.67  
## 3 100 5 0.8 26835.23  
## 4 175 3 0.8 26846.78  
## 5 100 7 0.8 26849.53  
## 6 75 3 0.8 26870.75  
## 7 75 5 0.8 26881.27  
## 8 175 5 0.8 26995.36  
## 9 150 3 0.8 27008.37  
## 10 150 5 0.8 27021.19  
## mtry node\_size sampe\_size OOB\_RMSE  
## 1 50 3 0.8 26981.17  
## 2 75 3 0.8 27000.85  
## 3 75 5 0.8 27040.55  
## 4 75 7 0.8 27086.80  
## 5 50 5 0.8 27113.23  
## 6 125 3 0.8 27128.26  
## 7 100 3 0.8 27131.08  
## 8 125 5 0.8 27136.93  
## 9 125 3 0.7 27155.03  
## 10 200 3 0.8 27171.37

Currently, the best random forest model we have found retains columnar categorical variables and uses mtry = 24, terminal node size of 5 observations, and a sample size of 80%. Lets repeat this model to get a better expectation of our error rate. We see that our expected error ranges between ~25,800-26,400 with a most likely just shy of 26,200.

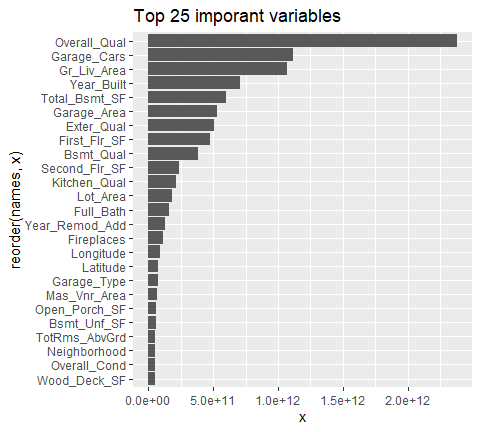
OOB\_RMSE <- vector(mode="numeric", length=100)  
  
for (i in seq\_along(OOB\_RMSE)){  
   
 optimal\_ranger <- ranger(  
 formula = Sale\_Price ~.,  
 data = ames\_train,  
 num.trees = 500,  
 mtry = 24,  
 min.node.size = 5,  
 sample.fraction =.8,  
 importance = "impurity"  
 )  
 OOB\_RMSE[i] <- sqrt(optimal\_ranger$prediction.error)  
}  
  
hist(OOB\_RMSE, breaks = 20)



Furthermore, you may have noticed we set imporance ="impurity in the above modeling, which allows us to assess variable importance. Variable importance is measured by recording the decrease in MSE each time a variable is used as a node split in a tree.

The remaining error left in predictive accuracy after a node split is known as **node impurity** and a variable that reduces this impurity is considered more imporant than those variables that do not. Consequently, we accumulate the reduction in MSE for each variable across all the trees and the variable with the greatest accumulated impact is considered the more important, or impactful. We see that Overall\_Qual has the greatest impact in reducing MSE across our trees, followed by Gr\_Liv\_Area, Garage\_Cars, etc.

optimal\_ranger$variable.importance %>%   
 tidy() %>%   
 dplyr::arrange(desc(x)) %>%   
 dplyr::top\_n(25) %>%   
 ggplot(aes(reorder(names,x), x))+  
 geom\_col()+  
 coord\_flip()+  
 ggtitle("Top 25 imporant variables")



**Full grid search with H2O**

If you ran the grid search code above you probably noticed the code took a while to run. Although ranger is computationally efficient, as the grid search space expands, the manual for loop process becomes less efficient. h2o is a powerful and efficient java-based interface that provides parallel distributed algorithms. Moreover, h2o allows for different optimal search paths in our grid search. This allows us to be more efficient in tuning our models. Here, I demonstrate how to tune a random forest model with h2o. Lets go ahead and start up h2o:

# start up h2o (I turn off progress bars when creating reports/tutorials)  
h2o.no\_progress()  
h2o.init(max\_mem\_size = "5g")

First, we can try a comprehensive ( **full cartesian**) grid search, which means we will examine every combination of hyperparameter settings that we specify in hyper\_grid.h2o. Here, we search across 96 models but since we perform a full cartesian search this process is not any faster than that which we did above. However, note that the best performing model has an OOB RMSE of 24504 (), which is lower than what we achieved previously. This is because some of the default settings regarding minimum node size, tree depth, etc. are more ?ggenerous?h than ranger and randomForest (i.e. h2o has a default minimum node size of one whereas ranger and randomForest default settings are 5).

# create feature names  
y <- "Sale\_Price"  
x <- setdiff(names(ames\_train), y)  
  
# turn training set into h2o object  
train.h2o <- as.h2o(ames\_train)  
  
# hyperparameter grid  
hyper\_grid.h2o <- list(  
 ntrees = seq(200, 500, by = 100),  
 mtries = seq(20, 30, by = 2),  
 sample\_rate = c(.55, .632, .70, .80)  
)  
  
# build grid search  
grid <- h2o.grid(  
 algorithm = "randomForest",  
 grid\_id = "rf\_grid",  
 x = x,  
 y =y,  
 training\_frame = train.h2o,  
 hyper\_params = hyper\_grid.h2o,  
 searchsearch\_criteria = list(strategy = "Cartesian")  
)  
  
# collect the results and sor by our model performance metric of choice  
grid\_perf <- h2o.getGrid(  
 grid\_id = "rf\_grid",  
 sort\_by = "mse",  
 decreasing = FALSE  
)  
  
print(grid\_perf)

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

For example, the following code searches a large ggrid search of 2,025 hyperparameter combinations. e create a random grid search that will stop if none of the last 10 models have managed to have a 0.5% improvement in MSE compared to the best model before that. If we continue to find improvements then I cut the grid search off after 600 seconds (30 minutes). Our grid search assessed 190 models and the best model (max\_depth = 30, min\_rows = 1, mtries = 25, nbins = 30, ntrees = 200, sample\_rate = .8) achived an RMSE of 24686 ().

#hyperparameter grid search criteria  
  
search\_criteria <- list(  
 strategy = "RandomDiscrete",  
 stopping\_metric = "mse",  
 stopping\_tolerance = 0.005,  
 stopping\_rounds = 10,  
 max\_runtime\_secs = 30\*60  
)  
  
# build grid search  
random\_grid <- h2o.grid(  
 algorithm = "randomForest",  
 grid\_id = "rf\_grid2",  
 x = x,  
 y = y,  
 training\_frame = train.h2o,  
 search\_criteria = search\_criteria  
)  
  
# collect the results and sort by our model performance metric of choice  
grid\_perf2 <- h2o.getGrid(  
 grid\_id = "rf\_grid2",   
 sort\_by = "mse",   
 decreasing = FALSE  
 )  
print(grid\_perf2)  
## H2O Grid Details  
## ================  
##   
## Grid ID: rf\_grid2   
## Used hyper parameters:   
## - max\_depth   
## - min\_rows   
## - mtries   
## - nbins   
## - ntrees   
## - sample\_rate   
## Number of models: 190   
## Number of failed models: 0   
##   
## Hyper-Parameter Search Summary: ordered by increasing mse  
## max\_depth min\_rows mtries nbins ntrees sample\_rate model\_ids  
## 1 30 1.0 25 30 200 0.8 rf\_grid2\_model\_114  
## 2 30 1.0 30 30 400 0.8 rf\_grid2\_model\_60  
## 3 25 1.0 20 25 200 0.8 rf\_grid2\_model\_62  
## 4 20 1.0 20 15 400 0.8 rf\_grid2\_model\_48  
## 5 20 1.0 15 15 350 0.75 rf\_grid2\_model\_149  
## mse  
## 1 6.09386451519276E8  
## 2 6.141013192008269E8  
## 3 6.143676001174936E8  
## 4 6.181798579219993E8  
## 5 6.182797259475644E8  
##   
## ---  
## max\_depth min\_rows mtries nbins ntrees sample\_rate model\_ids  
## 185 30 5.0 30 15 500 0.55 rf\_grid2\_model\_126  
## 186 35 5.0 15 10 300 0.55 rf\_grid2\_model\_84  
## 187 25 5.0 15 20 200 0.55 rf\_grid2\_model\_20  
## 188 35 5.0 15 10 200 0.55 rf\_grid2\_model\_184  
## 189 40 1.0 15 20 400 0.55 rf\_grid2\_model\_127  
## 190 30 1.0 25 20 500 0.632 rf\_grid2\_model\_189  
## mse  
## 185 7.474602079646143E8  
## 186 7.530174943920757E8  
## 187 7.591548840980767E8  
## 188 7.721963479865576E8  
## 189 1.0642428537243171E9  
## 190 1.4912496290688899E9

Once we have identifed the best model we can get that model and apply it to our hold-out test set to compute our final test error. We see that we?fve been able to reduce our RMSE to near $23,000, which is a $10K reduction compared to elastic nets and bagging.

#grab the model id for the top model, chosen by validation error  
  
best\_model\_id <- grid\_perf2@model\_ids[[1]]  
best\_model <- h2o.getModel(best\_model\_id)  
  
# Now let?fs evaluate the model performance on a test set  
ames\_test.h2o <- as.h2o(ames\_test)  
best\_model\_perf <- h2o.performance(model = best\_model, newdata = ames\_test.h2o)  
  
# RMSE of best model  
h2o.mse(best\_model\_perf) %>% sqrt()  
## [1] 23104.67

#### Predicting

Once we?fve identified our preferred model we can use the traditional predict function to make predictions on a new data set. We can use this for all our model types (randomForest, ranger, and h2o); although the outputs differ slightly. Also, not that the new data for the h2o model needs to be an h2o object.

# randomForest  
pred\_randomForest <- predict(ames\_randomForest, ames\_test)  
head(pred\_randomForest)  
## 1 2 3 4 5 6   
## 129459.5 185526.7 263271.6 196375.7 176455.2 392007.5  
## 1 2 3 4 5 6   
## 128266.7 153888.0 264044.2 379186.5 212915.1 210611.4  
  
# ranger  
pred\_ranger <- predict(ames\_ranger, ames\_test)  
head(pred\_ranger$predictions)  
## [1] 129130.5 186123.7 269912.0 198751.7 176939.0 395345.4  
## [1] 128440.6 154160.1 266428.5 389959.6 225927.0 214493.1  
  
# h2o - commented  
# pred\_h2o <- predict(best\_model, ames\_test.h2o)  
# head(pred\_h2o)  
## predict  
## 1 126903.1  
## 2 154215.9  
## 3 265242.9  
## 4 381486.6  
## 5 211334.3  
## 6 202046.5

#### Learning more

Random forests provide a very powerful out-of-the-box algorithm that often has great predictive accuracy. Because of their more simplistic tuning nature and the fact that they require very little, if any, feature pre-processing they are often one of the first go-to algorithms when facing a predictive modeling problem. To learn more I would start with the following resources listed in order of complexity:

* An Introduction to Statistical Learning
* Applied Predictive Modeling
* Computer Age Statistical Inference
* The Elements of Statistical Learning