

Regression models:

In-sample RMSE for linear regression on diamonds

```
# Fit lm model: model
model <- lm(price ~ ., diamonds)

# Predict on full data: p
p <- predict(model, diamonds)

# Compute errors: error
error <- p - diamonds[["price"]]

# Calculate RMSE
sqrt(mean(error^2))
```

Randomly order the data frame

One way you can take a train/test split of a dataset is to order the dataset randomly, then divide it into the two sets. This ensures that the training set and test set are both random samples and that any biases in the ordering of the dataset (e.g. if it had originally been ordered by price or size) are not retained in the samples we take for training and testing your models. You can think of this like shuffling a brand new deck of playing cards before dealing hands.

First, you set a random seed so that your work is reproducible and you get the same random split each time you run your script:

```
set.seed(42)
```

Next, you use the `sample()` function to shuffle the row indices of the `diamonds` dataset. You can later use these indices to reorder the dataset.

```
rows <- sample(nrow(diamonds))
```

Finally, you can use this random vector to reorder the diamonds dataset:

```
diamonds <- diamonds[rows, ]
```

```
# Set seed
set.seed(42)

# Shuffle row indices: rows
rows <- sample(nrow(diamonds))

# Randomly order data
diamonds <- diamonds[rows, ]
```

Try an 80/20 split

Now that your dataset is randomly ordered, you can split the first 80% of it into a training set, and the last 20% into a test set. You can do this by choosing a split point approximately 80% of the way through your data:

```
split <- round(nrow(mydata) * .80)
```

You can then use this point to break off the first 80% of the dataset as a training set:

```
mydata[1:split, ]
```

And then you can use that same point to determine the test set:

```
mydata[(split + 1):nrow(mydata), ]
```

```
# Determine row to split on: split
split <- round(nrow(diamonds) * .80)
```

```
# Create train
train <- diamonds[1:split, ]
```

```
# Create test
test <- diamonds[(split + 1):nrow(diamonds), ]
```

Predict on test set

Now that you have a randomly split training set and test set, you can use the `lm()` function as you did in the first exercise to fit a model to your training set, rather than the entire dataset. Recall that you can use the formula interface to the linear regression function to fit a model with a specified target variable using all other variables in the dataset as predictors:

```
mod <- lm(y ~ ., training_data)
```

You can use the `predict()` function to make predictions from that model on new data. The new dataset must have all of the columns from the training data, but they can be in a different order with different values. Here, rather than re-predicting on the training set, you can predict on the test set, which you did not use for training the model. This will allow you to determine the out-of-sample error for the model in the next exercise:

```
p <- predict(model, new_data)
```

```
# Fit lm model on train: model
```

```
model <- lm(price ~ ., train)
```

```
# Predict on test: p
```

```
p <- predict(model, test)
```

Calculate test set RMSE by hand

Now that you have predictions on the test set, you can use these predictions to calculate an error metric (in this case RMSE) on the test set and see how the model performs out-of-sample, rather than in-sample as you did in the first exercise. You first do this by calculating the errors between the predicted diamond prices and the actual diamond prices by subtracting the predictions from the actual values.

Once you have an error vector, calculating RMSE is as simple as squaring it, taking the mean, then taking the square root:

```
sqrt(mean(error^2))

# Compute errors: error
error <- p - test[["price"]]

# Calculate RMSE
sqrt(mean(error^2))
```

Comparing out-of-sample RMSE to in-sample RMSE

Because you overfit the training set and the test set contains data the model hasn't seen before.

Advantage of cross-validation

0-fold cross-validation

As you saw in the video, a better approach to validating models is to use multiple systematic test sets, rather than a single random train/test split. Fortunately, the `caret` package makes this very easy to do:

```
model <- train(y ~ ., my_data)
```

`caret` supports many types of cross-validation, and you can specify which type of cross-validation and the number of cross-validation folds with the `trainControl()` function, which you pass to the `trControl` argument in `train()`:

```
model <- train(
  y ~ ., my_data,
  method = "lm",
  trControl = trainControl(
    method = "cv", number = 10,
    verboseIter = TRUE
  )
)
```

It's important to note that you pass the method for modeling to the main `train()` function and the method for cross-validation to the `trainControl()` function.

Fit lm model using 10-fold CV: model

```
model <- train(
  price ~ ., diamonds,
  method = "lm",
  trControl = trainControl(
    method = "cv", number = 10,
    verboseIter = TRUE
  )
)
```

```
# Print model to console  
model
```

5-fold cross-validation

```
# Fit lm model using 5-fold CV: model  
model <- train(  
  medv ~ ., Boston,  
  method = "lm",  
  trControl = trainControl(  
    method = "cv", number = 5,  
    verboseIter = TRUE  
  )  
)  
# Print model to console  
model.
```

5 x 5-fold cross-validation

You can do more than just one iteration of cross-validation. Repeated cross-validation gives you a better estimate of the test-set error. You can also repeat the entire cross-validation procedure. This takes longer, but gives you many more out-of-sample datasets to look at and much more precise assessments of how well the model performs.

One of the awesome things about the `train()` function in `caret` is how easy it is to run very different models or methods of cross-validation just by tweaking a few simple arguments to the function call. For example, you could repeat your entire cross-validation procedure 5 times for greater confidence in your estimates of the model's out-of-sample accuracy, e.g.:

```
trControl = trainControl(  
  method = "cv", number = 5,
```

```
    repeats = 5, verboseIter = TRUE
  )
```

Fit lm model using 5 x 5-fold CV: model

```
model <- train(
  medv ~ ., Boston,
  method = "lm",
  trControl = trainControl(
    method = "cv", number = 5,
    repeats = 5, verboseIter = TRUE
  )
)
```

Print model to console

Model

Making predictions on new data

Finally, the model you fit with the `train()` function has the exact same `predict()` interface as the linear regression models you fit earlier in this chapter. After fitting a model with `train()`, you can simply call `predict()` with new data, e.g:

```
predict(my_model, new_data)
```

```
predict(model,Boston)
```

Section 2: Logistic regression on sonar

In this chapter, you'll fit classification models with `train ()` and evaluate their out-of-sample performance using cross-validation and area under the curve (AUC).

Try a 60/40 split

Shuffle row indices: rows

```
rows <- sample(nrow(Sonar))

# Randomly order data: Sonar
Sonar <- Sonar[rows, ]

# Identify row to split on: split
split <- round(nrow(Sonar) * .60)

# Create train
train <- Sonar[1:split, ]

# Create test
test <- Sonar[(split + 1):nrow(Sonar), ]
```

Fit a logistic regression model

```
# Fit glm model: model
model <- glm(Class ~ ., family="binomial",train)
```

```
# Predict on test: p
p <- predict(model,test,type="response")
```

Confusion matrix takeaways

```
# Calculate class probabilities: p_class
p_class <-
  ifelse(p > 0.50,
    "M",
    "R"
  )
```

```
# Create confusion matrix  
confusionMatrix(p_class, test$Class)
```

Probabilities and classes

Try another threshold

```
# Apply threshold of 0.9: p_class
```

```
p_class <-  
  ifelse(p > 0.90,  
    "M",  
    "R"  
  )
```

```
# Create confusion matrix  
confusionMatrix(p_class, test$Class)
```

From probabilities to confusion matrix

```
# Apply threshold of 0.10: p_class
```

```
p_class <-  
  ifelse(p > 0.10,  
    "M",  
    "R"  
  )
```

```
# Create confusion matrix  
confusionMatrix(p_class, test$Class)
```


Plot an ROC curve

```
# Predict on test: p
p <- predict(model,test,type="response")

# Make ROC curve
colAUC(p, test$Class, plotROC = TRUE)
```

Model, ROC, and AUC

Customizing trainControl

```
# Create trainControl object: myControl
myControl <- trainControl(
  method = "cv",
  number = 10,
  summaryFunction = twoClassSummary,
  classProbs = TRUE, # IMPORTANT!
  verboseIter = TRUE
)

# Train glm with custom trainControl: model
model <- train(Class ~ .,method = "glm",Sonar, trControl = myControl)

# Print model to console
Model
```

Section 3: Random forests and wine

In this chapter, you will use the `train()` function to tweak model parameters through cross-validation and grid search.

Fit a random forest

```
# Fit random forest: model
```

```
model <- train(  
  quality ~ .,  
  tuneLength = 1,  
  data = wine, method = "ranger",  
  trControl = trainControl(method = "cv", number = 5, verboseIter = TRUE)  
)
```

```
# Print model to console
```

```
Model
```

Explore a wider model space Try a longer tune length

```
# Fit random forest: model
```

```
model <- train(  
  quality ~ .,  
  tuneLength = 3,  
  data = wine, method = "ranger",  
  trControl = trainControl(method = "cv", number = 5, verboseIter = TRUE)  
)
```

```
# Print model to console
```

```
model
```

```
# Plot model
```

```
plot(model)
```

Custom tuning grids

Fit a random forest with custom tuning

```
# Fit random forest: model
```

```
model <- train(
```

```
  quality ~.,
```

```
  tuneGrid = data.frame(mtry = c(2, 3, 7)),
```

```
  data = wine, method = "ranger",
```

```
  trControl = trainControl(method = "cv", number = 5, verboseIter = TRUE)
```

```
)
```

```
# Print model to console
```

```
model
```

```
# Plot model
```

```
plot(model)
```

Introducing glmnet

Advantage of glmnet

glmnet models place constraints on your coefficients, which helps prevent over fitting.

Make a custom trainControl

The wine quality dataset was a regression problem, but now you are looking at a classification problem. This is a simulated dataset based on the "don't overfit" competition on Kaggle a number of years ago.

Classification problems are a little more complicated than regression problems because you have to provide a custom `summaryFunction` to the `train()` function to use the `AUC` metric to rank your models. Start by making a custom `trainControl`, as you did

in the previous chapter. Be sure to set `classProbs = TRUE`, otherwise the `twoClassSummary` for `summaryFunction` will break.

```
# Create custom trainControl: myControl
myControl <- trainControl(
  method = "cv", number = 10,
  summaryFunction = twoClassSummary,
  classProbs = TRUE, # IMPORTANT!
  verboseIter = TRUE
)
```

Fit glmnet with custom trainControl

Now that you have a custom `trainControl` object, fit a `glmnet` model to the "don't overfit" dataset. Recall from the video that `glmnet` is an extension of the generalized linear regression model (or `glm`) that places constraints on the magnitude of the coefficients to prevent overfitting. This is more commonly known as "penalized" regression modeling and is a very useful technique on datasets with many predictors and few values. `glmnet` is capable of fitting two different kinds of penalized models, controlled by the `alpha` parameter:

- Ridge regression (or `alpha = 0`)
- Lasso regression (or `alpha = 1`)

You'll now fit a `glmnet` model to the "don't overfit" dataset using the defaults provided by the `caret` package.

```
# Fit glmnet model: model
model <- train(
  y ~ ., overfit,
  method = "glmnet",
```

```
trControl = myControl
)
```

```
# Print model to console
```

```
model
```

```
# Print maximum ROC statistic
```

```
max(model[["results"]])
```

glmnet with custom tuning grid

Why a custom tuning grid?

The default tuning grid is very small and there are many more potential `glmnet` models you want to explore.

glmnet with custom trainControl and tuning

As you saw in the video, the `glmnet` model actually fits many models at once (one of the great things about the package). You can exploit this by passing a large number of `lambda` values, which control the amount of penalization in the model. `train()` is smart enough to only fit one model per `alpha` value and pass all of the `lambda` values at once for simultaneous fitting.

My favorite tuning grid for `glmnet` models is:

```
expand.grid(alpha = 0:1,
  lambda = seq(0.0001, 1, length = 100))
```

This grid explores a large number of `lambda` values (100, in fact), from a very small one to a very large one. (You could increase the maximum `lambda` to 10, but in this exercise 1 is a good upper bound.)

If you want to explore fewer models, you can use a shorter `lambda` sequence. For example, `lambda = seq(0.0001, 1, length = 10)` would fit 10 models per value of `alpha`.

You also look at the two forms of penalized models with this `tuneGrid`: ridge regression and lasso regression. `alpha = 0` is pure ridge regression, and `alpha = 1` is pure lasso regression. You can fit a mixture of the two models (i.e. an elastic net) using

an `alpha` between 0 and 1. For example, `alpha = .05` would be 95% ridge regression and 5% lasso regression.

In this problem you'll just explore the 2 extremes--pure ridge and pure lasso regression--for the purpose of illustrating their differences.

```
# Train glmnet with custom trainControl and tuning: model
```

```
model <- train(  
  y ~ ., overfit,  
  tuneGrid = expand.grid(alpha = 0:1,  
                          lambda = seq(0.0001, 1, length = 20)),  
  method = "glmnet",  
  trControl = myControl  
)
```

```
# Print model to console
```

```
model
```

```
# Print maximum ROC statistic
```

```
max(model[["results"]][["ROC"]])
```

Section 4: Preprocessing your data

In this chapter, you will practice using `train()` to preprocess data before fitting models, improving your ability to making accurate predictions.

Median imputation

Median imputation vs. omitting rows

What's the value of median imputation?

It lets you model data with missing values.

Apply median imputation

In this chapter, you'll be using a version of the Wisconsin Breast Cancer dataset. This dataset presents a classic binary classification problem: 50% of the samples are benign, 50% are malignant, and the challenge is to identify which are which.

This dataset is interesting because many of the predictors contain missing values and most rows of the dataset have at least one missing value. This presents a modeling challenge, because most machine learning algorithms cannot handle missing values out of the box. For example, your first instinct might be to fit a logistic regression model to this data, but prior to doing this you need a strategy for handling the `NA`s.

Fortunately, the `train()` function in `caret` contains an argument called `preProcess`, which allows you to specify that median imputation should be used to fill in the missing values. In previous chapters, you created models with the `train()` function using formulas such as `y ~ .`. An alternative way is to specify the `x` and `y` arguments to `train()`, where `x` is an object with samples in rows and features in columns and `y` is a numeric or factor vector containing the outcomes. Said differently, `x` is a matrix or data frame that contains the whole dataset you'd use for the `data` argument to the `lm()` call, for example, but excludes the response variable column; `y` is a vector that contains just the response variable column.

For this exercise, the argument `x` to `train()` is loaded in your workspace as `breast_cancer_x` and `y` as `breast_cancer_y`.

```
# Apply median imputation: model

model <- train(

  x = breast_cancer_x, y = breast_cancer_y,

  method = "glm",

  trControl = myControl,

  preProcess = "medianImpute"

)
```

```
# Print model to console

Model
```

KNN imputation

Will KNN imputation always be better than median imputation?

No, you should try both options and keep the one that gives more accurate models.

Use KNN imputation

```
# Apply KNN imputation: model2

model2 <- train(
  x = breast_cancer_x, y = breast_cancer_y,
  method = "glm",
  trControl = myControl,
  preProcess = "knnImpute"
)
```

```
# Print model to console
```

```
model2
```

Compare KNN and median imputation

All of the preprocessing steps in the `train()` function happen in the training set of each cross-validation fold, so the error metrics reported include the effects of the preprocessing.

This includes the imputation method used (e.g. `knnImpute` or `medianImpute`). This is useful because it allows you to compare different methods of imputation and choose the one that performs the best out-of-sample.

`median_model` and `knn_model` are available in your workspace, as is `resamples`, which contains the resampled results of both models. Look at the results of the models by calling

```
dotplot(resamples, metric = "ROC")
```

and choose the one that performs the best out-of-sample. Which method of imputation yields the highest out-of-sample ROC score for your `glm` model?

Multiple preprocessing methods

Order of operations

Which comes first in `caret`'s `preProcess()` function: median imputation or centering and scaling of variables?

Median imputation comes before centering and scaling.

Combining preprocessing methods

The `preProcess` argument to `train()` doesn't just limit you to imputing missing values. It also includes a wide variety of other `preProcess` techniques to make your life as a data scientist much easier. You can read a full list of them by typing `?preProcess` and reading the help page for this function.

One set of preprocessing functions that is particularly useful for fitting regression models is standardization: centering and scaling. You first *center* by subtracting the mean of each column from each value in that column, then you *scale* by dividing by the standard deviation.

Standardization transforms your data such that for each column, the mean is 0 and the standard deviation is 1. This makes it easier for regression models to find a good solution.

```
# Fit glm with median imputation: model1
model1 <- train(
  x = breast_cancer_x, y = breast_cancer_y,
  method = "glm",
  trControl = myControl,
  preProcess = "medianImpute"
)
```

```
# Print model1
```

```
model1
```

```
# Fit glm with median imputation and standardization: model2
```

```
model2 <- train(
```

```
  x = breast_cancer_x, y = breast_cancer_y,
```

```
  method = "glm",
```

```
  trControl = myControl,
```

```
  preProcess = c("medianImpute", "center", "scale")
```

```
)
```

```
# Print model2
```

```
model2
```

Handling low-information predictors

Why remove near zero variance predictors?

What's the best reason to remove near zero variance predictors from your data before building a model?

Remove near zero variance predictors

As you saw in the video, for the next set of exercises, you'll be using the blood-brain dataset. This is a biochemical dataset in which the task is to predict the following value for a set of biochemical compounds:

```
log((concentration of compound in brain) /  
    (concentration of compound in blood))
```

This gives a quantitative metric of the compound's ability to cross the blood-brain barrier, and is useful for understanding the biological properties of that barrier.

One interesting aspect of this dataset is that it contains many variables and many of these variables have extremely low variances. This means that there is very little information in these variables because they mostly consist of a single value (e.g. zero).

Fortunately, `caret` contains a utility function called `nearZeroVar()` for removing such variables to save time during modeling.

`nearZeroVar()` takes in data `x`, then looks at the ratio of the most common value to the second most common value, `freqCut`, and the percentage of distinct values out of the number of total samples, `uniqueCut`. By default, `caret` uses `freqCut = 19` and `uniqueCut = 10`, which is fairly conservative. I like to be a little more aggressive and use `freqCut = 2` and `uniqueCut = 20` when calling `nearZeroVar()`.

```
# Identify near zero variance predictors: remove_cols
```

```
remove_cols <- nearZeroVar(bloodbrain_x, names = TRUE,  
                           freqCut = 2, uniqueCut = 20)
```

```
# Get all column names from bloodbrain_x: all_cols
```

```
all_cols <- names(bloodbrain_x)
```

```
# Remove from data: bloodbrain_x_small
```

```
bloodbrain_x_small <- bloodbrain_x[, setdiff(all_cols, remove_cols)]
```

preProcess() and nearZeroVar()

Can you use the `preProcess` argument in `caret` to remove near-zero variance predictors? Or do you have to do this by hand, prior to modeling, using the `nearZeroVar()` function?

Yes! Set the `preProcess` argument equal to `"nzv"`.

Fit model on reduced blood-brain data

```
# Fit model on reduced data: model

model <- train(x = bloodbrain_x_small, y = bloodbrain_y, method = "glm")


# Print model to console

Model
```

Principle components analysis (PCA)

Using PCA as an alternative to nearZeroVar()

An alternative to removing low-variance predictors is to run PCA on your dataset. This is sometimes preferable because it does not throw out all of your data: many different low variance predictors may end up combined into one high variance PCA variable, which might have a positive impact on your model's accuracy.

This is an especially good trick for linear models: the `pca` option in the `preProcess` argument will center and scale your data, combine low variance variables, and ensure that all of your predictors are orthogonal. This creates an ideal dataset for linear regression modeling, and can often improve the accuracy of your models.

```
# Fit glm model using PCA: model

model <- train(

  x = bloodbrain_x, y = bloodbrain_y,

  method = "glm", preProcess = "pca"

)


# Print model to console

model
```

Section 5: Selecting models

In the final chapter of this course, you'll learn how to use `resamples ()` to compare multiple models and select (or ensemble) the best one(s).

Reusing a trainControl

Example: customer churn data

Summarize the target variables



```
> library(caret)
```

```
> library(C50) #Data is available in this library
```

```
> data(churn)
```

```
> table(churnTrain$churn) / nrow(churnTrain)
```

Why reuse a trainControl?

- So you can use the same `summaryFunction` and tuning parameters for multiple models.¹
-  So you don't have to repeat code when fitting multiple models.²
-  So you can compare models on the exact same training and test data.

Make custom train/test indices

As you saw in the video, for this chapter you will focus on a real-world dataset that brings together all of the concepts discussed in the previous chapters.

The churn dataset contains data on a variety of telecom customers and the modeling challenge is to predict which customers will cancel their service (or churn).

In this chapter, you will be exploring two different types of predictive models: `glmnet` and `rf`, so the first order of business is to create a reusable `trainControl` object you can use to reliably compare them.

Create custom indices: myFolds

```
myFolds <- createFolds(churn_y, k = 5)
```

Create reusable trainControl object: myControl

```
myControl <- trainControl(
```

```
  summaryFunction = twoClassSummary,
```

```
  classProbs = TRUE, # IMPORTANT!
```

```
  verboseIter = TRUE,
```

```
savePredictions = TRUE,  
index = myFolds  
)
```

Reintroduce glmnet

What makes `glmnet` a good baseline model?

It's simple, fast, and easy to interpret.

Fit the baseline model

Now that you have a reusable `trainControl` object called `myControl`, you can start fitting different predictive models to your churn dataset and evaluate their predictive accuracy. You'll start with one of my favorite models, `glmnet`, which penalizes linear and logistic regression models on the size and number of coefficients to help prevent overfitting.

```
# Fit glmnet model: model_glmnet
```

```
model_glmnet <- train(  
  x = churn_x, y = churn_y,  
  metric = "ROC",  
  method = "glmnet",  
  trControl = myControl  
)
```

```
plot(model_glmnet)
```

Reintroduce random forest

Random forest drawback

You no longer have model coefficients to help interpret the model.

Random forest with custom `trainControl`

Another one of my favorite models is the random forest, which combines an ensemble of non-linear decision trees into a highly flexible (and usually quite accurate) model.

Rather than using the classic `randomForest` package, you'll be using the `ranger` package, which is a re-implementation of `randomForest` that produces almost the exact same results, but is faster, more stable, and uses less memory. I highly recommend it as a starting point for random forest modeling in R.

```
# Fit random forest: model_rf
```

```
model_rf <- train(  
  x = churn_x , y = churn_y ,  
  metric = "ROC",  
  method = "ranger",  
  trControl = myControl  
)
```

```
plot(model_rf)
```

Comparing models

Create a resamples object

```
# Create model_list
```

```
model_list <- list(item1 = model_glmnet, item2 = model_rf)
```

```
# Pass model_list to resamples(): resamples
```

```
resamples <- resamples(model_list)
```

```
resamples
```

```
# Summarize the results
```

```
summary(resamples)
```

Create a box-and-whisker plot

`caret` provides a variety of methods to use for comparing models. All of these methods are based on the `resamples()` function. My favorite is the box-and-whisker plot, which allows you to compare the distribution of predictive accuracy (in this case AUC) for the two models.

In general, you want the model with the higher median AUC, as well as a smaller range between min and max AUC.

You can make this plot using the `bwplot()` function, which makes a box and whisker plot of the model's out of sample scores. Box and whisker plots show the median of each distribution as a line and the interquartile range of each distribution as a box around the median line. You can pass the `metric = "ROC"` argument to the `bwplot()` function to show a plot of the model's out-of-sample ROC scores and choose the model with the highest median ROC.

If you do not specify a metric to plot, `bwplot()` will automatically plot 3 of them.

```
# Create bwplot
```

```
bwplot(resamples , metric = "ROC")
```

Create a scatterplot

Another useful plot for comparing models is the scatterplot, also known as the xy-plot. This plot shows you how similar the two models' performances are on different folds.

It's particularly useful for identifying if one model is consistently better than the other across all folds, or if there are situations when the inferior model produces better predictions on a particular subset of the data.

```
# Create xyplot
```

```
xyplot(resamples , metric = "ROC")
```

Summary

What you've learned

- How to use the caret package

- Model fi\$ing and evaluation
- Parameter tuning for be\$er results
- Data preprocessing