Machine Learning Toolbox

Audrey Holloman

Last compiled: Mar 25, 2019

Contents

1 Prerequisites **5**

2 Regression models: fitting them and evaluating their performance **7**
2.1 In-sample RMSE for linear regression on `diamonds` . . . 7
2.2 Randomly order the data frame . . . 8
2.3 Try an 80/20 split . . . 9
2.4 Predict on test set . . . 10
2.5 Calculate test set RMSE . . . 11
2.6 10-fold cross-validation . . . 12
2.7 5-fold cross-validation . . . 13
2.8 5×5-fold cross-validation . . . 15
2.9 Making predictions on new data . . . 17

3 Classification models: fitting them and evaluating their performance **19**
3.1 Try a 60/40 split . . . 19
3.2 Fit a logistic regression model . . . 20
3.3 Calculate a confusion matrix . . . 21
3.4 Try another threshold . . . 24
3.5 From probabilites to confusion matrix . . . 25
3.6 Plot an ROC curve . . . 27
3.7 Customizing `trainControl` . . . 28
3.8 Using custom `trainControl` . . . 29

4 Tuning model parameters to improve performance **31**
4.1 Fit a random forest . . . 31
4.2 Try a longer tune length . . . 33
4.3 Fit a random forest with custom tuning . . . 35
4.4 Make a custom `trainControl` . . . 38
4.5 Fit glmnet with custom `trainControl` . . . 38
4.6 `glmnet` with custom `trainControl` and tuning . . . 40
4.7 Interpreting `glmnet` plots . . . 42

5 Preprocessing your data **43**
5.1 Apply median imputation . . . 43
5.2 Use KNN imputation . . . 45
5.3 Combining preprocessing methods . . . 47
5.4 Remove near zero variance predictors . . . 49
5.5 Fit model on reduced blood-brain data . . . 50
5.6 Using PCA as an alternative to nearZeroVar() . . . 50

6 Selecting models: a case study in churn prediction **53**
6.1 Make custom train/test indices . . . 53

6.2 Fit the baseline model 54
6.3 Random forest with custom `trainControl` 55
6.4 Create a resamples object 56
6.5 Create a box-and-whisker plot 57
6.6 Create a scatterplot 58
6.7 Ensembling models 58

Chapter 1

Prerequisites

This material is from the DataCamp course Machine Learning Toolbox by Zachary Deane-Mayer and Max Kuhn. Before using this material, the reader should have completed and be comfortable with the material in the DataCamp modules Introduction to R, Intermediate R, and Correlation and Regression.

Reminder to self: each `*.Rmd` file contains one and only one chapter, and a chapter is defined by the first-level heading `#`.

Chapter 2

Regression models: fitting them and evaluating their performance

In the first chapter of this course, you'll fit regression models with `train()` and evaluate their out-of-sample performance using cross-validation and root-mean-square error (RMSE).

Welcome to the Toolbox Video

In-sample RMSE for linear regression

RMSE is commonly calculated in-sample on your training set. What's a potential drawback to calculating training set error?

- There's no potential drawback to calculating training set error, but you should calculate R^2 instead of RMSE.
- **You have no idea how well your model generalizes to new data (i.e. overfitting).**
- You should manually inspect your model to validate its coefficients and calculate RMSE.

2.1 In-sample RMSE for linear regression on `diamonds`

`diamonds` is a classic dataset from the `ggplot2` package written by Wickham et al. (2018). The dataset contains physical attributes of diamonds as well as the price they sold for. One interesting modeling challenge is predicting diamond price based on their attributes using something like a linear regression.

Recall that to fit a linear regression, you use the `lm()` function in the following format:

```
mod <- lm(y ~ x, data = my_data)
```

To make predictions using `mod` on the original data, you call the `predict()` function:

```
pred <- predict(mod, newdata = my_data)
```

Exercise

- Fit a linear model on the diamonds dataset predicting price using all other variables as predictors (i.e. `price ~ .`). Save the result to `model`.

```
library(ggplot2)
# Fit lm model: model
model <- lm(price ~ ., data = diamonds)
```

- Make predictions using `model` on the full original dataset and save the result to `p`.

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

- Compute errors using the formula `errors = actual - predicted`. Save the result to `error`.

```
# Compute errors: error
error <- diamonds$price - p
```

- Compute RMSE and print it to the console.

```
# Compute RMSE
RMSE <- sqrt(mean(error^2))
RMSE
```

```
[1] 1129.843
```

Out-of-sample error measures video

Out-of-sample RMSE for linear regression

What is the advantage of using a train/test split rather than just validating your model in-sample on the training set?

- It takes less time to calculate error on the test set, since it is smaller than the training set.
- There is no advantage to using a test set. You can just use adjusted R^2 on your training set.
- **It gives you an estimate of how well your model performs on new data.**

2.2 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 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, ]
```

Exercise

- Set the random seed to 42.

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

- Make a vector of row indices called `rows`.

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

- Randomly reorder the `diamonds` data frame.

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

2.3 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) * 0.80)
```

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), ]
```

Exercise

- Choose a row index to split on so that the split point is approximately 80% of the way through the `diamonds` dataset. Call this index `split`.

```
# Determine row to split on: split
split <- round(nrow(diamonds)*0.80)
```

- Create a training set called `train` using that index.

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

- Create a test set called `test` using that index.

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

2.4 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 ~ ., data = 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, newdata = new_data)
```

Exercise

- Fit an `lm()` model called `model` to predict price using all other variables as covariates. Be sure to use the training set, `train`.

```
# Fit lm model on train: model
model <- lm(price ~ . , data = train)
```

- Predict on the test set, test, using `predict()`. Store these values in a vector called `p`.

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

2.5 Calculate test set RMSE

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))
```

Exercise

- Calculate the error between the predictions on the test set and the actual diamond prices in the test set. Call this `error`.

```
# Compute errors: error
error <- test$price - p
```

- Calculate RMSE using this error vector, just printing the result to the console.

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

```
[1] 1136.596
```

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

Why is the test set RMSE higher than the training set RMSE?

- **Because you overfit the training set and the test set contains data the model hasn't seen before.**
- Because you should not use a test set at all and instead just look at error on the training set.
- Because the test set has a smaller sample size the training set and thus the mean error is lower.

Cross Valdiation Video

Advantage of cross-validation

What is the advantage of cross-validation over a single train/test split?

- There is no advantage to cross-validation, just as there is no advantage to a single train/test split. You should be validating your models in-sample with a metric like adjusted R^2.
- You can pick the best test set to minimize the reported RMSE of your model.

- **It gives you multiple estimates of out-of-sample error, rather than a single estimate.**

Note: If all of your estimates give similar outputs, you can be more certain of the model's accuracy. If your estimates give different outputs, that tells you the model does not perform consistently and suggests a problem with it.

2.6 10-fold cross-validation

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 written by from Jed Wing et al. (2018) 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 is 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.

Exercise

- Load the `caret` package.

```
# Load the caret package
library(caret)
```

- Fit a linear regression to model price using all other variables in the `diamonds` dataset as predictors. Use the `train()` function and 10-fold cross-validation.

```
# Fit lm model using 10-fold CV: model
model <- train(
  price ~ ., data = diamonds,
  method = "lm",
  trControl = trainControl(
    method = "cv", number = 10,
    verboseIter = FALSE
  )
)
```

- Print the model to the console and examine the results.

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

```
Linear Regression

53940 samples
    9 predictor

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 48547, 48546, 48546, 48545, 48545, 48545, ...
Resampling results:

  RMSE      Rsquared   MAE
  1130.658  0.9197492  740.4646

Tuning parameter 'intercept' was held constant at a value of TRUE
```

```
model$finalModel  # show model coefficients
```

```
Call:
lm(formula = .outcome ~ ., data = dat)

Coefficients:
(Intercept)        carat        cut.L        cut.Q        cut.C
   5753.762    11256.978      584.457     -301.908      148.035
    `cut^4`      color.L      color.Q      color.C    `color^4`
    -20.794    -1952.160     -672.054     -165.283       38.195
  `color^5`    `color^6`    clarity.L    clarity.Q    clarity.C
    -95.793      -48.466     4097.431    -1925.004      982.205
`clarity^4`  `clarity^5`  `clarity^6`  `clarity^7`        depth
   -364.918      233.563        6.883       90.640      -63.806
      table            x            y            z
    -26.474    -1008.261        9.609      -50.119
```

```
# summary(model)  # to see all
```

2.7 5-fold cross-validation

In this tutorial, you will use a wide variety of datasets to explore the full flexibility of the `caret` package. Here, you will use the famous `Boston` housing dataset, where the goal is to predict median home values in various Boston suburbs.

You can use exactly the same code as in the previous exercise, but change the dataset used by the model:

```
model <- train(
  medv ~ ., Boston,
  method = "lm",
  trControl = trainControl(
    method = "cv", number = 10,
    verboseIter = FALSE
  )
)
```

Next, you can reduce the number of cross-validation folds from 10 to 5 using the number argument to the

`trainControl()` argument:

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

Exercise

- Load the `MASS` package.

```
# Load the MASS pacakge
library(MASS)
```

- Fit an `lm()` model to the `Boston` housing dataset, such that `medv` is the response variable and all other variables are explanatory variables. Use 5-fold cross-validation rather than 10-fold cross-validation.

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

- Print the model to the console and inspect the results.

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

```
Linear Regression

506 samples
 13 predictor

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 405, 403, 405, 406, 405
Resampling results:

  RMSE      Rsquared   MAE
  4.794707  0.7290369  3.372915

Tuning parameter 'intercept' was held constant at a value of TRUE
```

```
# show coefficients of model
model$finalModel
```

```
Call:
lm(formula = .outcome ~ ., data = dat)

Coefficients:
(Intercept)         crim           zn        indus         chas
```

```
  3.646e+01   -1.080e-01    4.642e-02    2.056e-02    2.687e+00
        nox           rm          age          dis          rad
 -1.777e+01    3.810e+00    6.922e-04   -1.476e+00    3.060e-01
        tax      ptratio        black        lstat
 -1.233e-02   -9.527e-01    9.312e-03   -5.248e-01
```

```
summary(model)
```

```
Call:
lm(formula = .outcome ~ ., data = dat)

Residuals:
    Min      1Q  Median      3Q     Max
-15.595  -2.730  -0.518   1.777  26.199

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.646e+01  5.103e+00   7.144 3.28e-12 ***
crim        -1.080e-01  3.286e-02  -3.287 0.001087 **
zn           4.642e-02  1.373e-02   3.382 0.000778 ***
indus        2.056e-02  6.150e-02   0.334 0.738288
chas         2.687e+00  8.616e-01   3.118 0.001925 **
nox         -1.777e+01  3.820e+00  -4.651 4.25e-06 ***
rm           3.810e+00  4.179e-01   9.116  < 2e-16 ***
age          6.922e-04  1.321e-02   0.052 0.958229
dis         -1.476e+00  1.995e-01  -7.398 6.01e-13 ***
rad          3.060e-01  6.635e-02   4.613 5.07e-06 ***
tax         -1.233e-02  3.760e-03  -3.280 0.001112 **
ptratio     -9.527e-01  1.308e-01  -7.283 1.31e-12 ***
black        9.312e-03  2.686e-03   3.467 0.000573 ***
lstat       -5.248e-01  5.072e-02 -10.347  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4.745 on 492 degrees of freedom
Multiple R-squared:  0.7406,    Adjusted R-squared:  0.7338
F-statistic: 108.1 on 13 and 492 DF,  p-value: < 2.2e-16
```

2.8 5 × 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 = "repeatedcv", number = 5,
  repeats = 5, verboseIter = TRUE
)
```

Exercise

- Re-fit the linear regression model to the `Boston` housing dataset. Use 5 repeats of 5-fold cross-validation.

```
# Fit lm model using 5 x 5-fold CV: model
model <- train(
  medv ~ ., data = Boston,
  method = "lm",
  trControl = trainControl(
    method = "repeatedcv", number = 5,
    repeats = 5, verboseIter = FALSE
  )
)
```

- Print the model to the console.

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

```
Linear Regression

506 samples
 13 predictor

No pre-processing
Resampling: Cross-Validated (5 fold, repeated 5 times)
Summary of sample sizes: 405, 405, 404, 405, 405, 406, ...
Resampling results:

  RMSE      Rsquared   MAE
  4.870744  0.7259058  3.400369

Tuning parameter 'intercept' was held constant at a value of TRUE
```

```
summary(model)
```

```
Call:
lm(formula = .outcome ~ ., data = dat)

Residuals:
    Min      1Q  Median      3Q     Max
-15.595  -2.730  -0.518   1.777  26.199

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  3.646e+01  5.103e+00   7.144 3.28e-12 ***
crim        -1.080e-01  3.286e-02  -3.287 0.001087 **
```

```
zn           4.642e-02  1.373e-02   3.382 0.000778 ***
indus        2.056e-02  6.150e-02   0.334 0.738288
chas         2.687e+00  8.616e-01   3.118 0.001925 **
nox         -1.777e+01  3.820e+00  -4.651 4.25e-06 ***
rm           3.810e+00  4.179e-01   9.116  < 2e-16 ***
age          6.922e-04  1.321e-02   0.052 0.958229
dis         -1.476e+00  1.995e-01  -7.398 6.01e-13 ***
rad          3.060e-01  6.635e-02   4.613 5.07e-06 ***
tax         -1.233e-02  3.760e-03  -3.280 0.001112 **
ptratio     -9.527e-01  1.308e-01  -7.283 1.31e-12 ***
black        9.312e-03  2.686e-03   3.467 0.000573 ***
lstat       -5.248e-01  5.072e-02 -10.347  < 2e-16 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 4.745 on 492 degrees of freedom
Multiple R-squared:  0.7406,    Adjusted R-squared:  0.7338
F-statistic: 108.1 on 13 and 492 DF,  p-value: < 2.2e-16
```

2.9 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.

After fitting a model with `train()`, you can call `predict()` with new data, e.g:

```
predict(my_model, newdata = new_data)
```

Exercise

- Use the `predict()` function to make predictions with `model` on the full `Boston` housing dataset. Print the result to the console.

```
# Predict on full Boston dataset
head(predict(model, newdata = Boston))
```

```
       1        2        3        4        5        6
30.00384 25.02556 30.56760 28.60704 27.94352 25.25628
```

```
tail(predict(model, newdata = Boston))
```

```
     501      502      503      504      505      506
20.46871 23.53334 22.37572 27.62743 26.12797 22.34421
```

Chapter 3

Classification models: fitting them and evaluating their performance

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).

Logistic regression on sonar video

Why a train/test split?

What is the point of making a train/test split for binary classification problems?

- To make the problem harder for the model by reducing the dataset size.
- **To evaluate your models out-of-sample, on new data.**
- To reduce the dataset size, so your models fit faster.
- There is no real reason; it is no different than evaluating your models in-sample.

3.1 Try a 60/40 split

As you saw in the video, you'll be working with the `Sonar` dataset in this chapter, using a 60% training set and a 40% test set. We'll practice making a train/test split one more time, just to be sure you have the hang of it. Recall that you can use the `sample()` function to get a random permutation of the row indices in a dataset, to use when making train/test splits, e.g.:

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

And then use those row indices to randomly reorder the dataset, e.g.:

```
my_data <- my_data[rows, ]
```

Once your dataset is randomly ordered, you can split off the first 60% as a training set and the last 40% as a test set.

Exercise

- Shuffle the row indices of `Sonar` and store the result in `rows`.

```
library(mlbench)
data(Sonar)
# Shuffle row indices: rows
set.seed(421)
rows <- sample(nrow(Sonar))
```

- Use `rows` to randomly reorder the rows of `Sonar`.

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

- Identify the proper row to split on for a 60/40 split. Store this row number as `split`.

```
# Identify row to split on: split
split <- round(nrow(Sonar)*.60, 0)
split
```

```
[1] 125
```

- Save the first 60% as a training set.

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

- Save the last 40% as the test set.

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

3.2 Fit a logistic regression model

Once you have your random training and test sets you can fit a logistic regression model to your training set using the `glm()` function. `glm()` is a more advanced version of `lm()` that allows for more varied types of regression models, aside from plain vanilla ordinary least squares regression.

Be sure to pass the argument `family = "binomial"` to `glm()` to specify that you want to do logistic (rather than linear) regression. For example:

```
glm(Target ~ ., family = "binomial", dataset)
```

Don't worry about warnings like

```
glm.fit: algorithm did not converge or glm.fit: fitted probabilities numerically 0 or 1 occurred
```

These are common on smaller datasets and usually don't cause any issues. They typically mean your dataset is perfectly separable, which can cause problems for the math behind the model, but R's `glm()` function is almost always robust enough to handle this case with no problems.

Once you have a `glm()` model fit to your dataset, you can predict the outcome (e.g. rock or mine) on the test set using the `predict()` function with the argument `type = "response"`:

```
predict(my_model, test, type = "response")
```

Exercise

- Fit a logistic regression called `model` to predict `Class` using all other variables as predictors. Use the training set for `Sonar`.

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

```
Warning: glm.fit: algorithm did not converge
```

```
Warning: glm.fit: fitted probabilities numerically 0 or 1 occurred
```

- Predict on the `test` set using that model. Call the result `p` like you've done before.

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

Confusion matrix video

Confusion Matrix

See https://en.wikipedia.org/wiki/Confusion_matrix for a table and formulas.

Confusion matrix takeaways

What information does a confusion matrix provide?

- True positive rates
- True negative rates
- False positive rates
- False negative rates
- **All of the above**

3.3 Calculate a confusion matrix

As you saw in the video, a confusion matrix is a very useful tool for calibrating the output of a model and examining all possible outcomes of your predictions (true positive, true negative, false positive, false negative).

Before you make your confusion matrix, you need to "cut" your predicted probabilities at a given threshold to turn probabilities into a factor of class predictions. Combine `ifelse()` with `factor()` as follows:

```
pos_or_neg <- ifelse(probability_prediction > threshold, positive_class, negative_class)
p_class <- factor(pos_or_neg, levels = levels(test_values))
```

`confusionMatrix()` in caret improves on `table()` from base `R` by adding lots of useful ancillary statistics in addition to the base rates in the table. You can calculate the confusion matrix (and the associated statistics) using the predicted outcomes as well as the actual outcomes, e.g.:

```
confusionMatrix(p_class, test_values)
```

Exercise

- Use `ifelse()` to create a character vector, `m_or_r` that is the positive class, `"M"`, when `p` is greater than 0.5, and the negative class, `"R"`, otherwise.

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

- Convert `m_or_r` to be a factor, `p_class`, with levels the same as those of `test[["Class"]]`.

```
p_class <- factor(m_or_r, levels = levels(test[["Class"]]))
# OR
p_class <- factor(m_or_r, levels = c("M", "R"))
```

- Make a confusion matrix with `confusionMatrix()`, passing `p_class` and the `"Class"` column from the `test` dataset.

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

```
Confusion Matrix and Statistics

          Reference
Prediction  M  R
         M 11 29
         R 33 10

               Accuracy : 0.253
                 95% CI : (0.1639, 0.3604)
    No Information Rate : 0.5301
    P-Value [Acc > NIR] : 1.0000

                  Kappa : -0.4907
 Mcnemar's Test P-Value : 0.7032

            Sensitivity : 0.2500
            Specificity : 0.2564
         Pos Pred Value : 0.2750
         Neg Pred Value : 0.2326
             Prevalence : 0.5301
         Detection Rate : 0.1325
   Detection Prevalence : 0.4819
```

```
      Balanced Accuracy : 0.2532

       'Positive' Class : M
```

```
# Using table()
table(p_class, test$Class)
```

```
p_class  M  R
      M 11 29
      R 33 10
```

```
# Using xtabs()
xtabs(~p_class + test$Class)
```

```
       test$Class
p_class  M  R
      M 11 29
      R 33 10
```

Exercise

Calculating accuracy—Use `confusionMatrix(p_class, test[["Class"]])` to calculate a confusion matrix on the test set.

- What is the test set accuracy of this model (rounded to the nearest percent)?

```
RES <- caret::confusionMatrix(p_class, test[["Class"]])
RES
```

```
Confusion Matrix and Statistics

          Reference
Prediction  M  R
         M 11 29
         R 33 10

               Accuracy : 0.253
                 95% CI : (0.1639, 0.3604)
    No Information Rate : 0.5301
    P-Value [Acc > NIR] : 1.0000

                  Kappa : -0.4907
 Mcnemar's Test P-Value : 0.7032

            Sensitivity : 0.2500
            Specificity : 0.2564
         Pos Pred Value : 0.2750
         Neg Pred Value : 0.2326
             Prevalence : 0.5301
         Detection Rate : 0.1325
   Detection Prevalence : 0.4819
      Balanced Accuracy : 0.2532
```

```
       'Positive' Class : M
```

```
RES$overall[1]
```

```
Accuracy
0.253012
```

The accuracy of this model is 25.3%.

- What is the test set true positive rate (or sensitivity) of this model (rounded to the nearest percent)?

```
Sens <- round(RES[[4]]["Sensitivity"]*100, 1)
Sens
```

```
Sensitivity
         25
```

The test set sensitivity of this model is 25%.

- What is the test set true negative rate (or specificity) of this model (rounded to the nearest percent)?

```
Spec <- round(RES[[4]]["Specificity"]*100, 1)
Spec
```

```
Specificity
       25.6
```

The test set specificity of this model is 25.6%.

Class probabilities and predictions video

Exercise

Probabilities and classes—What's the relationship between the predicted probabilities and the predicted classes?

- You determine the predicted probabilities by looking at the average accuracy of the predicted classes.
- There is no relationship; they're completely different things.
- **Predicted classes are based off of predicted probabilities plus a classification threshold.**

3.4 Try another threshold

In the previous exercises, you used a threshold of 0.50 to cut your predicted probabilities to make class predictions (rock vs mine). However, this classification threshold does not always align with the goals for a given modeling problem.

For example, pretend you want to identify the objects you are really certain are mines. In this case, you might want to use a probability threshold of 0.90 to get fewer predicted mines, but with greater confidence in each prediction.

- Use `ifelse()` to create a character vector, `m_or_r` that is the positive class, `"M"`, when `p` is greater than 0.9, and the negative class, `"R"`, otherwise.

```
# Apply threshold of 0.9
m_or_r <- ifelse(p > 0.90, "M", "R")
```

- Convert `m_or_r` to be a factor, `p_class`, with levels the same as those of `test[["Class"]]`.

```
p_class <- factor(m_or_r, levels = levels(test[["Class"]]))
```

- Make a confusion matrix with `confusionMatrix()`, passing `p_class` and the `"Class"` column from the `test` dataset.

```
# Create confusion matrix
confusionMatrix(p_class, test[["Class"]])
```

```
Confusion Matrix and Statistics

          Reference
Prediction  M  R
         M 10 27
         R 34 12

               Accuracy : 0.2651
                 95% CI : (0.1742, 0.3734)
    No Information Rate : 0.5301
    P-Value [Acc > NIR] : 1.0000

                  Kappa : -0.4603
 Mcnemar's Test P-Value : 0.4424

            Sensitivity : 0.2273
            Specificity : 0.3077
         Pos Pred Value : 0.2703
         Neg Pred Value : 0.2609
             Prevalence : 0.5301
         Detection Rate : 0.1205
   Detection Prevalence : 0.4458
      Balanced Accuracy : 0.2675

       'Positive' Class : M
```

3.5 From probabilites to confusion matrix

Conversely, say you want to be really certain that your model correctly identifies all the mines as mines. In this case, you might use a prediction threshold of 0.10, instead of 0.90.

- Use `ifelse()` to create a character vector, `m_or_r` that is the positive class, `"M"`, when `p` is greater than 0.1, and the negative class, `"R"`, otherwise.

```
# Apply threshold of 0.1
m_or_r <- ifelse(p > 0.10, "M", "R")
```

- Convert `m_or_r` to be a factor, `p_class`, with levels the same as those of `test[["Class"]]`.

```
p_class <- factor(m_or_r, levels = levels(test[["Class"]]))
```

- Make a confusion matrix with `confusionMatrix()`, passing `p_class` and the `"Class"` column from the `test` dataset.

```
# Create confusion matrix
confusionMatrix(p_class, test[["Class"]])
```

```
Confusion Matrix and Statistics

          Reference
Prediction  M  R
         M 11 30
         R 33  9

               Accuracy : 0.241
                 95% CI : (0.1538, 0.3473)
    No Information Rate : 0.5301
    P-Value [Acc > NIR] : 1.0000

                  Kappa : -0.517
 Mcnemar's Test P-Value : 0.8011

            Sensitivity : 0.2500
            Specificity : 0.2308
         Pos Pred Value : 0.2683
         Neg Pred Value : 0.2143
             Prevalence : 0.5301
         Detection Rate : 0.1325
   Detection Prevalence : 0.4940
      Balanced Accuracy : 0.2404

       'Positive' Class : M
```

Introducing the ROC curve video

What's the value of a ROC curve?

What is the primary value of an ROC curve?

- It has a cool acronym.
- It can be used to determine the true positive and false positive rates for a particular classification threshold.
- **It evaluates all possible thresholds for splitting predicted probabilities into predicted classes.**

3.6 Plot an ROC curve

As you saw in the video, an ROC curve is a really useful shortcut for summarizing the performance of a classifier over all possible thresholds. This saves you a lot of tedious work computing class predictions for many different thresholds and examining the confusion matrix for each.

My favorite package for computing ROC curves is `caTools` written by Tuszynski (2019), which contains a function called `colAUC()`. This function is very user-friendly and can actually calculate ROC curves for multiple predictors at once. In this case, you only need to calculate the ROC curve for one predictor, e.g.:

```
colAUC(predicted_probabilities, actual, plotROC = TRUE)
```

The function will return a score called AUC (more on that later) and the `plotROC = TRUE` argument will return the plot of the ROC curve for visual inspection.

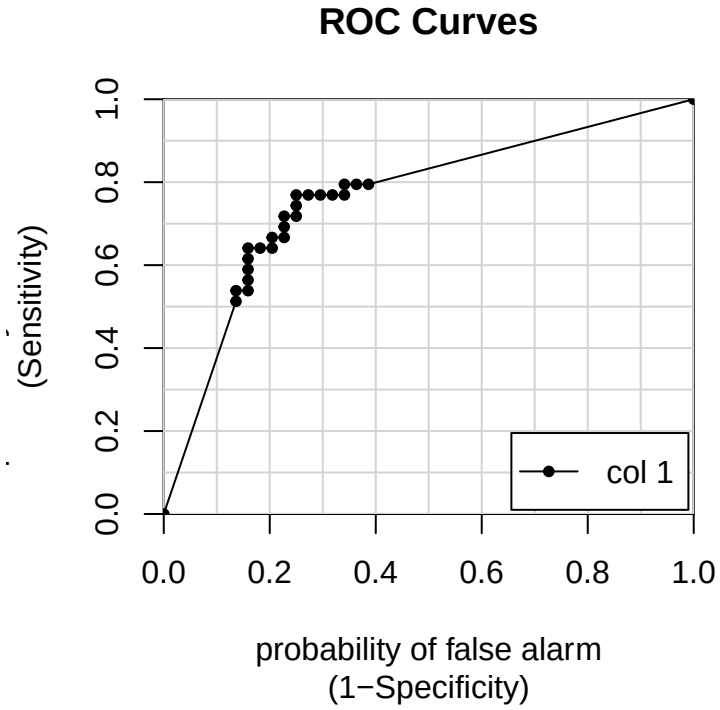
Exercise

- Predict probabilities (i.e. `type = "response"`) on the `test` set, then store the result as `p`.

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

- Make an ROC curve using the predicted test set probabilities.

```
colAUC(p, test$Class, plotROC = TRUE)
```




```
             [,1]
M vs. R 0.7645688
```

Area under the curve (AUC) video

Model, ROC, and AUC

What is the AUC of a perfect model?

- 0.00
- 0.50
- **1.00**

3.7 Customizing `trainControl`

As you saw in the video, area under the ROC curve is a very useful, single-number summary of a model's ability to discriminate the positive from the negative class (e.g. mines from rocks). An AUC of 0.5 is no better than random guessing, an AUC of 1.0 is a perfectly predictive model, and an AUC of 0.0 is perfectly anti-predictive (which rarely happens).

This is often a much more useful metric than simply ranking models by their accuracy at a set threshold, as different models might require different calibration steps (looking at a confusion matrix at each step) to find the optimal classification threshold for that model.

You can use the `trainControl()` function in `caret` to use AUC (instead of accuracy), to tune the parameters of your models. The `twoClassSummary()` convenience function allows you to do this easily.

When using `twoClassSummary()`, be sure to always include the argument `classProbs = TRUE` or your model will throw an error! (You cannot calculate AUC with just class predictions. You need to have class probabilities as well.)

Exercise

- Customize the `trainControl` object to use `twoClassSummary` rather than `defaultSummary`.
- Use 10-fold cross-validation.
- Be sure to tell `trainControl()`to return class probabilities.

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

3.8 Using custom `trainControl`

Now that you have a custom `trainControl` object, it's easy to fit caret models that use AUC rather than accuracy to tune and evaluate the model. You can just pass your custom `trainControl` object to the `train()` function via the `trControl` argument, e.g.:

```
train(<standard arguments here>, trControl = myControl)
```

This syntax gives you a convenient way to store a lot of custom modeling parameters and then use them across multiple different calls to `train()`. You will make extensive use of this trick in Chapter 5.

Exercise

- Use `train()` to fit a glm model (i.e. `method = "glm"`) to `Sonar` using your custom `trainControl` object, `myControl`. You want to predict `Class` from all other variables in the data (i.e. `Class ~ .`). Save the result to `model`.

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

- Print the `model` to the console and examine its output.

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

```
Generalized Linear Model

208 samples
 60 predictor
  2 classes: 'M', 'R'

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 187, 187, 187, 187, 187, 187, ...
Resampling results:

  ROC       Sens       Spec
  0.726835  0.7462121  0.6633333
```

Chapter 4

Tuning model parameters to improve performance

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

Random forests and wine video

Random forests vs. linear models

What's the primary advantage of random forests over linear models?

- They make you sound cooler during job interviews.
- You can't understand what's going on inside of a random forest model, so you don't have to explain it to anyone.
- **A random forest is a more flexible model than a linear model, but just as easy to fit.**

4.1 Fit a random forest

As you saw in the video, random forest models are much more flexible than linear models, and can model complicated nonlinear effects as well as automatically capture interactions between variables. They tend to give very good results on real world data, so let's try one out on the `wine` quality dataset, where the goal is to predict the human-evaluated quality of a batch of wine, given some of the machine-measured chemical and physical properties of that batch.

Fitting a random forest model is exactly the same as fitting a generalized linear regression model, as you did in the previous chapter. You simply change the method argument in the train function to be `"ranger"`. The `ranger` package written by Wright et al. (2019) is a rewrite of R's classic `randomForest` package written by Breiman et al. (2018) and fits models much faster, but gives almost exactly the same results. We suggest that all beginners use the `ranger` package for random forest modeling.

Exercise

- Train a random forest called `model` on the wine quality dataset, `wine`, such that `quality` is the response variable and all other variables are explanatory variables. Data is available from https://archive.ics.uci.edu/ml/machine-learning-databases/wine-quality/.
- Use `method = "ranger"`.
- Use a `tuneLength` of 1.
- Use 5 CV folds.
- Print `model` to the console.

```
library(caret)
# Load wine data set
wine <- read.csv("./Data/wine_dataset.csv")
set.seed(42)
# Fit random forest: model
model <- train(
  quality ~.,
  tuneLength = 1,
  data = wine,
  method = "ranger",
  trControl = trainControl(method = "cv",
                           number = 5,
                           verboseIter = FALSE)
)

# Print model to console
model
```

```
Random Forest

6497 samples
  12 predictor

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 5199, 5197, 5198, 5197, 5197
Resampling results across tuning parameters:

  splitrule   RMSE       Rsquared   MAE
  variance    0.5994113  0.5374130  0.4347958
  extratrees  0.6111844  0.5279777  0.4558074

Tuning parameter 'mtry' was held constant at a value of 3
Tuning
 parameter 'min.node.size' was held constant at a value of 5
RMSE was used to select the optimal model using the smallest value.
The final values used for the model were mtry = 3, splitrule =
 variance and min.node.size = 5.
```

```
model$finalModel
```

```
Ranger result

Call:
 ranger::ranger(dependent.variable.name = ".outcome", data = x,      mtry = min(param$mtry, ncol(x)), m

Type:                             Regression
Number of trees:                  500
Sample size:                      6497
Number of independent variables:  12
Mtry:                             3
Target node size:                 5
Variable importance mode:         none
Splitrule:                        variance
OOB prediction error (MSE):       0.337557
R squared (OOB):                  0.5573457
```

Explore a wider model space video

Advantage of a longer tune length

What's the advantage of a longer `tuneLength`?

- **You explore more potential models and can potentially find a better model.**
- Your models take less time to fit.
- There's no advantage; you'll always end up with the same final model.

4.2 Try a longer tune length

Recall from the video that random forest models have a primary tuning parameter of `mtry`, which controls how many variables are exposed to the splitting search routine at each split. For example, suppose that a tree has a total of 10 splits and `mtry = 2`. This means that there are 10 samples of 2 predictors each time a split is evaluated.

Use a larger tuning grid this time, but stick to the defaults provided by the `train()` function. Try a `tuneLength` of 3, rather than 1, to explore some more potential models, and plot the resulting model using the `plot` function.

Exercise

- Train a random forest model, `model`, using the `wine` dataset on the `quality` variable with all other variables as explanatory variables. (This will take a few seconds to run, so be patient!)

- Use `method = "ranger"`.

- Use a `tuneLength` of 3.

- Use 5 CV folds.

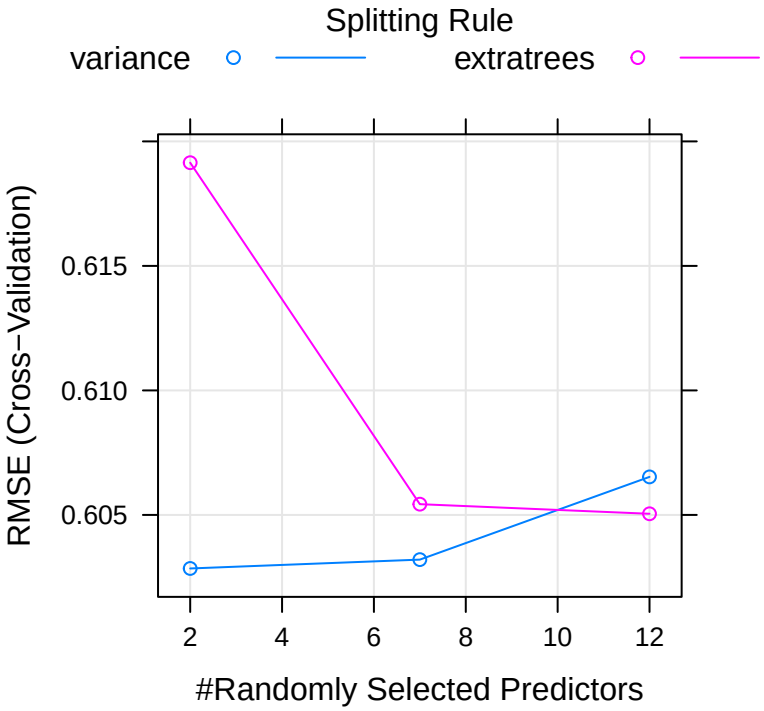
- Print `model` to the console.

- Plot the `model` after fitting it.

```
# Fit random forest: model
model <- train(
  quality ~ .,
  tuneLength = 3,
  data = wine, method = "ranger",
  trControl = trainControl(method = "cv", number = 5, verboseIter = FALSE)
)
# Print model to console
print(model)
```

```
Random Forest

6497 samples
  12 predictor

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 5198, 5197, 5198, 5198, 5197
Resampling results across tuning parameters:

  mtry  splitrule   RMSE       Rsquared   MAE
   2    variance    0.6028505  0.5364907  0.4415914
   2    extratrees  0.6191443  0.5227601  0.4665002
   7    variance    0.6032059  0.5280572  0.4352658
   7    extratrees  0.6054314  0.5297898  0.4456246
  12    variance    0.6065267  0.5209531  0.4358597
  12    extratrees  0.6050458  0.5271183  0.4428500

Tuning parameter 'min.node.size' was held constant at a value of 5
RMSE was used to select the optimal model using the smallest value.
The final values used for the model were mtry = 2, splitrule =
 variance and min.node.size = 5.
```

```
# Plot model
plot(model)
```



Custom tuning grids video

Advantages of a custom tuning grid

Why use a custom `tuneGrid`?

- There's no advantage; you'll always end up with the same final model.
- **It gives you more fine-grained control over the tuning parameters that are explored.**
- It always makes your models run faster.

4.3 Fit a random forest with custom tuning

Now that you've explored the default tuning grids provided by the `train()` function, let's customize your models a bit more.

You can provide any number of values for `mtry`, from 2 up to the number of columns in the dataset. In practice, there are diminishing returns for much larger values of `mtry`, so you will use a custom tuning grid that explores 2 simple models (`mtry = 2` and `mtry = 3`) as well as one more complicated model (`mtry = 7`).

Exercise

- Define a custom tuning grid.
  - Set the number of variables to possibly split at each node, `.mtry`, to a vector of 2, 3, and 7.
  - Set the rule to split on, `.splitrule`, to `"variance"`.
  - Set the minimum node size, `.min.node.size`, to 5.
- Train another random forest model, `model`, using the `wine` dataset on the `quality` variable with all other variables as explanatory variables.
  - Use `method = "ranger"`.
  - Use the custom `tuneGrid`.
  - Use 5 CV folds.

```
# Define the tuning grid: tuneGrid
tuneGrid <- data.frame(
  .mtry = c(2, 3, 7),
  .splitrule = "variance",
  .min.node.size = 5
)

# Fit random forest: model
model <- train(
  quality ~ .,
  tuneGrid = tuneGrid,
  data = wine,
  method = "ranger",
  trControl = trainControl(method = "cv",
                           number = 5,
                           verboseIter = FALSE)
)
```

- Print `model` to the console.

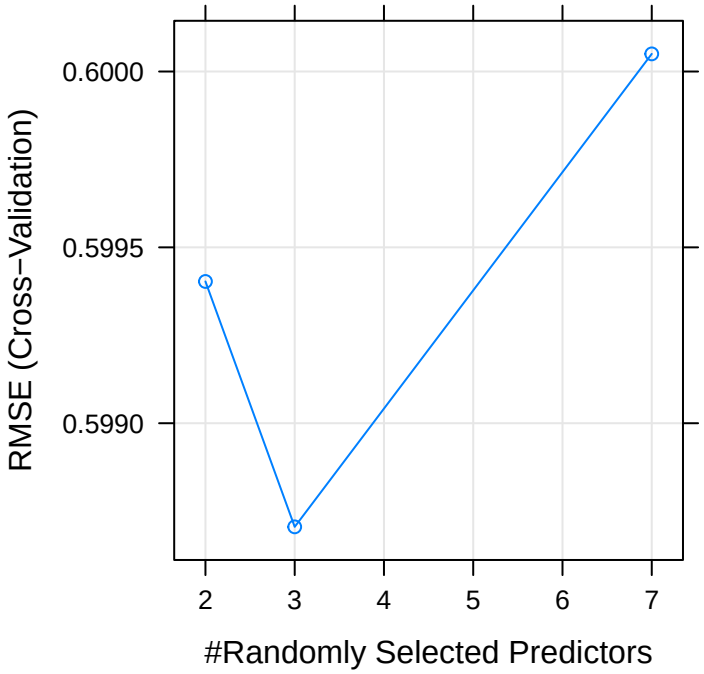
```
# Print model to console
model
```

```
Random Forest

6497 samples
  12 predictor

No pre-processing
Resampling: Cross-Validated (5 fold)
Summary of sample sizes: 5197, 5196, 5199, 5198, 5198
Resampling results across tuning parameters:

  mtry  RMSE       Rsquared   MAE
  2     0.5994031  0.5409142  0.4387527
  3     0.5987053  0.5384036  0.4354743
  7     0.6000501  0.5323440  0.4334891

Tuning parameter 'splitrule' was held constant at a value of
 variance
```

```
Tuning parameter 'min.node.size' was held constant at a value
 of 5
RMSE was used to select the optimal model using the smallest value.
The final values used for the model were mtry = 3, splitrule =
 variance and min.node.size = 5.
```

- Plot the `model` after fitting it using `plot()`.

```
# Plot model
plot(model)
```



Introducing `glmnet` video

Advantage of `glmnet`

What's the advantage of `glmnet` over regular `glm` models?

- `glmnet` models automatically find interaction variables.
- `glmnet` models don't provide p-values or confidence intervals on predictions.
- **`glmnet` models place constraints on your coefficients, which helps prevent overfitting.**

4.4 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.

Exercise

Make a custom `trainControl` called `myControl` for classification using the `trainControl` function.

- Use 10 CV folds.
- Use `twoClassSummary` for the `summaryFunction`.
- Be sure to set `classProbs = TRUE`.

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

4.5 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.

Exercise

Train a `glmnet` model called model on the `overfit` data. Use the custom `trainControl` from the previous exercise (`myControl`). The variable `y` is the response variable and all other variables are explanatory variables.

```
overfit <- read.csv("https://assets.datacamp.com/production/course_1048/datasets/overfit.csv")
model <- train(y ~ .,
               data = overfit,
               method = "glmnet",
               trControl = myControl)
```

```
Warning in train.default(x, y, weights = w, ...): The metric "Accuracy" was
not in the result set. ROC will be used instead.
```

- Print the `model` to the console.

```
# Print model
print(model)
```

```
glmnet

250 samples
200 predictors
  2 classes: 'class1', 'class2'

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 225, 225, 224, 225, 225, 225, ...
Resampling results across tuning parameters:

  alpha  lambda        ROC        Sens  Spec
  0.10   0.0001012745  0.4085145  0     0.9483696
  0.10   0.0010127448  0.4041667  0     0.9610507
  0.10   0.0101274483  0.4214674  0     0.9956522
  0.55   0.0001012745  0.4107790  0     0.9438406
  0.55   0.0010127448  0.4066123  0     0.9440217
  0.55   0.0101274483  0.4238225  0     0.9871377
  1.00   0.0001012745  0.3630435  0     0.9398551
  1.00   0.0010127448  0.3805254  0     0.9398551
  1.00   0.0101274483  0.4361413  0     0.9827899

ROC was used to select the optimal model using the largest value.
The final values used for the model were alpha = 1 and lambda = 0.01012745.
```

- Use the `max()` function to find the maximum of the ROC statistic contained somewhere in `model[["results"]]`.

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

```
[1] 0.4361413
```

---

glmnet with custom tuning grid video

Why a custom tuning grid?

Why use a custom tuning grid for a `glmnet` model?

- There's no reason to use a custom grid; the default is always the best.
- **The default tuning grid is very small and there are many more potential `glmnet` models you want to explore.**
- `glmnet` models are really slow, so you should never try more than a few tuning parameters.

4.6 `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.

Exercise

- Train a `glmnet` model on the `overfit` data such that `y` is the response variable and all other variables are explanatory variables. Make sure to use your custom `trainControl` from the previous exercise (`myControl`). Also, use a custom `tuneGrid` to explore `alpha = 0:1` and 20 values of `lambda` between 0.0001 and 1 per value of `alpha`.

```
# Train glmnet with custom trainControl and tuning: model
model <- train(
  y ~ ., data = overfit,
  tuneGrid = expand.grid(alpha  = 0:1,
                         lambda = seq(0.0001, 1, length = 20)),
  method = "glmnet",
  trControl = myControl
)
```

```
Warning in train.default(x, y, weights = w, ...): The metric "Accuracy" was
not in the result set. ROC will be used instead.
```

- Print `model` to the console.

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

```
glmnet

250 samples
200 predictors
  2 classes: 'class1', 'class2'

No pre-processing
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 225, 225, 226, 225, 225, 225, ...
Resampling results across tuning parameters:

  alpha  lambda      ROC        Sens  Spec
  0      0.00010000  0.4061594  0     0.9786232
  0      0.05272632  0.4379529  0     1.0000000
  0      0.10535263  0.4462862  0     1.0000000
  0      0.15797895  0.4613225  0     1.0000000
  0      0.21060526  0.4805254  0     1.0000000
  0      0.26323158  0.4910326  0     1.0000000
  0      0.31585789  0.4931159  0     1.0000000
  0      0.36848421  0.4972826  0     1.0000000
  0      0.42111053  0.4972826  0     1.0000000
  0      0.47373684  0.4929348  0     1.0000000
  0      0.52636316  0.4951087  0     1.0000000
  0      0.57898947  0.4971920  0     1.0000000
  0      0.63161579  0.4971920  0     1.0000000
  0      0.68424211  0.5015399  0     1.0000000
  0      0.73686842  0.5057065  0     1.0000000
  0      0.78949474  0.5057065  0     1.0000000
  0      0.84212105  0.5035326  0     1.0000000
  0      0.89474737  0.5035326  0     1.0000000
  0      0.94737368  0.5057065  0     1.0000000
  0      1.00000000  0.5057065  0     1.0000000
  1      0.00010000  0.3278080  0     0.9356884
  1      0.05272632  0.5268116  0     1.0000000
  1      0.10535263  0.5000000  0     1.0000000
  1      0.15797895  0.5000000  0     1.0000000
  1      0.21060526  0.5000000  0     1.0000000
  1      0.26323158  0.5000000  0     1.0000000
  1      0.31585789  0.5000000  0     1.0000000
  1      0.36848421  0.5000000  0     1.0000000
  1      0.42111053  0.5000000  0     1.0000000
  1      0.47373684  0.5000000  0     1.0000000
  1      0.52636316  0.5000000  0     1.0000000
  1      0.57898947  0.5000000  0     1.0000000
  1      0.63161579  0.5000000  0     1.0000000
  1      0.68424211  0.5000000  0     1.0000000
  1      0.73686842  0.5000000  0     1.0000000
  1      0.78949474  0.5000000  0     1.0000000
  1      0.84212105  0.5000000  0     1.0000000
  1      0.89474737  0.5000000  0     1.0000000
  1      0.94737368  0.5000000  0     1.0000000
```

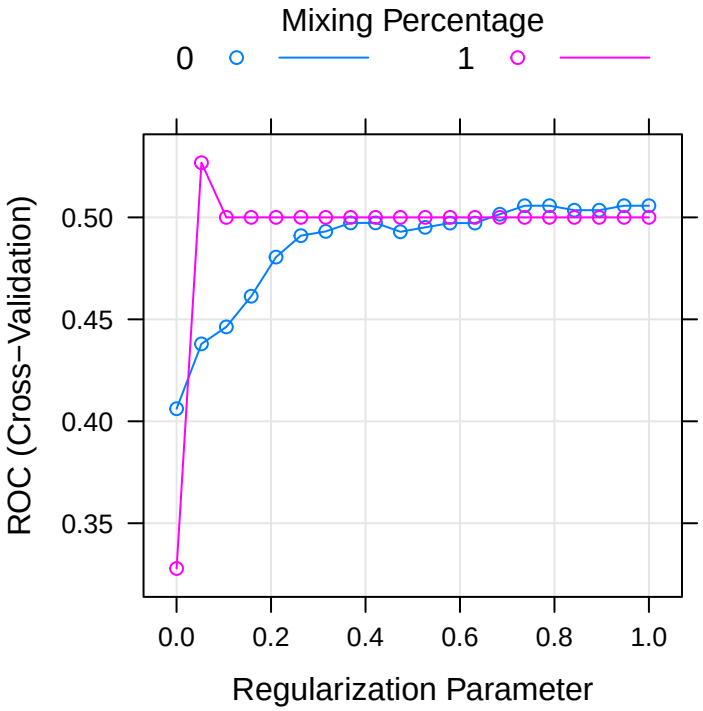


Figure 4.1: 'glmnet' plot

```
  1      1.00000000  0.5000000  0     1.0000000
```

```
ROC was used to select the optimal model using the largest value.
The final values used for the model were alpha = 1 and lambda = 0.05272632.
```

- Print the `max()` of the ROC statistic in `model[["results"]]`. You can access it using `model[["results"]][["ROC"]]`.

```
# Print maximum ROC statistic
max(model[["results"]][["ROC"]])
```

```
[1] 0.5268116
```

4.7 Interpreting `glmnet` plots

Figure 4.1 shows the tuning plot for the custom tuned `glmnet` model you created in the last exercise. For the `overfit` dataset, which value of `alpha` is better?

- `alpha = 0 (ridge)`
- **`alpha = 1 (lasso)`**

Chapter 5

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 vs. omitting rows

What's the value of median imputation?

- It removes some variance from your data, making it easier to model.
- **It lets you model data with missing values.**
- It's useless; you should just throw out rows of data with any missings.

5.1 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`.

```
url <- "https://assets.datacamp.com/production/course_1048/datasets/BreastCancer.RData"
download.file(url, "./Data/BreastCancer.RData")
load("./Data/BreastCancer.RData")
```

Exercise

- Use the `train()` function to fit a glm model called `model` to the breast cancer dataset. Use `preProcess = "medianImpute"` to handle the missing values.

```
library(caret)
# Create custom trainControl: myControl
myControl <- trainControl(
  method = "cv",
  number = 10,
  summaryFunction = twoClassSummary,
  classProbs = TRUE, # IMPORTANT!
  verboseIter = FALSE
)
# Apply median imputation: model
model <- train(
  x = breast_cancer_x, y = breast_cancer_y,
  method = "glm",
  trControl = myControl,
  preProcess = "medianImpute"
)
```

```
Warning in train.default(x = breast_cancer_x, y = breast_cancer_y, method =
"glm", : The metric "Accuracy" was not in the result set. ROC will be used
instead.
```

- Print model to the console.

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

```
Generalized Linear Model

699 samples
  9 predictor
  2 classes: 'benign', 'malignant'

Pre-processing: median imputation (9)
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 629, 630, 629, 629, 629, 628, ...
Resampling results:

  ROC        Sens       Spec
  0.9909642  0.9694686  0.9378333
```

Comparing KNN imputation to median 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.**
- Yes, KNN is a more complicated model than medians, so it's always better.
- No, medians are more statistically valid than KNN and should always be used.

5.2 Use KNN imputation

In the previous exercise, you used median imputation to fill in missing values in the breast cancer dataset, but that is not the only possible method for dealing with missing data.

An alternative to median imputation is k-nearest neighbors, or KNN, imputation. This is a more advanced form of imputation where missing values are replaced with values from other rows that are similar to the current row. While this is a lot more complicated to implement in practice than simple median imputation, it is very easy to explore in `caret` using the `preProcess` argument to `train()`. You can simply use `preProcess = "knnImpute"` to change the method of imputation used prior to model fitting.

Exercise

`breast_cancer_x` and `breast_cancer_y` are loaded in your workspace.

- Use the `train()` function to fit a glm model called `model2` to the breast cancer dataset.
- Use KNN imputation to handle missing values.

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

```
Warning in train.default(x = breast_cancer_x, y = breast_cancer_y, method =
"glm", : The metric "Accuracy" was not in the result set. ROC will be used
instead.
```

```
# Print model to console
model2
```

```
Generalized Linear Model

699 samples
  9 predictor
  2 classes: 'benign', 'malignant'

Pre-processing: nearest neighbor imputation (9), centered (9), scaled (9)
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 630, 629, 629, 629, 629, 629, ...
```

```
Resampling results:

  ROC        Sens       Spec
  0.9901472  0.9715942  0.942
```

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.

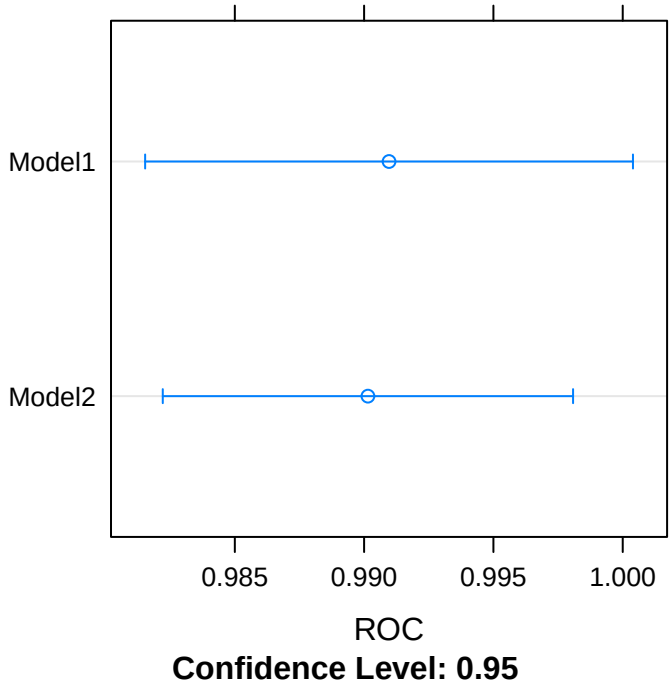
This includes the imputation method used (e.g. `knnImpute` or `medianImpute`). This is useful because it allows you to compare different methods of imputation and choose the one that performs the best out-of-sample.

`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?

```
median_model <- model
knn_model <- model2
ANS <- resamples(list(median_model, knn_model))
dotplot(ANS, metric = "ROC")
```



- KNN imputation is much better than median imputation.
- **KNN imputation is slightly better than median imputation.**
- Median imputation is much better than KNN imputation.

- Median imputation is slightly better than KNN imputation.

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.**
- Centering and scaling come before median imputation.

Note: Centering and scaling require data with no missing values.

5.3 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.

Exercise

`breast_cancer_x` and `breast_cancer_y` are loaded in your workspace. Fit two models called `model1` and `model2` to the breast cancer data, then print each to the console:

- A logistic regression model using only median imputation: `model1`

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

```
Warning in train.default(x = breast_cancer_x, y = breast_cancer_y, method =
"glm", : The metric "Accuracy" was not in the result set. ROC will be used
instead.
```

```
# Print model1
model1
```

```
Generalized Linear Model

699 samples
  9 predictor
```

```
  2 classes: 'benign', 'malignant'

Pre-processing: median imputation (9)
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 629, 629, 628, 630, 629, 629, ...
Resampling results:

  ROC        Sens       Spec
  0.9916449  0.9694203  0.9458333
```

- A logistic regression model using median imputation, centering, and scaling (in that order): `model2`

```
# 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")
)
```

```
Warning in train.default(x = breast_cancer_x, y = breast_cancer_y, method =
"glm", : The metric "Accuracy" was not in the result set. ROC will be used
instead.
```

```
# Print model2
model2
```

```
Generalized Linear Model

699 samples
  9 predictor
  2 classes: 'benign', 'malignant'

Pre-processing: median imputation (9), centered (9), scaled (9)
Resampling: Cross-Validated (10 fold)
Summary of sample sizes: 629, 629, 628, 630, 629, 629, ...
Resampling results:

  ROC        Sens      Spec
  0.9910757  0.969372  0.9418333
```

Why remove near zero variance predictors?

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

- Because they are guaranteed to have no effect on your model.
- Because their p-values in a linear regression will always be low.
- **To reduce model-fitting time without reducing model accuracy.**

Note: Low variance variables are unlikely to have a large impact on our models.

5.4 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()`.

Exercise

`bloodbrain_x` and `bloodbrain_y` are loaded in your workspace.

```
url <- "https://assets.datacamp.com/production/course_1048/datasets/BloodBrain.RData"
download.file(url, "./Data/BloodBrain.RData")
load("./Data/BloodBrain.RData")
```

- Identify the near zero variance predictors by running `nearZeroVar()` on the blood-brain dataset. Store the result as an object called `remove_cols`. Use `freqCut = 2` and `uniqueCut = 20` in the call to `nearZeroVar()`.

```
# Identify near zero variance predictors: remove
remove_cols <- nearZeroVar(bloodbrain_x, names = TRUE, freqCut = 2, uniqueCut = 20)
```

- Use `names()` to create a vector containing all column names of `bloodbrain_x`. Call this `all_cols`.

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

- Make a new data frame called `bloodbrain_x_small` with the near-zero variance variables removed. Use `setdiff()` to isolate the column names that you wish to keep (i.e. that you don't want to remove.)

```
# Remove from data: bloodbrain_x_small
bloodbrain_x_small <- bloodbrain_x[ , setdiff(all_cols, remove_cols)]
```

5.4.1 `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"`.**

- No, unfortunately. You have to do this by hand.

5.5 Fit model on reduced blood-brain data

Now that you've reduced your dataset, you can fit a glm model to it using the `train()` function. This model will run faster than using the full dataset and will yield very similar predictive accuracy.

Furthermore, zero variance variables can cause problems with cross-validation (e.g. if one fold ends up with only a single unique value for that variable), so removing them prior to modeling means you are less likely to get errors during the fitting process.

Exercise

`bloodbrain_x`, `bloodbrain_y`, `remove_cols`, and `bloodbrain_x_small` are loaded in your workspace.

- Fit a glm model using the `train()` function and the reduced blood-brain dataset you created in the previous exercise.

```
# Fit model on reduced data: model
model <- train(x = bloodbrain_x_small, y = bloodbrain_y, method = "glm")
```

- Print the result to the console.

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

```
Generalized Linear Model

208 samples
112 predictors

No pre-processing
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 208, 208, 208, 208, 208, 208, ...
Resampling results:

  RMSE      Rsquared  MAE
  1.640855  0.113519  1.104171
```

5.6 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.

Exercise

`bloodbrain_x` and `bloodbrain_y` are loaded in your workspace.

- Fit a `glm` model to the full blood-brain dataset using the `"pca"` option to `preProcess`.

```
# Fit glm model using PCA: model
model <- train(
  x = bloodbrain_x, y = bloodbrain_y,
  method = "glm", preProcess = "pca"
)
```

- Print the model to the console and inspect the result.

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

```
Generalized Linear Model

208 samples
132 predictors

Pre-processing: principal component signal extraction (132),
 centered (132), scaled (132)
Resampling: Bootstrapped (25 reps)
Summary of sample sizes: 208, 208, 208, 208, 208, 208, ...
Resampling results:

  RMSE       Rsquared   MAE
  0.6177954  0.4352449  0.4627265
```

Note that the PCA model's accuracy is slightly higher than the `nearZeroVar()` model from the previous exercise. PCA is generally a better method for handling low-information predictors than throwing them out entirely.

Chapter 6

Selecting models: a case study in churn prediction

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).

Why reuse a `trainControl`?

Why reuse a `trainControl`?

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

6.1 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.

Exercise

`churn_x` and `churn_y` are loaded in your workspace.

```
# library(C50)
# data(churn)
url <- "https://assets.datacamp.com/production/course_1048/datasets/Churn.RData"
download.file(url, "./Data/Churn.RData")
load("./Data/Churn.RData")
```

- Use `createFolds()` to create 5 CV folds on `churn_y`, your target variable for this exercise.

```
library(caret)
# Create custom indices: myFolds
myFolds <- createFolds(churn_y, k = 5)
```

- Pass them to `trainControl()` to create a reusable `trainControl` for comparing models.

```
# Create reusable trainControl object: myControl
myControl <- trainControl(
  summaryFunction = twoClassSummary,
  classProbs = TRUE, # IMPORTANT!
  verboseIter = FALSE,
  savePredictions = TRUE,
  index = myFolds
)
```

`glmnet` as a baseline model

What makes `glmnet` a good baseline model?

- **It's simple, fast, and easy to interpret.**
- It always gives poor predictions, so your other models will look good by comparison.
- Linear models with penalties on their coefficients always give better results.

6.2 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.

Exersize

Fit a `glmnet` model to the churn dataset called `model_glmnet`. Make sure to use `myControl`, which you created in the first exercise and is available in your workspace, as the `trainControl` object.

```
# Fit glmnet model: model_glmnet
model_glmnet <- train(
  x = churn_x, y = churn_y,
  metric = "ROC",
```

```
  method = "glmnet",
  trControl = myControl
)
```

Random forest drawback

What's the drawback of using a random forest model for churn prediction?

- Tree-based models are usually less accurate than linear models.
- **You no longer have model coefficients to help interpret the model.**
- Nobody else uses random forests to predict churn.

Note: Random forests are a little bit harder to interpret than linear models, though it is still possible to understand them.

6.3 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`.

Exercise

`churn_x` and `churn_y` are loaded in your workspace.

Fit a random forest model to the churn dataset. Be sure to use `myControl` as the `trainControl` like you've done before and implement the `"ranger"` method.

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

Matching train/test indices

What's the primary reason that train/test indices need to match when comparing two models?

- You can save a lot of time when fitting your models because you don't have to remake the datasets.
- There's no real reason; it just makes your plots look better.

- **Because otherwise you wouldn't be doing a fair comparison of your models and your results could be due to chance.**

Note: Train/test indexes allow you to evaluate your models out of sample so you know that they work!

6.4 Create a resamples object

Now that you have fit two models to the churn dataset, it's time to compare their out-of-sample predictions and choose which one is the best model for your dataset.

You can compare models in caret using the `resamples()` function, provided they have the same training data and use the same `trainControl` object with preset cross-validation folds. `resamples()` takes as input a list of models and can be used to compare dozens of models at once (though in this case you are only comparing two models).

Exercise

`model_glmnet` and `model_rf` are loaded in your workspace.

- Create a `list()` containing the glmnet model as item1 and the ranger model as item2.

```
# Create model_list
model_list <- list(glmnet = model_glmnet, rf = model_rf)
```

- Pass this list to the `resamples()` function and save the resulting object as `ANS`.

```
# Pass model_list to resamples(): ANS
ANS <- resamples(model_list)
```

- Summarize the results by calling `summary()` on `ANS`.

```
# Summarize the results
summary(ANS)
```

```
Call:
summary.resamples(object = ANS)

Models: glmnet, rf
Number of resamples: 5

ROC
            Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
glmnet 0.5422989 0.5485714 0.6301149 0.5996001 0.6315208 0.6454945    0
rf     0.5696552 0.6484306 0.6863736 0.6679428 0.7057143 0.7295402    0

Sens
            Min.   1st Qu.    Median      Mean   3rd Qu.      Max. NA's
glmnet 0.8793103 0.9080460 0.9428571 0.9357373 0.9657143 0.9827586    0
rf     0.8908046 0.9028571 0.9712644 0.9461215 0.9714286 0.9942529    0

Spec
```

```
       Min.   1st Qu.    Median      Mean 3rd Qu.      Max. NA's
glmnet    0 0.1538462 0.1538462 0.1495385    0.16 0.2800000    0
rf        0 0.0800000 0.1538462 0.1643077    0.28 0.3076923    0
```

6.5 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.

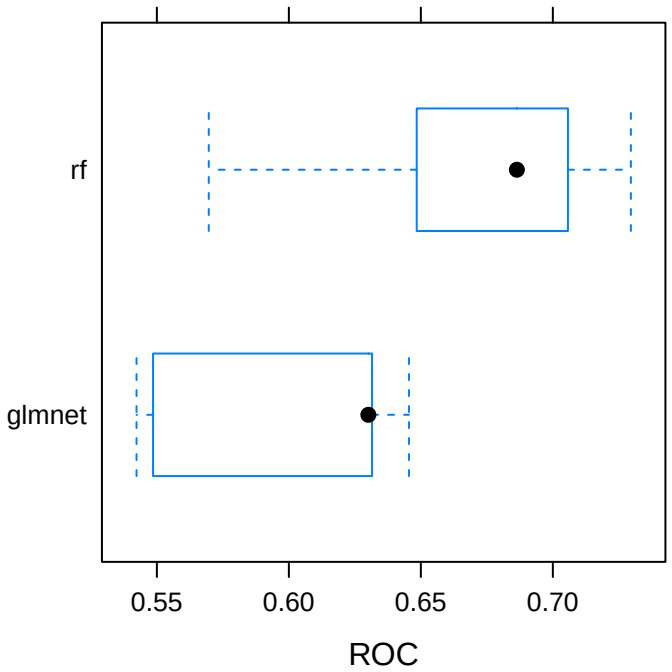
You can make this plot using the `bwplot()` function, which makes a box and whisker plot of the model's out of sample scores. Box and whisker plots show the median of each distribution as a line and the interquartile range of each distribution as a box around the median line. You can pass the `metric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

Exercise

Pass the `ANS` object to the `bwplot()` function to make a box-and-whisker plot. Look at the resulting plot and note which model has the higher median ROC statistic. Be sure to specify which metric you want to plot.

```
# Create bwplot
bwplot(ANS, metric = "ROC")
```



6.6 Create a scatterplot

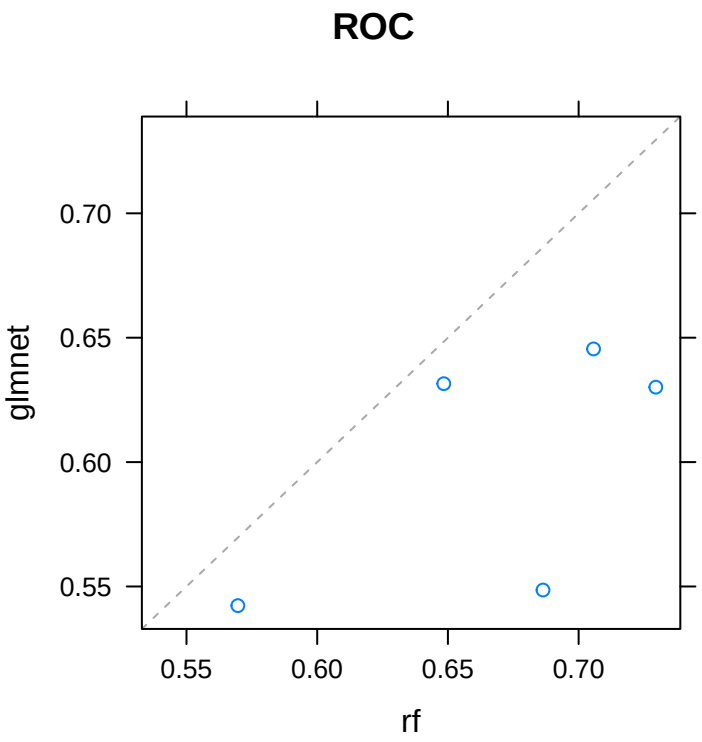
Another useful plot for comparing models is the scatterplot, also known as the xy-plot. This plot shows you how similar the two models' performances are on different folds.

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.

6.6.1 Exercise

Pass the `ANS` object to the `xyplot()` function. Look at the resulting plot and note how similar the two models' predictions are (or are not) on the different folds. Be sure to specify which metric you want to plot.

```
# Create xyplot
xyplot(ANS, metric = "ROC")
```



6.7 Ensembling models

That concludes the course! As a teaser for a future course on making ensembles of `caret` models, I'll show you how to fit a stacked ensemble of models using the `caretEnsemble` package.

`caretEnsemble` provides the `caretList()` function for creating multiple caret models at once on the same dataset, using the same resampling folds. You can also create your own lists of caret models.

In this exercise, I've made a `caretList` for you, containing the `glmnet` and `ranger` models you fit on the churn dataset. Use the `caretStack()` function to make a stack of caret models, with the two sub-models (`glmnet` and `ranger`) feeding into another (hopefully more accurate!) `caret` model.

Exercise

- Call the `caretStack()` function with two arguments, `model_list` and `method = "glm"`, to ensemble the two models using a logistic regression. Store the result as `stack`.

```
library(caretEnsemble)
models <- caretList(
  x = churn_x, y = churn_y,
  metric = "ROC",
  trControl = myControl,
  methodList = c("glmnet", "ranger")
)
# Create ensemble model: stack
stack <- caretStack(all.models = models, method = "glm")
```

- Summarize the resulting model with the `summary()` function.

```
summary(stack)
```

```
Call:
NULL

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-1.1683  -0.4982  -0.4446  -0.4202   2.2508

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
(Intercept)  -2.3877     0.1312 -18.196  < 2e-16 ***
glmnet       -0.6164     0.5115  -1.205    0.228
ranger        3.4854     0.6752   5.162 2.45e-07 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 765.13  on 999  degrees of freedom
Residual deviance: 727.72  on 997  degrees of freedom
AIC: 733.72

Number of Fisher Scoring iterations: 4
```

Bibliography

Breiman, L., Cutler, A., Liaw, A., and Wiener, M. (2018). *randomForest: Breiman and Cutler's Random Forests for Classification and Regression*. R package version 4.6-14.

from Jed Wing, M. K. C., Weston, S., Williams, A., Keefer, C., Engelhardt, A., Cooper, T., Mayer, Z., Kenkel, B., the R Core Team, Benesty, M., Lescarbeau, R., Ziem, A., Scrucca, L., Tang, Y., Candan, C., and Hunt., T. (2018). *caret: Classification and Regression Training*. R package version 6.0-81.

Tuszynski, J. (2019). *caTools: Tools: moving window statistics, GIF, Base64, ROC AUC, etc.* R package version 1.17.1.2.

Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., and Woo, K. (2018). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. R package version 3.1.0.

Wright, M. N., Wager, S., and Probst, P. (2019). *ranger: A Fast Implementation of Random Forests*. R package version 0.11.2.