Week 5 - Data Science II

Phileas Dazeley-Gaist

01/02/2022

Today: Cross validation and resampling methods.

Note: The standard training data to testing data ratio is 70% to 30%. Note: The central limit theorem states that the average of averages in a data set tends to be normally distributed.

```
# Applied Data Science II - Week 5
# # ---------------------------------------------------------------

# Today we are going to talk about RESAMPLING METHODS!
#
#
# # ---------------------------------------------------------------
#
# Load your libraries!
#
# # ---------------------------------------------------------------

library(ISLR2)
library(tidyverse)
```

```
## -- Attaching packages --------------------------------------- tidyverse 1.3.1 --
```

```
## v ggplot2 3.3.5     v purrr   0.3.4
## v tibble  3.1.6     v dplyr   1.0.7
## v tidyr   1.1.4     v stringr 1.4.0
## v readr   2.1.1     v forcats 0.5.1
```

```
## -- Conflicts ------------------------------------------ tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
```

```
library(caret)
```

```
## Loading required package: lattice
```

```
##
## Attaching package: 'caret'

## The following object is masked from 'package:purrr':
##
##     lift
```

```
library(palmerpenguins)
library(class)
library(nnet)
library(elasticnet)
```

```
## Loading required package: lars

## Loaded lars 1.2
```

```
library(boot)
```

```
##
## Attaching package: 'boot'

## The following object is masked from 'package:lattice':
##
##     melanoma
```

```
# do these not work? Then you'll have to install them!

# Uncomment the line below and run this command:
# install.packages(c("palmerpenguins","caret", "class","nnet", "boot","elasticnet"))
```

```
# # ----------------------------------------------------------------
#
# k-fold cross-validations
#
# # ----------------------------------------------------------------

# TLet's start by revisiting our simple penguin model from last week!


attach(penguins)


penguins_cleaned <- penguins %>%
        drop_na() %>%
        dplyr::select(species, bill_length_mm) %>%
```

```r
       mutate(adelie = as.factor(ifelse(species == 'Adelie',1,0))) %>%
       dplyr::select(-species)


# Now lets split it into test and train sets!
train <- sample(1:nrow(penguins_cleaned), nrow(penguins_cleaned) / 2)
test <- (-train)

# Let's set our "ground truth" labels.
truth <- penguins_cleaned$adelie[test]
```

```r
# Build our model
glm_train <- glm(adelie ~ ., family = binomial(link="logit"), data = penguins_cleaned, subset =

# And generate predictions on your test data ...
glm_preds <- predict(glm_train, penguins_cleaned[test,])
new_predictions <- rep(0,167)
new_predictions[glm_preds > 0.5] = 1

# now put them in a table and get our confusion matrix output:
xtab = table(new_predictions, truth)
caret::confusionMatrix(xtab)
```

```
## Confusion Matrix and Statistics
##
##                truth
## new_predictions  0  1
##               0 82  9
##               1  2 74
##
##                Accuracy : 0.9341
##                  95% CI : (0.8852, 0.9667)
##     No Information Rate : 0.503
##     P-Value [Acc > NIR] : < 2e-16
##
##                   Kappa : 0.8682
##
##  Mcnemar's Test P-Value : 0.07044
##
##             Sensitivity : 0.9762
##             Specificity : 0.8916
##          Pos Pred Value : 0.9011
##          Neg Pred Value : 0.9737
##              Prevalence : 0.5030
##          Detection Rate : 0.4910
##    Detection Prevalence : 0.5449
##       Balanced Accuracy : 0.9339
```

```
##
##        'Positive' Class : 0
##
```

```
# Okay, so as it stands - this model is approximately 56% accurate.
# Now, let's try using a 10-fold cross-validation approach.
# To do this in a way that is significantly less annoying than the textbook,
# we're going to use the caret library! So you're going to learn a few new things today!
```

Let's try that again with cross validation using caret (Classification and Regressions), and see if it improves our model.

```
# Let's build the model with caret:

glm_model_with_cv = train(
        form = adelie ~ .,
        data = penguins_cleaned,
        subset = train,
        trControl = trainControl(method = "cv", number = 10),
        method = "glm",
        family = "binomial"
)

# Let's break this down a little:
# glm_model_with_cv = train(                    -- the "train" function is from caret (??caret::
#         form = adelie ~ .,                    -- here, we specify the model form just like nor
#         data = penguins_cleaned,              -- specify the dataset, just like normal
#         subset = train,                       -- specify the subset, just like normal
#         trControl = trainControl(method = "cv", number = 10), --trControl is a SUPER powerfu
#         method = "glm",                       -- specify the model type like normal
#         family = "binomial"                   -- specify the model family like normal
# )

# Let's look at the model object

glm_model_with_cv
```

```
## Generalized Linear Model
##
## 333 samples
##   1 predictor
##   2 classes: '0', '1'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 150, 150, 148, 150, 149, 150, ...
## Resampling results:
```

```
##
##   Accuracy   Kappa
##   0.9330065  0.8590523
```

```
# And the summary...
summary(glm_model_with_cv) # this will report some stuff about the model including the trainin
```

```
##
## Call:
## NULL
##
## Deviance Residuals:
##      Min        1Q    Median        3Q       Max
## -1.98077  -0.14123  -0.02310   0.09211   2.52985
##
## Coefficients:
##                Estimate Std. Error z value Pr(>|z|)
## (Intercept)     43.2831     7.9061   5.475 4.38e-08 ***
## bill_length_mm  -1.0140     0.1848  -5.486 4.12e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 220.391  on 165  degrees of freedom
## Residual deviance:  46.379  on 164  degrees of freedom
## AIC: 50.379
##
## Number of Fisher Scoring iterations: 7
```

```
# let's now build our new predictions

even_newer_predictions <- predict(glm_model_with_cv, newdata = penguins_cleaned[test,])

caret::confusionMatrix(table(even_newer_predictions,truth))
```

```
## Confusion Matrix and Statistics
##
##                       truth
## even_newer_predictions  0  1
##                      0 81  6
##                      1  3 77
##
##                Accuracy : 0.9461
##                  95% CI : (0.9002, 0.9751)
##     No Information Rate : 0.503
```

```
##     P-Value [Acc > NIR] : <2e-16
##
##                   Kappa : 0.8922
##
##  Mcnemar's Test P-Value : 0.505
##
##             Sensitivity : 0.9643
##             Specificity : 0.9277
##          Pos Pred Value : 0.9310
##          Neg Pred Value : 0.9625
##              Prevalence : 0.5030
##          Detection Rate : 0.4850
##    Detection Prevalence : 0.5210
##       Balanced Accuracy : 0.9460
##
##        'Positive' Class : 0
##
```

```
# This new model is ~95% more accurate - an improvement of nearly 70%!
```

```
# # ---------------------------------------------------------------
#
# Stop! If you still have your Spotify code from last week, grab that up.
#
# # ---------------------------------------------------------------

# Let's try and improve our Spotify code!


spotify_data <- read_csv("w4 data/spotify_labels.csv")
```

```
## Rows: 13795 Columns: 15
## -- Column specification --------------------------------------------------------
## Delimiter: ","
## chr  (2): label, artist_name
## dbl (13): danceability, energy, key, loudness, mode, speechiness, acousticne...
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
```

```
cleaned_spotify <- spotify_data %>%
        dplyr::select(-artist_name,key) %>%
        mutate(mode = as.factor(mode),
               key = as.factor(key),
               time_signature = as.factor(time_signature),
               label = as.factor(label),
```

```r
              fun = energy * danceability,
              slowness = valence * tempo * loudness)


train <- sample(1:nrow(cleaned_spotify), nrow(cleaned_spotify) / 2)
test <- (-train)

multi_class_logit <- multinom(label ~ ., data = cleaned_spotify, subset = train)
```

```
## # weights:  116 (84 variable)
## initial  value 9561.272209
## iter  10 value 8915.232685
## iter  20 value 6444.860538
## iter  30 value 5843.676894
## iter  40 value 5612.104991
## iter  50 value 5534.346392
## iter  60 value 5498.988869
## iter  70 value 5478.625878
## iter  80 value 5475.171594
## iter  90 value 5475.074974
## final  value 5475.072944
## converged
```

```r
logit_pred <- predict(multi_class_logit, newdata=cleaned_spotify[test,], "class")
```

```r
# And generate predictions on your test data ...
logit_table <- table(cleaned_spotify$label[test], logit_pred)

caret::confusionMatrix(logit_table)
```

```
## Confusion Matrix and Statistics
##
##         logit_pred
##          hiphop indie metal  pop
##   hiphop    899   126    26  269
##   indie     122  1196   338  521
##   metal       5   240  1191   48
##   pop       164   391    41 1321
##
## Overall Statistics
##
##                Accuracy : 0.6679
##                  95% CI : (0.6566, 0.679)
##     No Information Rate : 0.313
##     P-Value [Acc > NIR] : < 2.2e-16
##
```

```
##                   Kappa : 0.5517
##
##  Mcnemar's Test P-Value : 3.102e-14
##
## Statistics by Class:
##
##                      Class: hiphop Class: indie Class: metal Class: pop
## Sensitivity                 0.7555       0.6124       0.7462     0.6119
## Specificity                 0.9262       0.8016       0.9447     0.8742
## Pos Pred Value              0.6811       0.5494       0.8026     0.6891
## Neg Pred Value              0.9478       0.8397       0.9252     0.8318
## Prevalence                  0.1725       0.2831       0.2314     0.3130
## Detection Rate              0.1303       0.1734       0.1727     0.1915
## Detection Prevalence        0.1914       0.3156       0.2151     0.2779
## Balanced Accuracy           0.8409       0.7070       0.8455     0.7430
```

```r
# Original accuracy is ~ 67%.
# newer accuracy ...

caret_spotify <- train(
        form = label ~ .,
        data = cleaned_spotify,
        subset = train,
        trControl = trainControl(method = "cv", number = 10),
        method = "multinom"
)
```

```
## # weights:  116 (84 variable)
## initial  value 8606.115394
## iter  10 value 8029.538501
## iter  20 value 5807.882967
## iter  30 value 5287.150471
## iter  40 value 5022.029707
## iter  50 value 4943.618803
## iter  60 value 4918.005790
## iter  70 value 4902.839489
## iter  80 value 4900.528244
## iter  90 value 4900.493108
## final  value 4900.488982
## converged
## # weights:  116 (84 variable)
## initial  value 8606.115394
## iter  10 value 8029.538522
## iter  20 value 5837.755535
## iter  30 value 5345.150436
## iter  40 value 5099.534215
## iter  50 value 5024.537424
```

```
## iter  60 value 5000.800975
## iter  70 value 4988.465114
## iter  80 value 4987.212856
## final  value 4987.206691
## converged
## # weights:  116 (84 variable)
## initial  value 8606.115394
## iter  10 value 8029.538501
## iter  20 value 5807.913785
## iter  30 value 5287.214347
## iter  40 value 5022.119722
## iter  50 value 4943.995095
## iter  60 value 4918.121670
## iter  70 value 4902.911229
## iter  80 value 4900.630841
## iter  90 value 4900.594821
## final  value 4900.591011
## converged
## # weights:  116 (84 variable)
## initial  value 8604.729099
## iter  10 value 8017.721672
## iter  20 value 5830.590047
## iter  30 value 5338.948958
## iter  40 value 5051.681290
## iter  50 value 4970.889837
## iter  60 value 4943.420026
## iter  70 value 4928.216207
## iter  80 value 4925.799243
## iter  90 value 4925.661793
## final  value 4925.646388
## converged
## # weights:  116 (84 variable)
## initial  value 8604.729099
## iter  10 value 8017.722483
## iter  20 value 5860.920455
## iter  30 value 5403.543012
## iter  40 value 5126.208424
## iter  50 value 5051.016418
## iter  60 value 5023.881697
## iter  70 value 5013.719333
## iter  80 value 5012.539014
## iter  90 value 5012.522740
## iter  90 value 5012.522735
## iter  90 value 5012.522735
## final  value 5012.522735
## converged
## # weights:  116 (84 variable)
## initial  value 8604.729099
```

```
## iter  10 value 8017.721672
## iter  20 value 5830.621054
## iter  30 value 5339.018714
## iter  40 value 5051.766970
## iter  50 value 4970.984323
## iter  60 value 4943.509202
## iter  70 value 4928.313827
## iter  80 value 4925.899849
## iter  90 value 4925.761833
## final  value 4925.746616
## converged
## # weights:  116 (84 variable)
## initial  value 8606.115394
## iter  10 value 8045.447378
## iter  20 value 5757.743075
## iter  30 value 5301.224803
## iter  40 value 5049.568502
## iter  50 value 4970.076596
## iter  60 value 4944.777241
## iter  70 value 4931.550261
## iter  80 value 4929.520991
## iter  90 value 4929.425368
## final  value 4929.421918
## converged
## # weights:  116 (84 variable)
## initial  value 8606.115394
## iter  10 value 8045.447495
## iter  20 value 5785.968395
## iter  30 value 5359.130394
## iter  40 value 5121.309399
## iter  50 value 5050.786081
## iter  60 value 5027.094351
## iter  70 value 5016.804106
## iter  80 value 5015.647859
## final  value 5015.642372
## converged
## # weights:  116 (84 variable)
## initial  value 8606.115394
## iter  10 value 8045.447387
## iter  20 value 5757.944541
## iter  30 value 5301.360896
## iter  40 value 5049.789055
## iter  50 value 4970.336006
## iter  60 value 4944.993479
## iter  70 value 4931.667308
## iter  80 value 4929.622813
## iter  90 value 4929.524704
## final  value 4929.521691
```

```
## converged
## # weights:  116 (84 variable)
## initial  value 8604.729099
## iter  10 value 7992.010657
## iter  20 value 5772.091999
## iter  30 value 5282.999784
## iter  40 value 5039.474483
## iter  50 value 4964.830765
## iter  60 value 4939.379709
## iter  70 value 4922.752118
## iter  80 value 4919.713990
## iter  90 value 4919.527945
## final  value 4919.525726
## converged
## # weights:  116 (84 variable)
## initial  value 8604.729099
## iter  10 value 7992.011734
## iter  20 value 5799.535310
## iter  30 value 5348.650947
## iter  40 value 5110.923246
## iter  50 value 5041.619761
## iter  60 value 5019.070541
## iter  70 value 5007.673304
## iter  80 value 5006.430555
## final  value 5006.427365
## converged
## # weights:  116 (84 variable)
## initial  value 8604.729099
## iter  10 value 7992.010658
## iter  20 value 5772.119756
## iter  30 value 5283.070367
## iter  40 value 5039.556130
## iter  50 value 4964.926094
## iter  60 value 4939.473381
## iter  70 value 4922.850823
## iter  80 value 4919.814712
## iter  90 value 4919.628894
## final  value 4919.626715
## converged
## # weights:  116 (84 variable)
## initial  value 8604.729099
## iter  10 value 8055.783980
## iter  20 value 5806.707105
## iter  30 value 5283.384374
## iter  40 value 5046.576074
## iter  50 value 4965.334325
## iter  60 value 4939.830605
## iter  70 value 4924.661623
```

```
## iter  80 value 4922.497769
## iter  90 value 4922.417117
## final  value 4922.416739
## converged
## # weights:  116 (84 variable)
## initial  value 8604.729099
## iter  10 value 8055.784201
## iter  20 value 5837.157024
## iter  30 value 5341.658397
## iter  40 value 5118.964503
## iter  50 value 5045.694704
## iter  60 value 5021.003235
## iter  70 value 5008.988053
## iter  80 value 5007.875747
## final  value 5007.869764
## converged
## # weights:  116 (84 variable)
## initial  value 8604.729099
## iter  10 value 8055.783980
## iter  20 value 5806.737720
## iter  30 value 5283.447718
## iter  40 value 5046.660603
## iter  50 value 4965.428637
## iter  60 value 4939.923171
## iter  70 value 4924.759413
## iter  80 value 4922.598188
## iter  90 value 4922.516767
## final  value 4922.516395
## converged
## # weights:  116 (84 variable)
## initial  value 8604.729099
## iter  10 value 8050.128595
## iter  20 value 5876.663430
## iter  30 value 5367.358098
## iter  40 value 5078.766797
## iter  50 value 4982.590016
## iter  60 value 4956.867262
## iter  70 value 4940.923201
## iter  80 value 4938.240763
## iter  90 value 4938.162301
## final  value 4938.158206
## converged
## # weights:  116 (84 variable)
## initial  value 8604.729099
## iter  10 value 8050.128755
## iter  20 value 5907.082682
## iter  30 value 5428.212068
## iter  40 value 5149.213345
```

```
## iter  50 value 5064.813689
## iter  60 value 5042.556917
## iter  70 value 5026.895440
## iter  80 value 5025.323984
## final  value 5025.313350
## converged
## # weights:  116 (84 variable)
## initial  value 8604.729099
## iter  10 value 8050.128596
## iter  20 value 5876.694976
## iter  30 value 5367.425834
## iter  40 value 5078.850065
## iter  50 value 4982.686815
## iter  60 value 4956.963854
## iter  70 value 4941.022429
## iter  80 value 4938.342890
## iter  90 value 4938.263757
## final  value 4938.259740
## converged
## # weights:  116 (84 variable)
## initial  value 8604.729099
## iter  10 value 8037.922000
## iter  20 value 5829.914544
## iter  30 value 5294.605530
## iter  40 value 5049.388950
## iter  50 value 4968.337732
## iter  60 value 4940.924364
## iter  70 value 4923.041123
## iter  80 value 4920.391349
## iter  90 value 4920.348873
## final  value 4920.344484
## converged
## # weights:  116 (84 variable)
## initial  value 8604.729099
## iter  10 value 8037.922207
## iter  20 value 5861.305257
## iter  30 value 5354.706994
## iter  40 value 5121.201063
## iter  50 value 5048.528066
## iter  60 value 5021.051828
## iter  70 value 5008.641979
## iter  80 value 5007.579546
## final  value 5007.574741
## converged
## # weights:  116 (84 variable)
## initial  value 8604.729099
## iter  10 value 8037.922000
## iter  20 value 5829.946326
```

```
## iter  30 value 5294.670959
## iter  40 value 5049.471880
## iter  50 value 4968.432847
## iter  60 value 4941.120391
## iter  70 value 4923.147439
## iter  80 value 4920.494375
## iter  90 value 4920.451162
## final  value 4920.446813
## converged
## # weights:  116 (84 variable)
## initial  value 8604.729099
## iter  10 value 8063.013821
## iter  20 value 5901.970473
## iter  30 value 5316.917202
## iter  40 value 5061.194358
## iter  50 value 4983.537983
## iter  60 value 4954.804376
## iter  70 value 4939.379153
## iter  80 value 4936.794803
## iter  90 value 4936.777380
## final  value 4936.776345
## converged
## # weights:  116 (84 variable)
## initial  value 8604.729099
## iter  10 value 8063.013855
## iter  20 value 5933.106255
## iter  30 value 5378.741329
## iter  40 value 5136.259034
## iter  50 value 5064.546682
## iter  60 value 5040.470386
## iter  70 value 5026.287666
## iter  80 value 5024.882584
## final  value 5024.875106
## converged
## # weights:  116 (84 variable)
## initial  value 8604.729099
## iter  10 value 8063.013821
## iter  20 value 5902.001901
## iter  30 value 5316.985088
## iter  40 value 5061.281760
## iter  50 value 4983.632487
## iter  60 value 4954.901775
## iter  70 value 4939.482382
## iter  80 value 4936.902733
## iter  90 value 4936.885190
## final  value 4936.884012
## converged
## # weights:  116 (84 variable)
```

```
## initial  value 8604.729099
## iter  10 value 8041.281618
## iter  20 value 5842.404532
## iter  30 value 5282.112789
## iter  40 value 5057.239036
## iter  50 value 4967.756393
## iter  60 value 4941.868830
## iter  70 value 4925.187355
## iter  80 value 4922.943170
## iter  90 value 4922.894632
## final  value 4922.888314
## converged
## # weights:  116 (84 variable)
## initial  value 8604.729099
## iter  10 value 8041.281636
## iter  20 value 5872.354913
## iter  30 value 5344.391318
## iter  40 value 5120.874444
## iter  50 value 5048.637712
## iter  60 value 5023.217471
## iter  70 value 5010.415306
## iter  80 value 5009.104511
## final  value 5009.096023
## converged
## # weights:  116 (84 variable)
## initial  value 8604.729099
## iter  10 value 8041.281618
## iter  20 value 5842.435810
## iter  30 value 5282.182868
## iter  40 value 5057.325741
## iter  50 value 4967.849671
## iter  60 value 4941.941084
## iter  70 value 4925.283998
## iter  80 value 4923.045163
## iter  90 value 4922.995875
## final  value 4922.989601
## converged
## # weights:  116 (84 variable)
## initial  value 8606.115394
## iter  10 value 8055.761569
## iter  20 value 5765.648913
## iter  30 value 5302.496025
## iter  40 value 5047.714545
## iter  50 value 4960.079693
## iter  60 value 4931.999692
## iter  70 value 4916.294362
## iter  80 value 4913.951208
## iter  90 value 4913.903338
```

```
## final  value 4913.895558
## converged
## # weights:  116 (84 variable)
## initial  value 8606.115394
## iter  10 value 8055.761738
## iter  20 value 5794.882873
## iter  30 value 5361.126940
## iter  40 value 5117.494946
## iter  50 value 5038.630219
## iter  60 value 5013.790066
## iter  70 value 5001.168419
## iter  80 value 4999.988711
## iter  90 value 4999.982690
## iter  90 value 4999.982682
## iter  90 value 4999.982682
## final  value 4999.982682
## converged
## # weights:  116 (84 variable)
## initial  value 8606.115394
## iter  10 value 8055.761569
## iter  20 value 5765.678415
## iter  30 value 5302.559512
## iter  40 value 5047.795172
## iter  50 value 4960.171059
## iter  60 value 4932.089813
## iter  70 value 4916.393371
## iter  80 value 4914.054301
## iter  90 value 4914.005601
## final  value 4913.997814
## converged
## # weights:  116 (84 variable)
## initial  value 9561.272209
## iter  10 value 8915.232685
## iter  20 value 6444.895046
## iter  30 value 5843.746106
## iter  40 value 5612.189474
## iter  50 value 5534.439822
## iter  60 value 5499.094124
## iter  70 value 5478.726102
## iter  80 value 5475.273294
## iter  90 value 5475.175724
## final  value 5475.173706
## converged
```

```
new_pred <- predict(caret_spotify, newdata=cleaned_spotify[test,])
new_table <- table(cleaned_spotify$label[test], new_pred)
caret::confusionMatrix(table(cleaned_spotify$label[test],new_pred))
```

```
## Confusion Matrix and Statistics
##
##         new_pred
##          hiphop indie metal  pop
##   hiphop    899   126    26  269
##   indie     122  1196   338  521
##   metal       5   240  1191   48
##   pop       164   391    41 1321
##
## Overall Statistics
##
##                Accuracy : 0.6679
##                  95% CI : (0.6566, 0.679)
##     No Information Rate : 0.313
##     P-Value [Acc > NIR] : < 2.2e-16
##
##                   Kappa : 0.5517
##
##  Mcnemar's Test P-Value : 3.102e-14
##
## Statistics by Class:
##
##                      Class: hiphop Class: indie Class: metal Class: pop
## Sensitivity                 0.7555       0.6124       0.7462     0.6119
## Specificity                 0.9262       0.8016       0.9447     0.8742
## Pos Pred Value              0.6811       0.5494       0.8026     0.6891
## Neg Pred Value              0.9478       0.8397       0.9252     0.8318
## Prevalence                  0.1725       0.2831       0.2314     0.3130
## Detection Rate              0.1303       0.1734       0.1727     0.1915
## Detection Prevalence        0.1914       0.3156       0.2151     0.2779
## Balanced Accuracy           0.8409       0.7070       0.8455     0.7430
```

```
# slight improvements, but not all that much!

# # ----------------------------------------------------------------
#
#  Stop! Go back to the presentation
#
# # ----------------------------------------------------------------
```

Bootstrapping is resampling with replacement a bunch of times and using the data bins to generate a parameter estimate (mean, median, anything really…)

```
# # ----------------------------------------------------------------
#
#  The Bootstrap!
#
```

```r
# # -----------------------------------------------------------------

# One of the great advantages of the bootstrap approach is that it can be
# applied in almost all situations. No complicated mathematical calculations
# are required. Performing a bootstrap analysis in R entails only two steps.
# First, we must create a function that computes the statistic of interest.
# Second, we use the boot() function, which is part of the boot library,
# to boot() perform the bootstrap by repeatedly sampling observations from the data
# set with replacement.
#
# Note: one of the main motivations for doing the below exercise is to get you
# comfortable with the idea of *writing your own functions*!

# let's right a function that uses the bootstrap to get the
# standard error estimates of a linear regresion model!

# write a function!
boot.fn <- function(data, index){
        coef(lm(mpg ~ horsepower, data = data, subset = index))
}

# This function - called boot.fn - takes two arguments: "data" and "index".
# As you can see, what this does is it plugs in whatever you pass into the "data" argument and
# arguments into a linear regression and records the coefficients. Note that this is
# also "hard-coded" to only work with the Auto dataset.

# Run the function once! (Just to try it out)
boot.fn(Auto, 1:392)
```

```
## (Intercept)  horsepower
##  39.9358610  -0.1578447
```

```r
# This is the same as:
coef(lm(mpg ~ horsepower, data = Auto))
```

```
## (Intercept)  horsepower
##  39.9358610  -0.1578447
```

```r
# Now, let's get bootstrappy!
set.seed(1)
boot.fn(Auto, sample(392, 392, replace = T))
```

```
## (Intercept)  horsepower
##  40.3404517  -0.1634868
```

```r
# What did this do? It randomnly sampled - with replacement - from the existing 392 elements.

# Now we can use the boot function from the boot library (??boot) to do this 1000 times!
boot(Auto, boot.fn, 1000)
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = Auto, statistic = boot.fn, R = 1000)
##
##
## Bootstrap Statistics :
##       original        bias    std. error
## t1* 39.9358610  0.0549915227 0.841925746
## t2* -0.1578447 -0.0006210818 0.007348956
```

```r
# Now compare this to the normal standard errors...
summary(lm(mpg ~ horsepower, data = Auto))$coef
```

```
##               Estimate  Std. Error   t value      Pr(>|t|)
## (Intercept) 39.9358610 0.717498656  55.65984 1.220362e-187
## horsepower  -0.1578447 0.006445501 -24.48914  7.031989e-81
```

```r
# What do you notice?
# t1 and t2 are the terms of the regression (intercept, horsepower, in this case)
```

```r
# # ---------------------------------------------------------------
#
#  Coding Project!
#
# # ---------------------------------------------------------------

# # ---------------------------------------------------------------
#
#  Quick lesson - Adding Cross-Validation to Regression (+ learning more Caret stuff!)
#
# # ---------------------------------------------------------------

# For this part, we're going to load in the Diamonds dataset

data(diamonds)


# We're going to build a model that predicts the PRICE of the diamond data.
```

```r
# We're going to use a 70/30 split - that is, we're going to use 70% of the data to train and
# 30% of the data to split.
#
# ... not sure how to do that easily? No worries, caret got you!

# Caret has a handy function called "createDataPartition. Check it out.

diamonds_indx = createDataPartition(diamonds$price, p = 0.70, list = FALSE)
# This splits the diamonds dataset into a 70%/30% split.
diamonds_train = diamonds[diamonds_indx, ]
diamonds_test = diamonds[-diamonds_indx, ]
```

```r
# Now, let's build a simple linear regression with caret to see how we'd do it...

diamonds_linear <- train(
        form = price ~ .,
        data = diamonds_train,
        method = "lm"
)

# what's our training RMSE?
diamonds_linear
```

```
## Linear Regression
##
## 37759 samples
##     9 predictor
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 37759, 37759, 37759, 37759, 37759, 37759, ...
## Resampling results:
##
##   RMSE     Rsquared  MAE
##   1136.44  0.918622  742.8201
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```r
# what's our test RMSE?
diamonds_linear_prediction <- predict(diamonds_linear,diamonds_test)

# we can use this handy function:
postResample(pred = diamonds_linear_prediction, obs = diamonds_test$price)
```

```
##         RMSE     Rsquared          MAE
## 1133.6559562    0.9187107  741.6051467
```

```r
# not bad!

# ...how would we do a lasso model? pretty simple!
# first we setup our values for lambda again (this is our penalty variable)
lambda <- c(seq(0.1, 2, by =0.1) ,  seq(2, 5, 0.5) , seq(5, 25, 1))

# and let's process our data a bit to make sure that lasso works well on it!

y_train = diamonds_train$price
x_train <- model.matrix( ~ .-price, diamonds_train)
x_test <- model.matrix( ~ . -price, diamonds_test)

lasso<-train(y= y_train,
             x = x_train,
             method = 'glmnet',
             tuneGrid = expand.grid(alpha = 1, lambda = lambda),
             trControl = trainControl(method = "cv", number = 10)
)

lasso
```

```
## glmnet
##
## 37759 samples
##    24 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 33983, 33983, 33983, 33983, 33982, 33983, ...
## Resampling results across tuning parameters:
##
##   lambda  RMSE      Rsquared   MAE
##    0.1    1130.503  0.9199922  744.4903
##    0.2    1130.503  0.9199922  744.4903
##    0.3    1130.503  0.9199922  744.4903
##    0.4    1130.503  0.9199922  744.4903
##    0.5    1130.503  0.9199922  744.4903
##    0.6    1130.503  0.9199922  744.4903
##    0.7    1130.503  0.9199922  744.4903
##    0.8    1130.503  0.9199922  744.4903
##    0.9    1130.503  0.9199922  744.4903
##    1.0    1130.503  0.9199922  744.4903
##    1.1    1130.503  0.9199922  744.4903
##    1.2    1130.503  0.9199922  744.4903
##    1.3    1130.503  0.9199922  744.4903
##    1.4    1130.503  0.9199922  744.4903
##    1.5    1130.503  0.9199922  744.4903
```

```
##    1.6    1130.503  0.9199922  744.4903
##    1.7    1130.503  0.9199922  744.4903
##    1.8    1130.503  0.9199922  744.4903
##    1.9    1130.503  0.9199922  744.4903
##    2.0    1130.514  0.9199908  744.5321
##    2.5    1130.674  0.9199694  745.1548
##    3.0    1130.877  0.9199425  745.7887
##    3.5    1131.109  0.9199115  746.4496
##    4.0    1131.370  0.9198762  747.1253
##    4.5    1131.664  0.9198366  747.8191
##    5.0    1131.987  0.9197929  748.5497
##    6.0    1132.733  0.9196916  750.0543
##    7.0    1133.598  0.9195739  751.6315
##    8.0    1134.590  0.9194385  753.3115
##    9.0    1135.711  0.9192848  755.1001
##   10.0    1136.949  0.9191148  756.9768
##   11.0    1138.309  0.9189273  758.9425
##   12.0    1139.794  0.9187221  761.0154
##   13.0    1141.395  0.9185001  763.2026
##   14.0    1143.115  0.9182609  765.4952
##   15.0    1144.946  0.9180053  767.8804
##   16.0    1146.884  0.9177340  770.3424
##   17.0    1148.931  0.9174466  772.9002
##   18.0    1151.082  0.9171436  775.5631
##   19.0    1153.312  0.9168284  778.2796
##   20.0    1155.641  0.9164981  781.0612
##   21.0    1157.962  0.9161682  783.7925
##   22.0    1160.380  0.9158235  786.5855
##   23.0    1162.467  0.9155266  788.9891
##   24.0    1164.655  0.9152144  791.4436
##   25.0    1166.172  0.9149988  792.7209
##
## Tuning parameter 'alpha' was held constant at a value of 1
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were alpha = 1 and lambda = 1.9.
```

```
# what's our test RMSE? (RMSE is the average error in the unit of the thing we are predicting)
diamonds_lasso_prediction <- predict(lasso,x_test)

# we can use this handy function:
postResample(pred = diamonds_lasso_prediction, obs = diamonds_test$price)
```

```
##         RMSE     Rsquared          MAE
## 1132.7784333    0.9188236  744.6726725
```

```
# slightly better but not by much!
```

```
# # ---------------------------------------------------------------
#
#  Your turn!
#
# Check out the file called "abalone.csv" on the Google Drive. This file
# is measurement of approximately 4,000 abalone. Your job is to try and predict
# the abalone's age (measured in rings that are viewed upon shucking) from the
# variables provided. I want you to make sure you use cross-validation to produce the
# minimum RMSE that you can. You can use a linear model, lasso model, or a ridge regression
# (which, if you remember, you can do if you set the "alpha = 0" above).
# # ---------------------------------------------------------------

abalone <- read.csv("w5 data/abalone.csv")
abalone <- as.data.frame(unclass(abalone),
                       stringsAsFactors = TRUE)
head(abalone)
```

```
##   sex length diameter height whole_weight shucked_weight viscera_weight
## 1   M  0.455    0.365  0.095       0.5140         0.2245         0.1010
## 2   M  0.350    0.265  0.090       0.2255         0.0995         0.0485
## 3   F  0.530    0.420  0.135       0.6770         0.2565         0.1415
## 4   M  0.440    0.365  0.125       0.5160         0.2155         0.1140
## 5   I  0.330    0.255  0.080       0.2050         0.0895         0.0395
## 6   I  0.425    0.300  0.095       0.3515         0.1410         0.0775
##   shell_weight age_in_rings
## 1        0.150           15
## 2        0.070            7
## 3        0.210            9
## 4        0.155           10
## 5        0.055            7
## 6        0.120            8
```

```
# Caret has a handy function called "createDataPartition. Check it out.

abalone_indx = createDataPartition(abalone$age_in_rings, p = 0.70, list = FALSE)
# This splits the diamonds dataset into a 70%/30% split.
abalone_train = abalone[abalone_indx, ]
abalone_test = abalone[-abalone_indx, ]
```

```
# Now, let's build a simple linear regression with caret to see how we'd do it...

abalone_linear <- train(
        form = age_in_rings ~ .,
        data = abalone_train,
```

```r
        method = "lm"
)

# what's our training RMSE?
abalone_linear
```

```
## Linear Regression
##
## 2925 samples
##    8 predictor
##
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 2925, 2925, 2925, 2925, 2925, 2925, ...
## Resampling results:
##
##   RMSE      Rsquared   MAE
##   2.218979  0.5366875  1.592062
##
## Tuning parameter 'intercept' was held constant at a value of TRUE
```

```r
# what's our test RMSE?
abalone_linear_prediction <- predict(abalone_linear,abalone_test)

# we can use this handy function:
postResample(pred = abalone_linear_prediction, obs = abalone_test$age_in_rings)
```

```
##      RMSE  Rsquared       MAE
## 2.2108383 0.5184602 1.5845033
```

```r
# not bad!
```

```r
# ...how would we do a lasso model? pretty simple!
# first we set-up our values for lambda again (this is our penalty variable)
lambda <- c(seq(0.1, 2, by =0.1) ,  seq(2, 5, 0.5) , seq(5, 25, 1))

# and let's process our data a bit to make sure that lasso works well on it!

y_train = abalone_train$age_in_rings
x_train <- model.matrix( ~ .-age_in_rings, abalone_train)
x_test <- model.matrix( ~ . -age_in_rings, abalone_test)

lasso<-train(y= y_train,
             x = x_train,
             method = 'glmnet',
             tuneGrid = expand.grid(alpha = 1, lambda = lambda),
```

```r
             trControl = trainControl(method = "cv", number = 10)
)
```

```
## Warning in nominalTrainWorkflow(x = x, y = y, wts = weights, info = trainInfo, :
## There were missing values in resampled performance measures.
```

```r
lasso
```

```
## glmnet
##
## 2925 samples
##   10 predictor
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 2634, 2634, 2632, 2632, 2631, 2633, ...
## Resampling results across tuning parameters:
##
##   lambda  RMSE      Rsquared   MAE
##    0.1    2.341012  0.4878119  1.666749
##    0.2    2.461582  0.4318266  1.773311
##    0.3    2.522378  0.4040757  1.828298
##    0.4    2.539786  0.4017143  1.843074
##    0.5    2.562067  0.3985242  1.861845
##    0.6    2.588277  0.3947400  1.884285
##    0.7    2.614211  0.3929436  1.905898
##    0.8    2.640690  0.3934423  1.925820
##    0.9    2.670668  0.3943019  1.947674
##    1.0    2.704330  0.3952737  1.971680
##    1.1    2.741520  0.3964034  1.997917
##    1.2    2.782105  0.3976926  2.026363
##    1.3    2.825950  0.3990672  2.057645
##    1.4    2.872900  0.4003442  2.092869
##    1.5    2.922790  0.4008200  2.131289
##    1.6    2.975272  0.4008960  2.171878
##    1.7    3.030136  0.4008960  2.214052
##    1.8    3.087264  0.4008960  2.256997
##    1.9    3.146532  0.4008960  2.300614
##    2.0    3.207821  0.4008960  2.347368
##    2.5    3.238553        NaN  2.374812
##    3.0    3.238553        NaN  2.374812
##    3.5    3.238553        NaN  2.374812
##    4.0    3.238553        NaN  2.374812
##    4.5    3.238553        NaN  2.374812
##    5.0    3.238553        NaN  2.374812
##    6.0    3.238553        NaN  2.374812
```

```
##    7.0    3.238553        NaN  2.374812
##    8.0    3.238553        NaN  2.374812
##    9.0    3.238553        NaN  2.374812
##   10.0    3.238553        NaN  2.374812
##   11.0    3.238553        NaN  2.374812
##   12.0    3.238553        NaN  2.374812
##   13.0    3.238553        NaN  2.374812
##   14.0    3.238553        NaN  2.374812
##   15.0    3.238553        NaN  2.374812
##   16.0    3.238553        NaN  2.374812
##   17.0    3.238553        NaN  2.374812
##   18.0    3.238553        NaN  2.374812
##   19.0    3.238553        NaN  2.374812
##   20.0    3.238553        NaN  2.374812
##   21.0    3.238553        NaN  2.374812
##   22.0    3.238553        NaN  2.374812
##   23.0    3.238553        NaN  2.374812
##   24.0    3.238553        NaN  2.374812
##   25.0    3.238553        NaN  2.374812
##
## Tuning parameter 'alpha' was held constant at a value of 1
## RMSE was used to select the optimal model using the smallest value.
## The final values used for the model were alpha = 1 and lambda = 0.1.
```

```r
# what's our test RMSE? (RMSE is the average error in the unit of the thing we are predicting)
abalone_lasso_prediction <- predict(lasso,x_test)

# we can use this handy function:
postResample(pred = abalone_lasso_prediction, obs = abalone_test$age_in_rings)
```

```
##      RMSE  Rsquared       MAE
## 2.2677698 0.4980483 1.6158348
```