

A Machine Learning Book

Alfonso R. Reyes

2019-09-20

Contents

Prerequisites	11
I Algorithms Comparison	13
1 Introduction to h2o.	15
1.1 Bad Loans dataset. (<i>GLM, RF, GBM, DL, NB</i>)	15
1.2 Introduction	15
1.3 Algorithms	18
1.4 GLM	18
1.5 Random Forest	21
1.6 Gradient Boosting Machine (GBM)	26
1.7 Deep Learning	34
1.8 Naive Bayes model	41
2 Classification algorithms comparison. Diabetes dataset. (<i>CART, LDA, SVM, KNN, RF</i>)	45
2.1 PimaIndiansDiabetes dataset	45
2.2 Introduction	45
2.3 Workflow	45
2.4 Train the models using cross-validation	46
2.5 Compare models	47
2.6 Plot comparison	47
3 Multiclass classification comparison. Diabetes dataset. (<i>LDA, CART, KNN, SVM, RF</i>)	53
3.1 <i>iris</i> dataset	53
3.2 Introduction	53
3.3 Workflow	53
3.4 Peek at the dataset	54
3.5 Levels of the class	55
3.6 class distribution	55
3.7 Visualize the dataset	56
3.8 Evaluate algorithms	58
3.9 Make predictions	60
4 Regression algorithms comparison. Boston dataset. (<i>LM, GKM, GLMNET, SVM, CART, KNN</i>)	61
4.1 Boston dataset	61
4.2 Introduction	61
4.3 Workflow	62
4.4 Evaluation	71
4.5 Feature selection	73
4.6 Evaluate Algorithms: Box-Cox Transform	75

4.7	Tune SVM	77
4.8	Ensembling	79
4.9	Finalize the model	83
5	Classification algorithms comparison. Breast cancer dataset, (<i>LG, LDA, GLMNET, KNN, CART, NB, SVM</i>)	105
5.1	BreastCancer dataset	105
5.2	Introduction	105
5.3	Workflow	106
5.4	Inspect the dataset	107
5.5	clean up	111
5.6	Analyze the class variable	112
5.7	Unimodal visualization	113
5.8	Multimodal visualization	114
5.9	Algorithms Evaluation	116
5.10	Data transform	118
5.11	Tuning SVM	121
5.12	Tuning KNN	123
5.13	Ensemble	124
5.14	Finalize model	126
5.15	Prepare the validation set	126
6	Classification algorithms comparison. outbreaks dataset. (<i>RF, GLMNET, KNN, PDA, LDA, NSC, C5, PLS</i>)	129
6.1	Introduction	129
6.2	The data	131
6.3	Features	136
6.4	Test, train and validation data sets	144
6.5	Comparing Machine Learning algorithms	149
6.6	Comparing accuracy of models	162
6.7	Predicting unknown outcomes	165
6.8	Conclusions	175
II	Classification	177
7	A gentle introduction to support vector machines using R	179
7.1	Introduction	179
7.2	The rationale	179
7.3	The kernel trick	183
7.4	Support vector machines in R	184
7.5	SVM on <i>iris</i> dataset	184
7.6	SVM with Radial Basis Function kernel. Linear	186
7.7	SVM with Radial Basis Function kernel. Non-linear	188
7.8	Wrapping up	189
8	Classification with SVM. Social Network dataset	191
8.1	Introduction	191
8.2	Data Operations	191
9	Broad view of SVM	197
9.1	Introduction	197
9.2	Maximal Margin Classifier	198
9.3	Support Vector Classifiers	199
9.4	Support Vector Machines	202

9.5 SVMs for Multiple Classes	204
9.6 Application	206
10 Sonar Standalone Model with Random Forest	209
10.1 Introduction	209
10.2 Load libraries	209
10.3 Explore data	209
10.4 Apply tuning parameters for final model	212
10.5 Save model	213
10.6 Use the saved model	213
10.7 Make prediction with new data	213
11 Glass classification	215
12 Ozone SVM	219
13 A gentle introduction to support vector machines using R	221
13.1 Support vector machines in R	221
13.2 SVM on <code>iris</code> dataset	221
13.3 SVM with Radial Basis Function kernel. Linear	224
13.4 SVM with Radial Basis Function kernel. Non-linear	225
13.5 Wrapping up	226
14 SMS spam. Naive Bayes. Classification	229
14.1 Some conversion	230
14.2 Convert to Document Term Matrix (dtm)	231
14.3 split in training and test datasets	232
14.4 plot wordcloud	232
14.5 Limit Frequent words	235
14.6 Improve model performance	237
15 Classification Tree: Vehicle example	239
15.1 Load packages	239
15.2 Prepare data	240
15.3 Estimate the decision tree	240
15.4 Assess model	242
15.5 Make predictions	243
16 Bike sharing demand	245
16.1 Hypothesis Generation	246
16.2 Understanding the Data Set	246
16.3 Importing the dataset and Data Exploration	247
16.4 Hypothesis Testing (using multivariate analysis)	251
16.5 Feature Engineering	261
16.6 Model Building	269
16.7 End Notes	274
17 Breast Cancer Wisconsin	275
17.1 Read and process the data	275
17.2 Principal Component Analysis (PCA)	276
17.3 Feature importance	282
17.4 Feature Selection	283
17.5 Model comparison	287
17.6 Create comparison tables	292
17.7 Notes	293

18 Titanic with Naive-Bayes Classifier	295
19 Can we Do any Better?	297
20 Building a Naive Bayes Classifier in R	299
20.1 8. Building a Naive Bayes Classifier in R	299
III Feature Engineering	303
21 Employee attrition. Employee-Attrition dataset. <i>LIME</i> package	305
21.1 Introduction	305
21.2 Modeling Employee attrition	308
21.3 Model	309
21.4 Predict	309
21.5 Performance	310
21.6 The <i>lime</i> package	311
21.7 Feature Importance Visualization	313
21.8 Conclusions	319
22 Dealing with unbalanced data	321
22.1 Breast cancer dataset	321
22.2 Introduction	321
22.3 Read and process the data	321
22.4 Under-sampling	324
22.5 Oversampling	324
22.6 Predictions	327
22.7 Final notes	328
23 Ten different methods to assess Variable Importance	331
23.1 Glaucoma dataset	331
23.2 Introduction	331
23.3 1. Boruta	333
23.4 Variable Importance from Machine Learning Algorithms	335
23.5 Lasso Regression	337
23.6 Step wise Forward and Backward Selection	339
23.7 Relative Importance from Linear Regression	340
23.8 Recursive Feature Elimination (RFE)	341
23.9 Genetic Algorithm	343
23.10 Simulated Annealing	344
23.11 Information Value and Weights of Evidence	345
23.12 DALEX Package	347
23.13 Conclusion	349
24 Imputting missing values with Random Forest	351
24.1 Flu Prediction. <i>fluH7N9_china_2013</i> dataset	351
24.2 The data	352
24.3 Features	355
24.4 Imputing missing values	356
24.5 Test, train and validation data sets	358
24.6 Machine Learning algorithms	360
24.7 Comparing accuracy of models	361

IV Meta ML	365
25 PCA: prcomp vs princomp	367
25.1 General methods for principal component analysis	367
25.2 prcomp() and princomp() functions	367
25.3 factoextra	367
25.4 demo dataset	368
25.5 Compute PCA in R using prcomp()	368
25.6 Plots: quality and contribution	369
25.7 Access to the PCA results	371
25.8 Predict using PCA	375
25.9 Supplementary variables	377
25.10 Theory behind PCA results	379
26 Principal Components Methods	383
26.1 Data standardization	384
26.2 Eigenvalues / Variances	385
26.3 Graph of variables	386
26.4 Correlation circle	387
26.5 Quality of representation	388
26.6 Contributions of variables to PCs	391
26.7 Color by a custom continuous variable	395
26.8 Color by groups	395
26.9 Dimension description	396
26.10 Graph of individuals	397
26.11 Plots: quality and contribution	398
26.12 Color by a custom continuous variable	400
26.13 Color by groups	401
26.14 Graph customization	403
26.15 Size and shape of plot elements	405
26.16 Ellipses	406
26.17 Group mean points	407
26.18 Axis lines	408
26.19 Graphical parameters	408
26.20 Biplot	409
26.21 Supplementary elements	412
26.22 Quantitative variables	412
26.23 Individuals	416
26.24 Qualitative variables	417
26.25 Filtering results	418
26.26 Exporting results	420
26.27 Export results to txt/csv files	422
26.28 Summary	422
27 Biplot of the Iris data set	425
27.1 Iris: underlying principal components	426
27.2 Iris. Compute the eigenvectors and eigenvalues	427
28 Logistic Regression. Diabetes	429
28.1 Introduction	429
28.2 Exploring the data	429
28.3 Logistic regression with R	431
28.4 A second model	433
28.5 Classification rate and confusion matrix	434
28.6 Plots and decision boundaries	434

29 Sensitivity analysis for neural networks	439
29.1 Introduction	439
29.2 The Lek profile function	441
29.3 Getting a dataframe from <code>lek</code>	443
29.4 The <code>lek</code> function works with <code>lm</code>	443
29.5 <code>lek</code> function works with <code>RSNNS</code>	444
30 References	447
31 What is <code>.hat</code> in regression output	449
32 Q-Q normal to compare data to distributions	451
32.1 Introduction	451
32.2 Why we want to compare empirical vs theoretical distributions	452
32.3 The normal q-q plot	452
32.4 Using R's built-in functions	455
32.5 Using the <code>ggplot2</code> plotting environment	455
33 QQ and PP Plots	459
33.1 QQ Plot	459
33.2 Some Examples	461
33.3 Calibrating the Variability	462
33.4 Scalability	464
33.5 Comparing Two Distributions	465
33.6 PP Plots	466
33.7 Plots For Assessing Model Fit	469
34 Data Visualization: Working with models	471
34.1 Introduction	471
34.2 Show several fits at once, with a legend	473
34.3 Look inside model objects	475
34.4 Get model-based graphics right	477
34.5 Generate predictions to graph	478
34.6 Tidy model objects with <code>broom</code>	480
34.7 Grouped analysis and list-columns	486
34.8 Plot marginal effects	489
34.9 Plots from complex surveys	493
34.10 Where to go next	497
35 Visualizing residuals	501
35.1 Simple Linear Regression	502
35.2 Step 4: use residuals to adjust	504
V Neural Networks	509
36 Building deep neural nets with <code>h2o</code> that predict arrhythmia of the heart	511
36.1 Introduction	511
36.2 Arrhythmia data	513
36.3 Converting the dataframe to a <code>h2o</code> object	517
36.4 Training, test and validation data	519
36.5 Modeling	521
36.6 Model performance	524
36.7 Test data	533
36.8 Final conclusions: How useful is the model?	538

37 Credit Scoring	541
37.1 Introduction	541
37.2 Motivation	541
37.3 load the data	541
37.4 Objective	544
37.5 Steps	544
37.6 Test the neural network	545
38 Build a fully connected neural network from scratch	547
38.1 Introduction	547
39 Wine with neuralnet	557
39.1 The dataset	557
39.2 Preprocessing	566
39.3 Fitting the model with neuralnet	567
39.4 Cross validating the classifier	568
40 Classification and Regression with H2O Deep Learning	571
40.1 Introduction	571
40.2 H2O R Package	571
40.3 Start H2O	572
40.4 Let's have some fun first: Decision Boundaries	574
40.5 Cover Type Dataset	577
40.6 Regression and Binary Classification	620
40.7 Unsupervised Anomaly detection	621
40.8 H2O Deep Learning Tips & Tricks	622
40.9 All done, shutdown H2O	630
41 Regression with ANN - Yacht Hydrodynamics	631
41.1 Introduction	631
41.2 Replication Requirements	631
41.3 Data Preparation	632
41.4 1st Regression ANN	634
41.5 Regression Hyperparameters	635
41.6 Wrapping Up	637
42 Regression - cereals dataset	639
42.1 Introduction	639
42.2 The Basics of Neural Networks	639
42.3 Fitting a Neural Network in R	640
42.4 End Notes	644
43 Fitting a neural network	645
43.1 Introduction	645
43.2 The dataset	645
43.3 Preparing to fit the neural network	647
43.4 Parameters	647
43.5 Predicting medv using the neural network	648
43.6 A (fast) cross validation	649
43.7 A final note on model interpretability	651
44 Visualization of neural networks	653
44.1 caret and plot NN	657
44.2 Multiple hidden layers	658
44.3 Binary predictors	659

44.4 color coding the input layer	660
VI Linear Regression	663
45 Temperature modeling using nested dataframes	665
45.1 Prepare the data	665
45.2 Define the models	668
45.3 Test modeling on one dataset	670
45.4 Making a nested dataframe	671
45.5 Apply multiple models on a nested structure	674
45.6 Using broom package to look at model-statistics	679
46 Linear Regression. World Happiness	681
46.1 Introduction	681
46.2 A quick exploration of the data	681
46.3 Linear regression with R	684
46.4 Regression summary	684
46.5 Regression analysis	686
46.6 Analysis of collinearity	687
46.7 What drives happiness	688
47 Linear Regression on Advertising	689
48 Lab 3A: Regression. iris dataset	695
48.1 Introduction	695
48.2 Explore the Data	695
48.3 Create Training and Test Sets	697
48.4 Predict with Simple Linear Regression	697
48.5 Predict with Multiple Regression	699
48.6 5. Predict with Neural Network Regression	700
48.7 6. Evaluate all the regression Models	703
49 Regression 3b. Rates dataset. (SLR, MLR, NN)	705
49.1 Introduction	705
49.2 Split the Data into Test and Training Sets	707
49.3 Predict with Simple Linear Regression	708
49.4 Predict with Multiple Linear Regression	709
49.5 Predict with Neural Network Regression	710
49.6 Evaluate the Regression Models	712
50 Regression Boston nnet	715
50.1 Neural Network	717
50.2 Linear Regression	718
51 Comparing Multiple vs. Neural Network Regression	721
51.1 Introduction	721
51.2 Multiple Regression	722
51.3 Neural Network	727

Prerequisites

This is a *sample* book written in **Markdown**. You can use anything that Pandoc's Markdown supports, e.g., a math equation $a^2 + b^2 = c^2$.

The **bookdown** package can be installed from CRAN or Github:

```
install.packages("bookdown")
# or the development version
# devtools::install_github("rstudio/bookdown")
```


Part I

Algorithms Comparison

Chapter 1

Introduction to h2o.

1.1 Bad Loans dataset. (*GLM, RF, GBM, DL, NB*)

Source: <https://github.com/h2oai/h2o-tutorials/blob/master/h2o-open-tour-2016/chicago/intro-to-h2o.R>

1.2 Introduction

Introductory H2O Machine Learning Tutorial Prepared for H2O Open Chicago 2016: <http://open.h2o.ai/chicago.html>

1.2.1 Install and download h2o

First step is to download & install the h2o R library. The latest version is available by clicking on the R tab here: http://h2o-release.s3.amazonaws.com/h2o/latest_stable.html

Load the H2O library and start up the H2O cluster locally on your machine:

```
# Load the H2O library and start up the H2O cluster locally on your machine
library(h2o)
h2o.init(nthreads = -1, #Number of threads -1 means use all cores on your machine
          max_mem_size = "8G") #max mem size is the maximum memory to allocate to H2O
#>
#> H2O is not running yet, starting it now...
#>
#> Note: In case of errors look at the following log files:
#>       /tmp/RtmpBLLJK4/h2o_datascience_started_from_r.out
#>       /tmp/RtmpBLLJK4/h2o_datascience_started_from_r.err
#>
#>
#> Starting H2O JVM and connecting: . Connection successful!
#>
#> R is connected to the H2O cluster:
#>   H2O cluster uptime:      1 seconds 326 milliseconds
#>   H2O cluster timezone:    America/Chicago
#>   H2O data parsing timezone: UTC
#>   H2O cluster version:    3.22.1.1
```

```
#> H2O cluster version age: 8 months and 23 days !!!
#> H2O cluster name: H2O_started_from_R_datascience_mwl453
#> H2O cluster total nodes: 1
#> H2O cluster total memory: 7.11 GB
#> H2O cluster total cores: 8
#> H2O cluster allowed cores: 8
#> H2O cluster healthy: TRUE
#> H2O Connection ip: localhost
#> H2O Connection port: 54321
#> H2O Connection proxy: NA
#> H2O Internal Security: FALSE
#> H2O API Extensions: XGBoost, Algos, AutoML, Core V3, Core V4
#> R Version: R version 3.6.0 (2019-04-26)
```

1.2.2 Load the dataset

Next we will import a cleaned up version of the Lending Club “Bad Loans” dataset. The purpose here is to predict whether a loan will be bad (not repaid to the lender). The response column, `bad_loan`, is 1 if the loan was bad, and 0 otherwise.

Import the data `loan_csv <- "/Volumes/H2OTOUR/loan.csv"`

Alternatively, you can import the data directly from a URL

```
# modify this for your machine
loan_csv <- "https://raw.githubusercontent.com/h2oai/app-consumer-loan/master/data/loan.csv"
data <- h2o.importFile(loan_csv) # 163,987 rows x 15 columns
#>
| |
|-----| 0%
|-----| 22%
|-----| 47%
|-----| 59%
|-----| 84%
|-----| 97%
|-----| 100%
dim(data)
#> [1] 163987      15
# [1] 163987      15

url <- "https://raw.githubusercontent.com/h2oai/app-consumer-loan/master/data/loan.csv"
loans <- read.csv(url)
write.csv(loans, file = file.path(data_raw_dir, "loan.csv"))
```

1.2.3 Feature Engineering

Since we want to train a binary classification model, we must ensure that the response is coded as a factor. If the response is 0/1, H2O will assume it's numeric, which means that H2O will train a regression model.

instead

```
data$bad_loan <- as.factor(data$bad_loan) #encode the binary response as a factor
h2o.levels(data$bad_loan) #optional: after encoding, this shows the two factor levels, '0' and '1'
#> [1] "0" "1"
# [1] "0" "1"
```

1.2.4 Partition data

Partition the data into training, validation and test sets

```
# Partition the data into training, validation and test sets
splits <- h2o.splitFrame(data = data,
                         ratios = c(0.7, 0.15), #partition data into 70%, 15%, 15% chunks
                         seed = 1) #setting a seed will guarantee reproducibility
train <- splits[[1]]
valid <- splits[[2]]
test <- splits[[3]]
```

Take a look at the size of each partition Notice that h2o.splitFrame uses approximate splitting not exact splitting (for efficiency) so these are not exactly 70%, 15% and 15% of the total rows

```
nrow(train) # 114908
#> [1] 114908
nrow(valid) # 24498
#> [1] 24498
nrow(test) # 24581
#> [1] 24581
```

1.2.5 Identify response and predictor variables

```
# Identify response and predictor variables
y <- "bad_loan"
x <- setdiff(names(data), c(y, "int_rate")) # remove the interest rate column because it's correlated to bad_loan
print(x)
#> [1] "loan_amnt"           "term"
#> [3] "emp_length"          "home_ownership"
#> [5] "annual_inc"          "purpose"
#> [7] "addr_state"          "dti"
#> [9] "delinq_2yrs"         "revol_util"
#> [11] "total_acc"          "longest_credit_length"
#> [13] "verification_status"
#> [1] "loan_amnt"           "term"
#> [3] "emp_length"          "home_ownership"
#> [5] "annual_inc"          "verification_status"
#> [7] "purpose"              "addr_state"
#> [9] "dti"                  "delinq_2yrs"
#> [11] "revol_util"          "total_acc"
#> [13] "longest_credit_length"
```

1.3 Algorithms

Now that we have prepared the data, we can train some models. We will start by training a single model from each of the H2O supervised algos:

1. Generalized Linear Model (GLM)
2. Random Forest (RF)
3. Gradient Boosting Machine (GBM)
4. Deep Learning (DL)
5. Naive Bayes (NB)

1.4 GLM

Let's start with a basic binomial Generalized Linear Model. By default, h2o.glm uses a regularized, elastic net model

```
glm_fit1 <- h2o.glm(x = x,
                      y = y,
                      training_frame = train,
                      model_id = "glm_fit1",
                      family = "binomial") #similar to R's glm, h2o.glm has the family argument
#>
| | | | | 0%
| | | | |=====
| | | | | 100%
```

Next we will do some automatic tuning by passing in a validation frame and setting `lambda_search = True`. Since we are training a GLM with regularization, we should try to find the right amount of regularization (to avoid overfitting). The model parameter, `lambda`, controls the amount of regularization in a GLM model and we can find the optimal value for `lambda` automatically by setting `lambda_search = TRUE` and passing in a validation frame (which is used to evaluate model performance using a particular value of `lambda`).

```
glm_fit2 <- h2o.glm(x = x,
                      y = y,
                      training_frame = train,
                      model_id = "glm_fit2",
                      validation_frame = valid,
                      family = "binomial",
                      lambda_search = TRUE)
#>
| | | | | 0%
| | | | |=====
| | | | | 29%
| | | | |=====
| | | | | 57%
| | | | |=====
| | | | | 100%
```

Let's compare the performance of the two GLMs

```
# Let's compare the performance of the two GLMs
glm_perf1 <- h2o.performance(model = glm_fit1,
                               newdata = test)
```

```

glm_perf2 <- h2o.performance(model = glm_fit2,
                             newdata = test)

# Print model performance
glm_perf1
#> H2OBinomialMetrics: glm
#>
#> MSE: 0.142
#> RMSE: 0.377
#> LogLoss: 0.451
#> Mean Per-Class Error: 0.37
#> AUC: 0.677
#> pr_auc: 0.327
#> Gini: 0.355
#> R^2: 0.0639
#> Residual Deviance: 22176
#> AIC: 22280
#>
#> Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
#>          0    1   Error      Rate
#> 0  13647 6344 0.317343  =6344/19991
#> 1   1939 2651 0.422440  =1939/4590
#> Totals 15586 8995 0.336968  =8283/24581
#>
#> Maximum Metrics: Maximum metrics at their respective thresholds
#>                   metric threshold     value idx
#> 1                  max f1  0.193323 0.390283 222
#> 2                  max f2  0.118600 0.556655 307
#> 3                  max f0point5 0.276272 0.354086 146
#> 4                  max accuracy 0.494244 0.814410  29
#> 5                  max precision 0.744500 1.000000    0
#> 6                  max recall   0.001225 1.000000 399
#> 7                  max specificity 0.744500 1.000000    0
#> 8                  max absolute_mcc 0.198334 0.210606 216
#> 9  max min_per_class_accuracy 0.180070 0.627783 236
#> 10 max mean_per_class_accuracy 0.193323 0.630109 222
#>
#> Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T>)
glm_perf2
#> H2OBinomialMetrics: glm
#>
#> MSE: 0.142
#> RMSE: 0.377
#> LogLoss: 0.451
#> Mean Per-Class Error: 0.372
#> AUC: 0.677
#> pr_auc: 0.326
#> Gini: 0.354
#> R^2: 0.0635
#> Residual Deviance: 22186
#> AIC: 22282
#>
#> Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:

```

```

#>          0    1    Error      Rate
#> 0     13699 6292 0.314742 =6292/19991
#> 1      1968 2622 0.428758 =1968/4590
#> Totals 15667 8914 0.336032 =8260/24581
#>
#> Maximum Metrics: Maximum metrics at their respective thresholds
#>                         metric threshold   value idx
#> 1                      max f1 0.194171 0.388329 216
#> 2                      max f2 0.119200 0.555998 306
#> 3                      max f0point5 0.256488 0.351893 153
#> 4                      max accuracy 0.474001 0.814654 32
#> 5                      max precision 0.736186 1.000000 0
#> 6                      max recall 0.001255 1.000000 399
#> 7                      max specificity 0.736186 1.000000 0
#> 8                      max absolute_mcc 0.198114 0.208337 212
#> 9  max min_per_class_accuracy 0.180131 0.625181 231
#> 10 max mean_per_class_accuracy 0.194171 0.628250 216
#>
#> Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T>)

```

Instead of printing the entire model performance metrics object, it is probably easier to print just the metric that you are interested in comparing. Retreive test set AUC

```

h2o.auc(glm_perf1) #0.677449084114
#> [1] 0.677
h2o.auc(glm_perf2) #0.677675858276
#> [1] 0.677

```

Compare test AUC to the training AUC and validation AUC

```

# Compare test AUC to the training AUC and validation AUC
h2o.auc(glm_fit2, train = TRUE) #0.674306164325
#> [1] 0.673
h2o.auc(glm_fit2, valid = TRUE) #0.675512216705
#> [1] 0.675
glm_fit2@model$validation_metrics #0.675512216705
#> H2OBinomialMetrics: glm
#> ** Reported on validation data. **
#>
#> MSE: 0.142
#> RMSE: 0.377
#> LogLoss: 0.451
#> Mean Per-Class Error: 0.37
#> AUC: 0.675
#> pr_auc: 0.316
#> Gini: 0.351
#> R^2: 0.0597
#> Residual Deviance: 22101
#> AIC: 22197
#>
#> Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
#>          0    1    Error      Rate
#> 0     13591 6365 0.318952 =6365/19956
#> 1      1916 2626 0.421841 =1916/4542
#> Totals 15507 8991 0.338028 =8281/24498

```

```

#> #> Maximum Metrics: Maximum metrics at their respective thresholds
#> #>           metric threshold     value idx
#> 1             max f1    0.193519 0.388088 217
#> 2             max f2    0.116436 0.555055 308
#> 3             max f0point5 0.288405 0.343386 132
#> 4             max accuracy 0.487882 0.815250 29
#> 5             max precision 0.576333 0.681818 9
#> 6             max recall   0.004789 1.000000 398
#> 7             max specificity 0.715719 0.999950 0
#> 8             max absolute_mcc 0.195417 0.209494 215
#> 9 max min_per_class_accuracy 0.180760 0.627731 230
#> 10 max mean_per_class_accuracy 0.192639 0.629672 218
#>
#> #> Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T/
```

1.5 Random Forest

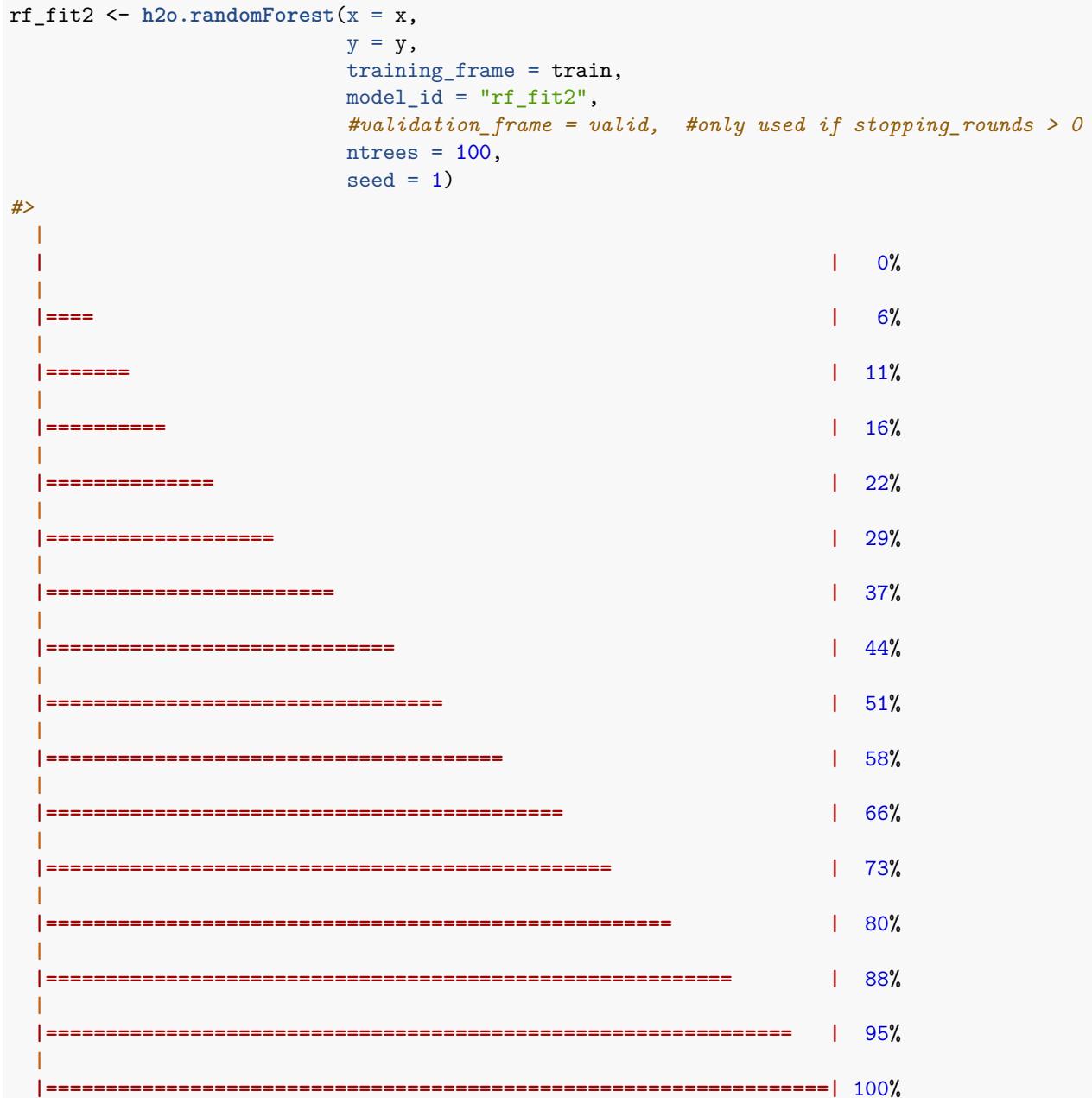
H2O's Random Forest (RF) implements a distributed version of the standard Random Forest algorithm and variable importance measures. First we will train a basic Random Forest model with default parameters. The Random Forest model will infer the response distribution from the response encoding. A seed is required for reproducibility.

```
rf_fit1 <- h2o.randomForest(x = x,
                            y = y,
                            training_frame = train,
                            model_id = "rf_fit1",
                            seed = 1)

#>
#> [=====] 0%
#> [====] 4%
#> [=====] 12%
#> [=====] 22%
#> [=====] 32%
#> [=====] 46%
#> [=====] 60%
#> [=====] 76%
#> [=====] 88%
#> [=====] 100%
```

Next we will increase the number of trees used in the forest by setting `ntrees = 100`. The default number of trees in an H2O Random Forest is 50, so this RF will be twice as big as the default. Usually increasing the number of trees in a RF will increase performance as well. Unlike Gradient Boosting Machines (GBMs),

Random Forests are fairly resistant (although not free from) overfitting. See the GBM example below for additional guidance on preventing overfitting using H2O's early stopping functionality.



Let's compare the performance of the two RFs

```

# Let's compare the performance of the two RFs
rf_perf1 <- h2o.performance(model = rf_fit1,
                             newdata = test)
rf_perf2 <- h2o.performance(model = rf_fit2,
                             newdata = test)

# Print model performance
rf_perf1
#> H2OBinomialMetrics: drf

```

```

#>
#> MSE: 0.144
#> RMSE: 0.379
#> LogLoss: 0.459
#> Mean Per-Class Error: 0.379
#> AUC: 0.663
#> pr_auc: 0.311
#> Gini: 0.327
#>
#> Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
#>          0   1   Error      Rate
#> 0    12842 7149 0.357611  =7149/19991
#> 1     1839 2751 0.400654  =1839/4590
#> Totals 14681 9900 0.365648  =8988/24581
#>
#> Maximum Metrics: Maximum metrics at their respective thresholds
#>           metric threshold      value idx
#> 1             max f1 0.193397 0.379710 229
#> 2             max f2 0.077797 0.547982 344
#> 3             max f0point5 0.277524 0.344251 158
#> 4             max accuracy 0.543639 0.813799 29
#> 5             max precision 0.817454 1.000000 0
#> 6             max recall 0.001181 1.000000 399
#> 7             max specificity 0.817454 1.000000 0
#> 8             max absolute_mcc 0.252480 0.195090 178
#> 9 max min_per_class_accuracy 0.186549 0.619826 235
#> 10 max mean_per_class_accuracy 0.192603 0.620890 230
#>
#> Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T>,
rf_perf2
#> H2OBinomialMetrics: drf
#>
#> MSE: 0.143
#> RMSE: 0.378
#> LogLoss: 0.454
#> Mean Per-Class Error: 0.377
#> AUC: 0.669
#> pr_auc: 0.32
#> Gini: 0.339
#>
#> Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
#>          0   1   Error      Rate
#> 0    13172 6819 0.341103  =6819/19991
#> 1     1891 2699 0.411983  =1891/4590
#> Totals 15063 9518 0.354339  =8710/24581
#>
#> Maximum Metrics: Maximum metrics at their respective thresholds
#>           metric threshold      value idx
#> 1             max f1 0.196407 0.382620 225
#> 2             max f2 0.092270 0.549691 331
#> 3             max f0point5 0.291396 0.349342 144
#> 4             max accuracy 0.555908 0.813840 20
#> 5             max precision 0.651522 0.785714 5

```

```
#> 6           max recall  0.004212 1.000000 398
#> 7           max specificity 0.711667 0.999950   0
#> 8           max absolute_mcc 0.229251 0.204236 194
#> 9   max min_per_class_accuracy 0.184594 0.619829 235
#> 10  max mean_per_class_accuracy 0.196407 0.623457 225
#>
#> Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T/
```

Retreive test set AUC

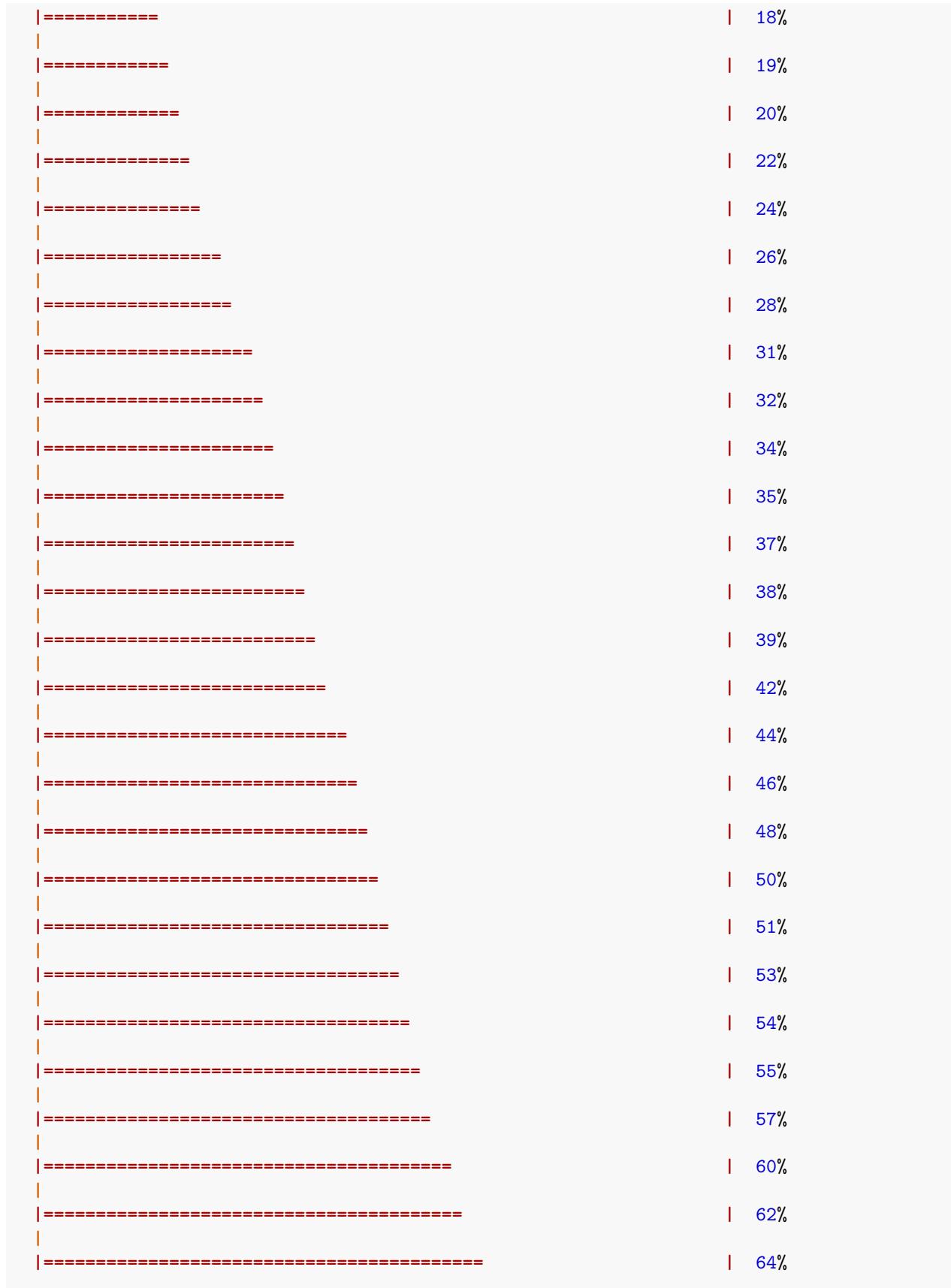
```
h2o.auc(rf_perf1) # 0.662266990734
#> [1] 0.663
h2o.auc(rf_perf2) # 0.66525468051
#> [1] 0.669
```

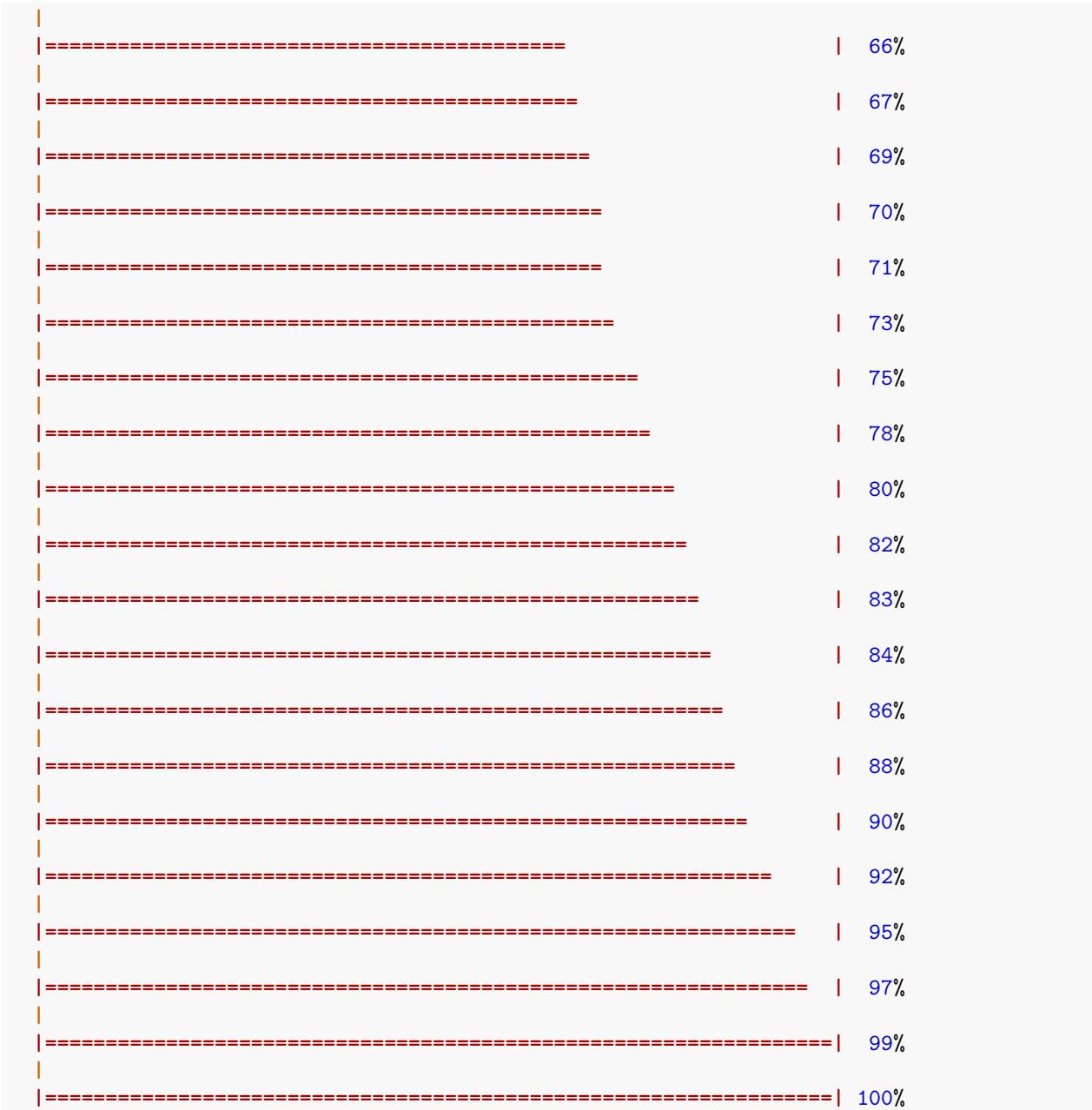
1.5.1 Cross-validate performance

Rather than using held-out test set to evaluate model performance, a user may wish to estimate model performance using cross-validation. Using the RF algorithm (with default model parameters) as an example, we demonstrate how to perform k-fold cross-validation using H2O. No custom code or loops are required, you simply specify the number of desired folds in the nfolds argument. Since we are not going to use a test set here, we can use the original (full) dataset, which we called data rather than the subsampled **train** dataset. Note that this will take approximately k (nfolds) times longer than training a single RF model, since it will train k models in the cross-validation process (trained on $n(k-1)/k$ rows), in addition to the final model trained on the full training_frame dataset with n rows.

```
rf_fit3 <- h2o.randomForest(x = x,
                             y = y,
                             training_frame = train,
                             model_id = "rf_fit3",
                             seed = 1,
                             nfolds = 5)

#>
| |
| = | 0%
| == | 1%
| === | 2%
| === | 4%
| === | 5%
| ===== | 7%
| ===== | 10%
| ===== | 12%
| ===== | 14%
| ===== | 16%
```





To evaluate the cross-validated AUC, do the following:

```
# To evaluate the cross-validated AUC, do the following:
h2o.auc(rf_fit3, xval = TRUE) # 0.661201482614
#> [1] 0.659
```

1.6 Gradient Boosting Machine (GBM)

H2O's Gradient Boosting Machine (GBM) offers a Stochastic GBM, which can increase performance quite a bit compared to the original GBM implementation.

Now we will train a basic GBM model. The GBM model will infer the response distribution from the re-

sponse encoding if not specified explicitly through the `distribution` argument. A seed is required for reproducibility.

```
gbm_fit1 <- h2o.gbm(x = x,
                      y = y,
                      training_frame = train,
                      model_id = "gbm_fit1",
                      seed = 1)
#>
|-----| 0%
|-----| 20%
|-----| 44%
|-----| 70%
|-----| 96%
|-----| 100%
```

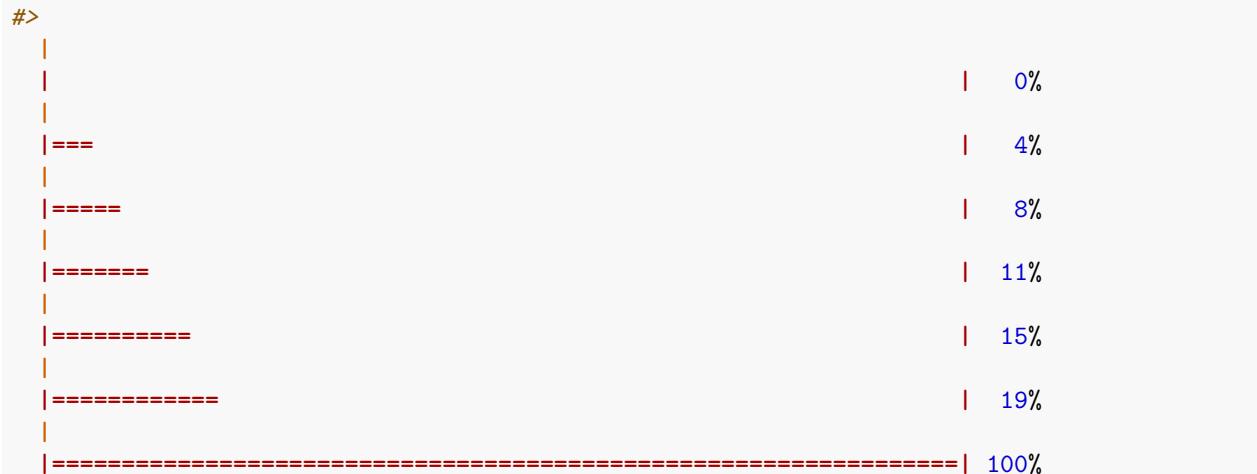
Next we will increase the number of trees used in the GBM by setting `ntrees=500`. The default number of trees in an H2O GBM is 50, so this GBM will trained using ten times the default. Increasing the number of trees in a GBM is one way to increase performance of the model, however, you have to be careful not to overfit your model to the training data by using too many trees. To automatically find the optimal number of trees, you must use H2O's early stopping functionality. This example will not do that, however, the following # example will.

```
gbm_fit2 <- h2o.gbm(x = x,
                      y = y,
                      training_frame = train,
                      model_id = "gbm_fit2",
                      #validation_frame = valid, #only used if stopping_rounds > 0
                      ntrees = 500,
                      seed = 1)
#>
|-----| 0%
|---| 3%
|---| 5%
|---| 8%
|---| 11%
|---| 16%
|---| 21%
|---| 29%
|-----| 48%
```



We will again set `ntrees = 500`, however, this time we will use early stopping in order to prevent overfitting (from too many trees). All of H2O's algorithms have early stopping available, however early stopping is not enabled by default (with the exception of Deep Learning). There are several parameters that should be used to control early stopping. The three that are common to all the algorithms are: `stopping_rounds`, `stopping_metric` and `stopping_tolerance`. The stopping metric is the metric by which you'd like to measure performance, and so we will choose AUC here. The `score_tree_interval` is a parameter specific to the Random Forest model and the GBM. Setting `score_tree_interval = 5` will score the model after every five trees. The parameters we have set below specify that the model will stop training after there have been three scoring intervals where the AUC has not increased more than 0.0005. Since we have specified a validation frame, the stopping tolerance will be computed on validation AUC rather than training AUC.

```
gbm_fit3 <- h2o.gbm(x = x,
                      y = y,
                      training_frame = train,
                      model_id = "gbm_fit3",
                      validation_frame = valid, #only used if stopping_rounds > 0
                      ntrees = 500,
                      score_tree_interval = 5,      #used for early stopping
                      stopping_rounds = 3,          #used for early stopping
                      stopping_metric = "AUC",      #used for early stopping
                      stopping_tolerance = 0.0005,   #used for early stopping
                      seed = 1)
```



Let's compare the performance of the two GBMs

```
# Let's compare the performance of the two GBMs
gbm_perf1 <- h2o.performance(model = gbm_fit1,
                               newdata = test)
gbm_perf2 <- h2o.performance(model = gbm_fit2,
                               newdata = test)
gbm_perf3 <- h2o.performance(model = gbm_fit3,
                               newdata = test)
```

```

# Print model performance
gbm_perf1
#> H2OBinomialMetrics: gbm
#>
#> MSE: 0.141
#> RMSE: 0.376
#> LogLoss: 0.448
#> Mean Per-Class Error: 0.367
#> AUC: 0.684
#> pr_auc: 0.332
#> Gini: 0.368
#>
#> Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
#>          0      1   Error      Rate
#> 0    12143  7848 0.392577 =7848/19991
#> 1     1571  3019 0.342266 =1571/4590
#> Totals 13714 10867 0.383182 =9419/24581
#>
#> Maximum Metrics: Maximum metrics at their respective thresholds
#>           metric threshold      value idx
#> 1             max f1  0.171139 0.390632 251
#> 2             max f2  0.108885 0.560103 328
#> 3             max f0point5 0.285149 0.356430 145
#> 4             max accuracy 0.510077 0.814410 27
#> 5             max precision 0.601699 0.636364 8
#> 6             max recall  0.037543 1.000000 398
#> 7             max specificity 0.719189 0.999950 0
#> 8             max absolute_mcc 0.220471 0.215401 199
#> 9  max min_per_class_accuracy 0.176689 0.628540 244
#> 10 max mean_per_class_accuracy 0.170225 0.632624 252
#>
#> Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T/F>)
gbm_perf2
#> H2OBinomialMetrics: gbm
#>
#> MSE: 0.142
#> RMSE: 0.376
#> LogLoss: 0.449
#> Mean Per-Class Error: 0.367
#> AUC: 0.684
#> pr_auc: 0.329
#> Gini: 0.368
#>
#> Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
#>          0      1   Error      Rate
#> 0    13661  6330 0.316642 =6330/19991
#> 1     1918  2672 0.417865 =1918/4590
#> Totals 15579  9002 0.335544 =8248/24581
#>
#> Maximum Metrics: Maximum metrics at their respective thresholds
#>           metric threshold      value idx
#> 1             max f1  0.189615 0.393172 234
#> 2             max f2  0.096969 0.558918 333

```

```

#> 3          max f0point5  0.2787716 0.359560 160
#> 4          max accuracy 0.521287 0.814287 37
#> 5          max precision 0.901295 1.000000 0
#> 6          max recall   0.018504 1.000000 398
#> 7          max specificity 0.901295 1.000000 0
#> 8          max absolute_mcc 0.231089 0.217255 196
#> 9  max min_per_class_accuracy 0.174914 0.630834 249
#> 10 max mean_per_class_accuracy 0.156944 0.633792 267
#>
#> Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T>)
gbm_perf3
#> H2OBinomialMetrics: gbm
#>
#> MSE: 0.141
#> RMSE: 0.376
#> LogLoss: 0.448
#> Mean Per-Class Error: 0.367
#> AUC: 0.684
#> pr_auc: 0.331
#> Gini: 0.369
#>
#> Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
#>      0    1    Error      Rate
#> 0  13652 6339 0.317093 =6339/19991
#> 1   1917 2673 0.417647 =1917/4590
#> Totals 15569 9012 0.335869 =8256/24581
#>
#> Maximum Metrics: Maximum metrics at their respective thresholds
#>                  metric threshold      value idx
#> 1          max f1  0.189870 0.393030 234
#> 2          max f2  0.109837 0.559580 321
#> 3          max f0point5 0.294571 0.356508 148
#> 4          max accuracy 0.510869 0.814572 40
#> 5          max precision 0.797620 1.000000 0
#> 6          max recall   0.026373 1.000000 397
#> 7          max specificity 0.797620 1.000000 0
#> 8          max absolute_mcc 0.231530 0.218255 197
#> 9  max min_per_class_accuracy 0.176566 0.631808 248
#> 10 max mean_per_class_accuracy 0.175468 0.633400 249
#>
#> Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T>)

# Retreive test set AUC
h2o.auc(gbm_perf1) # 0.682765594191
#> [1] 0.684
h2o.auc(gbm_perf2) # 0.671854616713
#> [1] 0.684
h2o.auc(gbm_perf3) # 0.68309902855
#> [1] 0.684

```

To examine the scoring history, use the `scoring_history` method on a trained model. If `score_tree_interval` is not specified, it will score at various intervals, as we can see for `h2o.scoreHistory()` below. However, regular 5-tree intervals are used for `h2o.scoreHistory()`. The `gbm_fit2` was trained only using a training set (no validation set), so the scoring history is calculated for training set performance metrics only.

```

h2o.scoreHistory(gbm_fit2)
#> Scoring History:
#>   timestamp duration number_of_trees training_rmse
#> 1 2019-09-20 14:19:45 0.002 sec          0 0.38563
#> 2 2019-09-20 14:19:45 0.146 sec          1 0.38370
#> 3 2019-09-20 14:19:45 0.211 sec          2 0.38206
#> 4 2019-09-20 14:19:45 0.258 sec          3 0.38069
#> 5 2019-09-20 14:19:45 0.299 sec          4 0.37954
#>   training_logloss training_auc training_pr_auc training_lift
#> 1          0.47403    0.50000    0.00000  1.00000
#> 2          0.46913    0.65779    0.30174  2.68330
#> 3          0.46512    0.66583    0.31164  2.79399
#> 4          0.46184    0.66851    0.31500  2.97100
#> 5          0.45912    0.67011    0.31822  2.97544
#>   training_classification_error
#> 1          0.81825
#> 2          0.40069
#> 3          0.33325
#> 4          0.34475
#> 5          0.33180
#>
#> ---
#>   timestamp duration number_of_trees training_rmse
#> 48 2019-09-20 14:19:49 3.719 sec         47 0.36727
#> 49 2019-09-20 14:19:49 3.805 sec         48 0.36717
#> 50 2019-09-20 14:19:49 3.886 sec         49 0.36709
#> 51 2019-09-20 14:19:49 3.967 sec         50 0.36699
#> 52 2019-09-20 14:19:53 7.983 sec        223 0.36129
#> 53 2019-09-20 14:19:56 10.802 sec       500 0.36129
#>   training_logloss training_auc training_pr_auc training_lift
#> 48          0.43065    0.71457    0.37309  3.66973
#> 49          0.43044    0.71493    0.37382  3.65538
#> 50          0.43025    0.71524    0.37443  3.68408
#> 51          0.43002    0.71568    0.37493  3.66016
#> 52          0.41770    0.74024    0.41866  4.25823
#> 53          0.41770    0.74024    0.41866  4.25823
#>   training_classification_error
#> 48          0.28870
#> 49          0.30103
#> 50          0.30056
#> 51          0.30179
#> 52          0.25719
#> 53          0.25719

```

When early stopping is used, we see that training stopped at 105 trees instead of the full 500. Since we used a validation set in `gbm_fit3`, both training and validation performance metrics are stored in the scoring history object. Take a look at the validation AUC to observe that the correct stopping tolerance was enforced.

```

h2o.scoreHistory(gbm_fit3)
#> Scoring History:
#>   timestamp duration number_of_trees training_rmse
#> 1 2019-09-20 14:19:56 0.004 sec          0 0.38563
#> 2 2019-09-20 14:19:57 0.288 sec          5 0.37853
#> 3 2019-09-20 14:19:57 0.541 sec         10 0.37508

```

```

#> 4 2019-09-20 14:19:57 0.811 sec      15 0.37307
#> 5 2019-09-20 14:19:57 1.076 sec      20 0.37152
#> 6 2019-09-20 14:19:58 1.336 sec      25 0.37041
#> 7 2019-09-20 14:19:58 1.597 sec      30 0.36947
#> 8 2019-09-20 14:19:58 1.839 sec      35 0.36877
#> 9 2019-09-20 14:19:59 2.102 sec      40 0.36808
#> 10 2019-09-20 14:19:59 2.351 sec      45 0.36748
#> 11 2019-09-20 14:19:59 2.748 sec      50 0.36699
#> 12 2019-09-20 14:19:59 3.007 sec      55 0.36651
#> 13 2019-09-20 14:20:00 3.266 sec      60 0.36607
#> 14 2019-09-20 14:20:00 3.519 sec      65 0.36574
#> 15 2019-09-20 14:20:00 3.785 sec      70 0.36531
#> 16 2019-09-20 14:20:00 4.038 sec      75 0.36503
#> 17 2019-09-20 14:20:01 4.302 sec      80 0.36460
#> 18 2019-09-20 14:20:01 4.571 sec      85 0.36426
#> 19 2019-09-20 14:20:01 4.828 sec      90 0.36394
#> 20 2019-09-20 14:20:02 5.116 sec      95 0.36362
#>   training_logloss training_auc training_pr_auc training_lift
#> 1      0.47403    0.50000    0.00000  1.00000
#> 2      0.45676    0.67362    0.32194  3.04518
#> 3      0.44884    0.68117    0.33330  3.19128
#> 4      0.44424    0.68708    0.34105  3.30132
#> 5      0.44060    0.69498    0.34930  3.46878
#> 6      0.43800    0.69983    0.35479  3.47356
#> 7      0.43578    0.70425    0.36014  3.48792
#> 8      0.43410    0.70746    0.36373  3.53576
#> 9      0.43252    0.71082    0.36837  3.61710
#> 10     0.43116    0.71346    0.37174  3.66495
#> 11     0.43002    0.71568    0.37493  3.66016
#> 12     0.42899    0.71765    0.37852  3.70801
#> 13     0.42801    0.71953    0.38157  3.73193
#> 14     0.42726    0.72093    0.38411  3.78934
#> 15     0.42632    0.72277    0.38733  3.82762
#> 16     0.42572    0.72396    0.38955  3.85154
#> 17     0.42477    0.72596    0.39282  3.95202
#> 18     0.42407    0.72713    0.39550  3.99508
#> 19     0.42338    0.72852    0.39819  4.02857
#> 20     0.42272    0.72984    0.40069  4.02378
#>   training_classification_error validation_rmse validation_logloss
#> 1      0.81825    0.38864    0.47953
#> 2      0.32117    0.38233    0.46398
#> 3      0.32202    0.37958    0.45742
#> 4      0.32027    0.37828    0.45428
#> 5      0.33371    0.37739    0.45210
#> 6      0.32537    0.37676    0.45053
#> 7      0.29722    0.37636    0.44949
#> 8      0.29544    0.37604    0.44866
#> 9      0.28871    0.37587    0.44818
#> 10     0.30181    0.37574    0.44781
#> 11     0.30179    0.37560    0.44744
#> 12     0.29464    0.37552    0.44718
#> 13     0.30343    0.37547    0.44703
#> 14     0.28692    0.37543    0.44694

```

```

#> 15          0.28579   0.37536   0.44676
#> 16          0.26903   0.37536   0.44673
#> 17          0.28476   0.37534   0.44664
#> 18          0.26950   0.37537   0.44671
#> 19          0.27036   0.37538   0.44671
#> 20          0.26573   0.37540   0.44673
#> validation_auc validation_pr_auc validation_lift
#> 1      0.50000   0.00000   1.00000
#> 2      0.66168   0.30421   2.75098
#> 3      0.66767   0.30959   2.68582
#> 4      0.67061   0.31287   2.70784
#> 5      0.67426   0.31757   2.79590
#> 6      0.67685   0.32121   2.99403
#> 7      0.67866   0.32312   2.97202
#> 8      0.68006   0.32477   3.03806
#> 9      0.68113   0.32493   2.90597
#> 10     0.68183   0.32549   2.88396
#> 11     0.68251   0.32659   2.86194
#> 12     0.68326   0.32626   2.86194
#> 13     0.68354   0.32686   2.88396
#> 14     0.68364   0.32705   2.83993
#> 15     0.68405   0.32754   2.97202
#> 16     0.68423   0.32747   2.88396
#> 17     0.68454   0.32735   2.86194
#> 18     0.68431   0.32727   2.83993
#> 19     0.68434   0.32718   2.95000
#> 20     0.68436   0.32678   2.92799
#> validation_classification_error
#> 1      0.81460
#> 2      0.35387
#> 3      0.35285
#> 4      0.39028
#> 5      0.36770
#> 6      0.35240
#> 7      0.34848
#> 8      0.34386
#> 9      0.34807
#> 10     0.38681
#> 11     0.33774
#> 12     0.34215
#> 13     0.34431
#> 14     0.34146
#> 15     0.34329
#> 16     0.33872
#> 17     0.34276
#> 18     0.34256
#> 19     0.34289
#> 20     0.35097

```

Look at scoring history for third GBM model

```

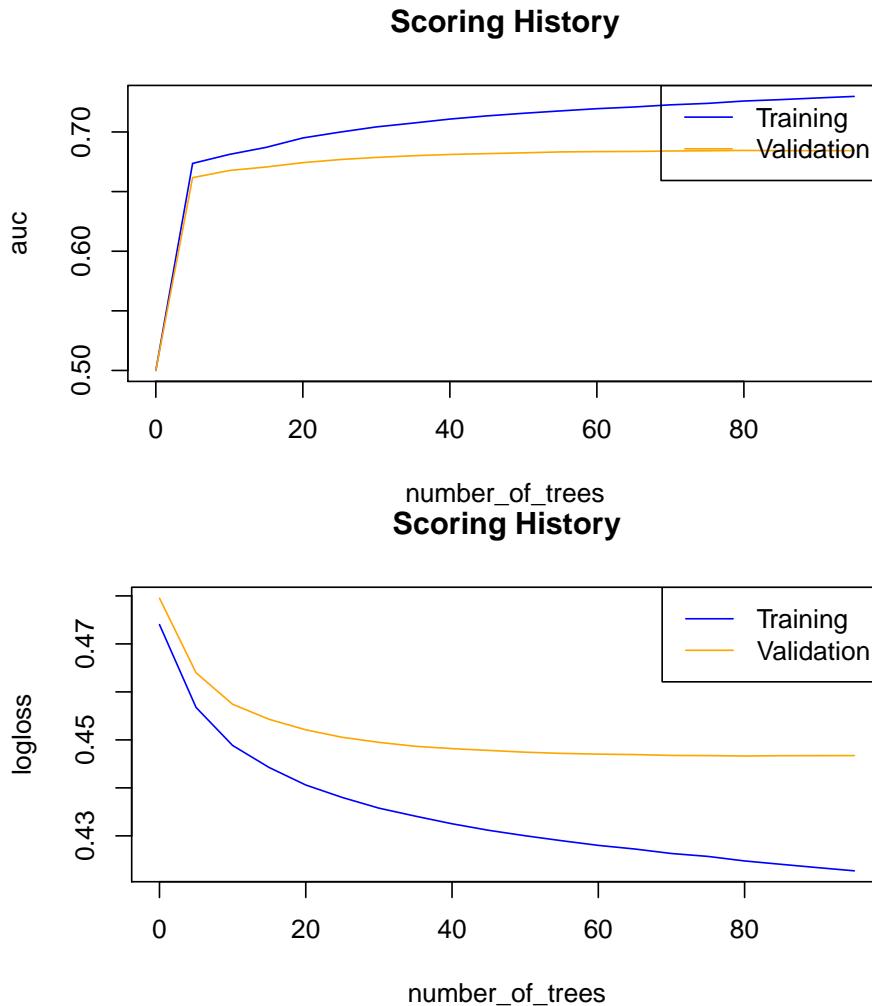
# Look at scoring history for third GBM model
plot(gbm_fit3,
      timestep = "number_of_trees",

```

```

metric = "AUC")
plot(gbm_fit3,
      timestep = "number_of_trees",
      metric = "logloss")

```



1.7 Deep Learning

H2O's Deep Learning algorithm is a multilayer feed-forward artificial neural network. It can also be used to train an autoencoder. In this example we will train a standard supervised prediction model.

1.7.1 Train a default DL

First we will train a basic DL model with default parameters. The DL model will infer the response distribution from the response encoding if it is not specified explicitly through the `distribution` argument. H2O's DL will not be reproducible if it is run on more than a single core, so in this example, the performance metrics below may vary slightly from what you see on your machine. In H2O's DL, early stopping is enabled by default, so below, it will use the training set and default stopping parameters to perform early stopping.

```
dl_fit1 <- h2o.deeplearning(x = x,
                             y = y,
                             training_frame = train,
                             model_id = "dl_fit1",
                             seed = 1)
#>
| |
|-----| 0%
|-----| 9%
|-----| 17%
|-----| 26%
|-----| 35%
|-----| 44%
|-----| 52%
|-----| 61%
|-----| 70%
|-----| 78%
|-----| 87%
|-----| 96%
|-----| 100%
```

1.7.2 Train a DL with new architecture and more epochs.

Next we will increase the number of epochs used in the GBM by setting `epochs=20` (the default is 10). Increasing the number of epochs in a deep neural net may increase performance of the model, however, you have to be careful not to overfit your model to your training data. To automatically find the optimal number of epochs, you must use H2O's early stopping functionality. Unlike the rest of the H2O algorithms, H2O's DL will use early stopping by default, so for comparison we will first turn off early stopping. We do this in the next example by setting `stopping_rounds=0`.

```
dl_fit2 <- h2o.deeplearning(x = x,
                             y = y,
                             training_frame = train,
                             model_id = "dl_fit2",
                             #validation_frame = valid, #only used if stopping_rounds > 0
                             epochs = 20,
                             hidden= c(10,10),
                             stopping_rounds = 0, # disable early stopping
                             seed = 1)
#>
|
```



1.7.3 Train a DL with early stopping

This example will use the same model parameters as `dl_fit2`. This time, we will turn on early stopping and specify the stopping criterion. We will also pass a validation set, as is recommended for early stopping.

```
dl_fit3 <- h2o.deeplearning(x = x,
                             y = y,
                             training_frame = train,
                             model_id = "dl_fit3",
                             validation_frame = valid, #in DL, early stopping is on by default
                             epochs = 20,
                             hidden = c(10,10),
                             score_interval = 1,          #used for early stopping
                             stopping_rounds = 3,         #used for early stopping
                             stopping_metric = "AUC",     #used for early stopping
                             stopping_tolerance = 0.0005, #used for early stopping
                             seed = 1)
```



Let's compare the performance of the three DL models

```
# Let's compare the performance of the three DL models
dl_perf1 <- h2o.performance(model = dl_fit1,
                             newdata = test)
dl_perf2 <- h2o.performance(model = dl_fit2,
                             newdata = test)
dl_perf3 <- h2o.performance(model = dl_fit3,
                             newdata = test)

# Print model performance
```



```

#> 4      max accuracy  0.515466 0.814694 35
#> 5      max precision 0.783239 1.000000 0
#> 6      max recall   0.012191 1.000000 396
#> 7      max specificity 0.783239 1.000000 0
#> 8      max absolute_mcc 0.196576 0.212984 227
#> 9      max min_per_class_accuracy 0.189148 0.630834 235
#> 10     max mean_per_class_accuracy 0.191007 0.633403 233
#>
#> Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T>,
dl_perf3
#> H2OBinomialMetrics: deeplearning
#>
#> MSE: 0.143
#> RMSE: 0.378
#> LogLoss: 0.452
#> Mean Per-Class Error: 0.365
#> AUC: 0.683
#> pr_auc: 0.333
#> Gini: 0.366
#>
#> Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
#>          0    1    Error      Rate
#> 0      13005 6986 0.349457 =6986/19991
#> 1      1743 2847 0.379739 =1743/4590
#> Totals 14748 9833 0.355112 =8729/24581
#>
#> Maximum Metrics: Maximum metrics at their respective thresholds
#>                  metric threshold      value idx
#> 1                  max f1 0.228083 0.394786 222
#> 2                  max f2 0.146546 0.556546 301
#> 3                  max f0point5 0.338461 0.361426 133
#> 4                  max accuracy 0.562225 0.814857 27
#> 5                  max precision 0.744195 1.000000 0
#> 6                  max recall   0.009547 1.000000 398
#> 7                  max specificity 0.744195 1.000000 0
#> 8                  max absolute_mcc 0.243434 0.216859 209
#> 9      max min_per_class_accuracy 0.223531 0.633435 226
#> 10     max mean_per_class_accuracy 0.228083 0.635402 222
#>
#> Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T>,
# Retreive test set AUC
h2o.auc(dl_perf1) # 0.6774335
#> [1] 0.677
h2o.auc(dl_perf2) # 0.678446
#> [1] 0.681
h2o.auc(dl_perf3) # 0.6770498
#> [1] 0.683

```

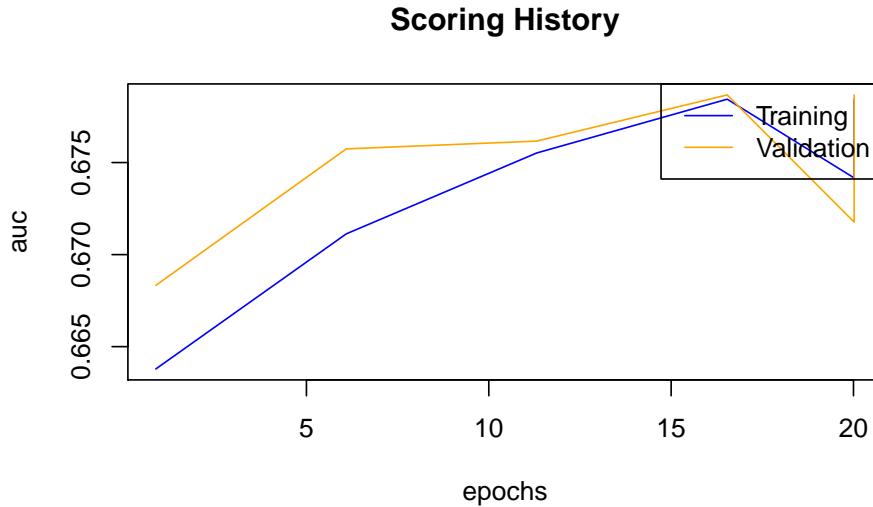
Look at scoring history for third DL model

```

# Look at scoring history for third DL model
plot(dl_fit3,
timestep = "epochs",

```

```
metric = "AUC")
```



1.7.4 Scoring history

```
# Scoring history
h2o.scoreHistory(dl_fit3)
#> Scoring History:
#>   timestamp duration training_speed epochs iterations
#> 1 2019-09-20 0.000 sec NA 0.00000 0
#> 2 2019-09-20 0.380 sec 377328 obs/sec 0.87019 1
#> 3 2019-09-20 1.481 sec 544275 obs/sec 6.09129 7
#> 4 2019-09-20 2.609 sec 556052 obs/sec 11.31383 13
#> 5 2019-09-20 3.670 sec 571338 obs/sec 16.53733 19
#> 6 2019-09-20 4.361 sec 583091 obs/sec 20.01859 23
#> 7 2019-09-20 4.429 sec 582796 obs/sec 20.01859 23
#>
#>   samples training_rmse training_logloss training_r2 training_auc
#> 1 0.000000 NA NA NA NA
#> 2 99992.000000 0.38222 0.46751 0.03339 0.66379
#> 3 699938.000000 0.37765 0.45229 0.05635 0.67113
#> 4 1300050.000000 0.37930 0.45919 0.04809 0.67552
#> 5 1900272.000000 0.37809 0.45273 0.05418 0.67844
#> 6 2300296.000000 0.37881 0.45656 0.05056 0.67416
#> 7 2300296.000000 0.37809 0.45273 0.05418 0.67844
#>
#>   training_pr_auc training_lift training_classification_error
#> 1 NA NA NA
#> 2 0.31001 2.93921 0.36812
#> 3 0.31705 2.66706 0.35750
#> 4 0.31536 2.61263 0.33445
#> 5 0.32043 2.72149 0.37894
#> 6 0.31587 2.61263 0.37561
#> 7 0.32043 2.72149 0.37894
#>
#>   validation_rmse validation_logloss validation_r2 validation_auc
#> 1 NA NA NA NA
#> 2 0.38175 0.46603 0.03505 0.66833
#> 3 0.37699 0.45116 0.05900 0.67574
#> 4 0.37885 0.45929 0.04968 0.67617
```

```

#> 5      0.37804      0.45311      0.05374      0.67868
#> 6      0.37899      0.45761      0.04899      0.67177
#> 7      0.37804      0.45311      0.05374      0.67868
#> validation_pr_auc validation_lift validation_classification_error
#> 1          NA          NA                      NA
#> 2      0.31118      2.64179
#> 3      0.31512      2.46567
#> 4      0.32010      2.61978
#> 5      0.32047      2.64179
#> 6      0.31141      2.50970
#> 7      0.32047      2.64179
# Scoring History:
# timestamp duration training_speed epochs
# 1 2016-05-03 10:33:29 0.000 sec          0.00000
# 2 2016-05-03 10:33:29 0.347 sec 424697 rows/sec 0.86851
# 3 2016-05-03 10:33:30 1.356 sec 601925 rows/sec 6.09185
# 4 2016-05-03 10:33:31 2.348 sec 717617 rows/sec 13.05168
# 5 2016-05-03 10:33:32 3.281 sec 777538 rows/sec 20.00783
# 6 2016-05-03 10:33:32 3.345 sec 777275 rows/sec 20.00783
# iterations samples training_MSE training_r2
# 1      0 0.000000
# 2      1 99804.000000 0.14402 0.03691
# 3      7 700039.000000 0.14157 0.05333
# 4     15 1499821.000000 0.14033 0.06159
# 5     23 2299180.000000 0.14079 0.05853
# 6     23 2299180.000000 0.14157 0.05333
# training_logloss training_AUC training_lift
# 1
# 2      0.45930      0.66685      2.20727
# 3      0.45220      0.68133      2.59354
# 4      0.44710      0.67993      2.70390
# 5      0.45100      0.68192      2.81426
# 6      0.45220      0.68133      2.59354
# training_classification_error validation_MSE validation_r2
# 1
# 2          0.36145      0.14682      0.03426
# 3          0.33647      0.14500      0.04619
# 4          0.37126      0.14411      0.05204
# 5          0.32868      0.14474      0.04793
# 6          0.33647      0.14500      0.04619
# validation_logloss validation_AUC validation_lift
# 1
# 2      0.46692      0.66582      2.53209
# 3      0.46256      0.67354      2.64124
# 4      0.45789      0.66986      2.44478
# 5      0.46292      0.67117      2.70672
# 6      0.46256      0.67354      2.64124
# validation_classification_error
# 1
# 2          0.37197
# 3          0.34716
# 4          0.34385
# 5          0.36544

```

```
# 6          0.34716
```

1.8 Naive Bayes model

The Naive Bayes (NB) algorithm does not usually beat an algorithm like a Random Forest or GBM, however it is still a popular algorithm, especially in the text domain (when your input is text encoded as “Bag of Words”, for example). The Naive Bayes algorithm is for binary or multiclass classification problems only, not regression. Therefore, your response must be a factor instead of a numeric.

```
# First we will train a basic NB model with default parameters.
nb_fit1 <- h2o.naiveBayes(x = x,
                           y = y,
                           training_frame = train,
                           model_id = "nb_fit1")
#>
| | | |-----| 0%
| | | |-----| 17%
| | | |-----| 100%
```

1.8.1 Train a NB model with Laplace Smoothing

One of the few tunable model parameters for the Naive Bayes algorithm is the amount of Laplace smoothing. The H2O Naive Bayes model will not use any Laplace smoothing by default.

```
nb_fit2 <- h2o.naiveBayes(x = x,
                           y = y,
                           training_frame = train,
                           model_id = "nb_fit2",
                           laplace = 6)
#>
| | | |-----| 0%
| | | |-----| 83%
| | | |-----| 100%

# Let's compare the performance of the two NB models
nb_perf1 <- h2o.performance(model = nb_fit1,
                             newdata = test)
nb_perf2 <- h2o.performance(model = nb_fit2,
                             newdata = test)

# Print model performance
nb_perf1
#> H2OBinomialMetrics: naibayes
#>
#> MSE:  0.15
#> RMSE:  0.387
```

```

#> LogLoss: 0.489
#> Mean Per-Class Error: 0.39
#> AUC: 0.651
#> pr_auc: 0.297
#> Gini: 0.303
#>
#> Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
#>          0      1      Error      Rate
#> 0    13184 6807 0.340503 =6807/19991
#> 1     2021 2569 0.440305 =2021/4590
#> Totals 15205 9376 0.359139 =8828/24581
#>
#> Maximum Metrics: Maximum metrics at their respective thresholds
#>                         metric threshold      value idx
#> 1                           max f1 0.225948 0.367893 236
#> 2                           max f2 0.090634 0.545538 346
#> 3                           max f0point5 0.340471 0.335807 166
#> 4                           max accuracy 0.999554 0.812945 0
#> 5                           max precision 0.559607 0.428747 70
#> 6                           max recall 0.000196 1.000000 399
#> 7                           max specificity 0.999554 0.999550 0
#> 8                           max absolute_mcc 0.287231 0.188641 196
#> 9   max min_per_class_accuracy 0.208724 0.602832 250
#> 10  max mean_per_class_accuracy 0.225948 0.609596 236
#>
#> Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T>,
nb_perf2
#> H2OBinomialMetrics: naivebayes
#>
#> MSE: 0.15
#> RMSE: 0.387
#> LogLoss: 0.489
#> Mean Per-Class Error: 0.39
#> AUC: 0.651
#> pr_auc: 0.297
#> Gini: 0.303
#>
#> Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
#>          0      1      Error      Rate
#> 0    14002 5989 0.299585 =5989/19991
#> 1     2207 2383 0.480828 =2207/4590
#> Totals 16209 8372 0.333428 =8196/24581
#>
#> Maximum Metrics: Maximum metrics at their respective thresholds
#>                         metric threshold      value idx
#> 1                           max f1 0.242206 0.367690 222
#> 2                           max f2 0.088660 0.545677 347
#> 3                           max f0point5 0.362995 0.336012 152
#> 4                           max accuracy 0.999564 0.812945 0
#> 5                           max precision 0.574610 0.428775 63
#> 6                           max recall 0.000207 1.000000 399
#> 7                           max specificity 0.999564 0.999550 0
#> 8                           max absolute_mcc 0.286635 0.189479 192

```

```
#> 9  max min_per_class_accuracy 0.207609 0.604357 248
#> 10 max mean_per_class_accuracy 0.247906 0.609878 218
#>
#> Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T/
```

```
# Retreive test set AUC
h2o.auc(nb_perf1) # 0.6488014
#> [1] 0.651
h2o.auc(nb_perf2) # 0.6490678
#> [1] 0.651
```


Chapter 2

Classification algorithms comparison. Diabetes dataset. (*CART, LDA, SVM, KNN, RF*)

2.1 PimaIndiansDiabetes dataset

2.2 Introduction

We compare the following classification algorithms:

- CART
- LDA
- SVM
- KNN
- RF

2.3 Workflow

1. Load dataset
2. Create the train dataset
3. Train the models
4. Collect resamples
5. Plot comparison
6. Summarize p-values

```
# load packages
library(mlbench)
library(caret)
# load the dataset
data(PimaIndiansDiabetes)

dplyr::glimpse(PimaIndiansDiabetes)
#> Observations: 768
#> Variables: 9
#> $ pregnant <dbl> 6, 1, 8, 1, 0, 5, 3, 10, 2, 8, 4, 10, 10, 1, 5, 7, 0, ...
```

```
#> $ glucose <dbl> 148, 85, 183, 89, 137, 116, 78, 115, 197, 125, 110, 1...
#> $ pressure <dbl> 72, 66, 64, 66, 40, 74, 50, 0, 70, 96, 92, 74, 80, 60...
#> $ triceps <dbl> 35, 29, 0, 23, 35, 0, 32, 0, 45, 0, 0, 0, 0, 23, 19, ...
#> $ insulin <dbl> 0, 0, 0, 94, 168, 0, 88, 0, 543, 0, 0, 0, 0, 846, 175...
#> $ mass <dbl> 33.6, 26.6, 23.3, 28.1, 43.1, 25.6, 31.0, 35.3, 30.5, ...
#> $ pedigree <dbl> 0.627, 0.351, 0.672, 0.167, 2.288, 0.201, 0.248, 0.13...
#> $ age <dbl> 50, 31, 32, 21, 33, 30, 26, 29, 53, 54, 30, 34, 57, 5...
#> $ diabetes <fct> pos, neg, pos, neg, pos, neg, pos, neg, pos, pos, neg...
```

```
tibble::as_tibble(PimaIndiansDiabetes)
#> # A tibble: 768 x 9
#>   pregnant glucose pressure triceps insulin mass pedigree age diabetes
#>       <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <dbl>    <fct>
#> 1       6      148      72      35      0    33.6    0.627    50    pos
#> 2       1      85       66      29      0    26.6    0.351    31    neg
#> 3       8      183      64       0      0    23.3    0.672    32    pos
#> 4       1      89       66      23     94    28.1    0.167    21    neg
#> 5       0      137      40      35    168    43.1    2.29     33    pos
#> 6       5      116      74       0      0    25.6    0.201    30    neg
#> # ... with 762 more rows
```

2.4 Train the models using cross-validation

```
# prepare training scheme
trainControl <- trainControl(method = "repeatedcv",
                               number=10,
                               repeats=3)

# CART
set.seed(7)
fit.cart <- train(diabetes~., data=PimaIndiansDiabetes,
                   method = "rpart", trControl=trainControl)

# LDA: Linear Discriminant Analysis
set.seed(7)
fit.lda <- train(diabetes~., data=PimaIndiansDiabetes,
                  method="lda", trControl=trainControl)

# SVM
set.seed(7)
fit.svm <- train(diabetes~., data=PimaIndiansDiabetes,
                   method="svmRadial", trControl=trainControl)

# KNN
set.seed(7)
fit.knn <- train(diabetes~., data=PimaIndiansDiabetes,
                  method="knn", trControl=trainControl)

# Random Forest
set.seed(7)
fit.rf <- train(diabetes~., data=PimaIndiansDiabetes,
                 method="rf", trControl=trainControl)
```

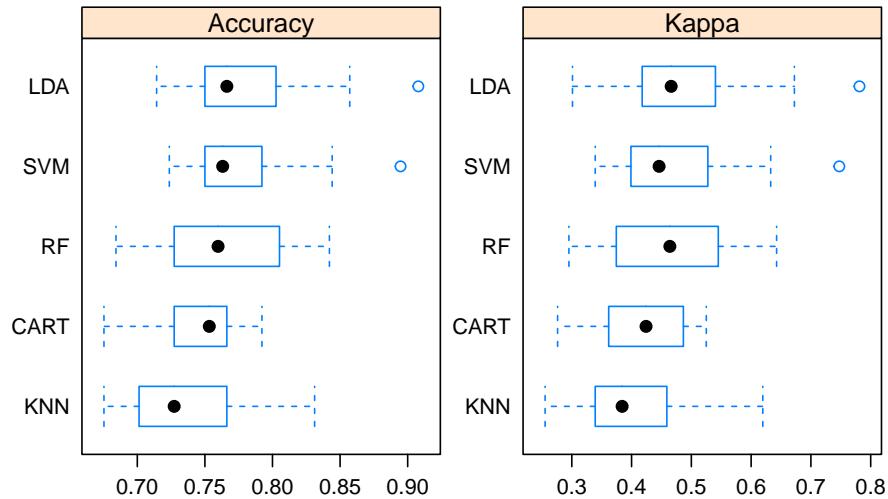
```
# collect resamples
results <- resamples(list(CART=fit.cart,
                           LDA=fit lda,
                           SVM=fit.svm,
                           KNN=fit.knn,
                           RF=fit.rf))
```

2.5 Compare models

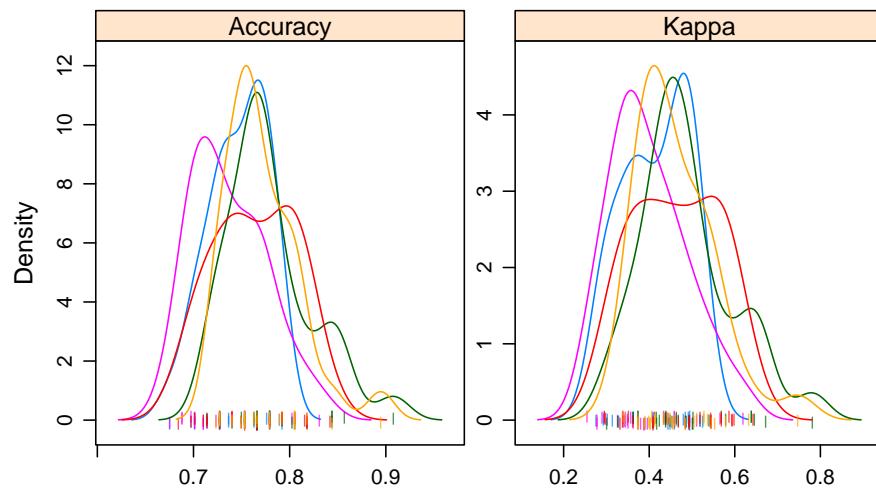
```
# summarize differences between models
summary(results)
#>
#> Call:
#> summary.resamples(object = results)
#>
#> Models: CART, LDA, SVM, KNN, RF
#> Number of resamples: 30
#>
#> Accuracy
#>      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> CART 0.675 0.727 0.753 0.747 0.766 0.792 0
#> LDA 0.714 0.751 0.766 0.779 0.800 0.908 0
#> SVM 0.724 0.751 0.763 0.771 0.792 0.895 0
#> KNN 0.675 0.704 0.727 0.737 0.766 0.831 0
#> RF 0.684 0.731 0.760 0.764 0.802 0.842 0
#>
#> Kappa
#>      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> CART 0.276 0.362 0.424 0.415 0.486 0.525 0
#> LDA 0.301 0.419 0.466 0.486 0.531 0.781 0
#> SVM 0.339 0.400 0.446 0.462 0.523 0.748 0
#> KNN 0.255 0.341 0.384 0.398 0.454 0.620 0
#> RF 0.295 0.378 0.464 0.463 0.545 0.643 0
```

2.6 Plot comparison

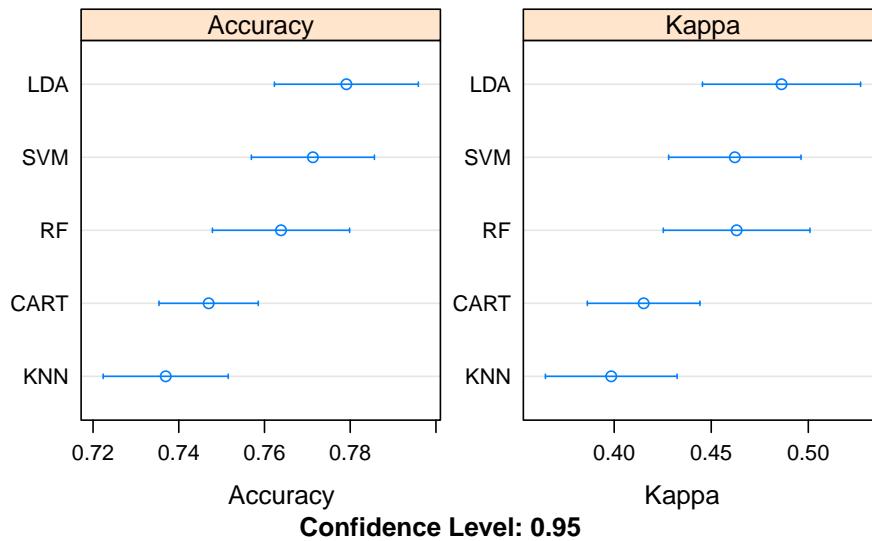
```
# box and whisker plots to compare models
scales <- list(x=list(relation="free"), y=list(relation="free"))
bwplot(results, scales=scales)
```



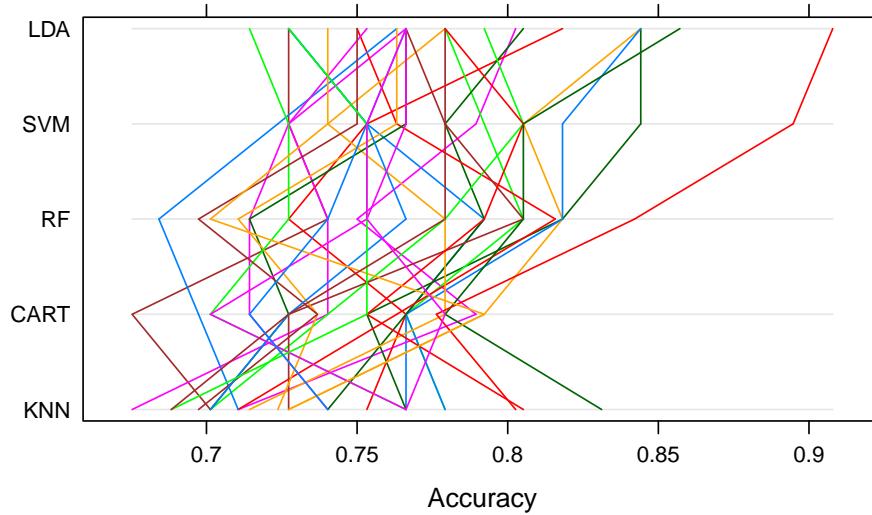
```
# density plots of accuracy
scales <- list(x=list(relation="free"), y=list(relation="free"))
densityplot(results, scales=scales, pch = "|")
```



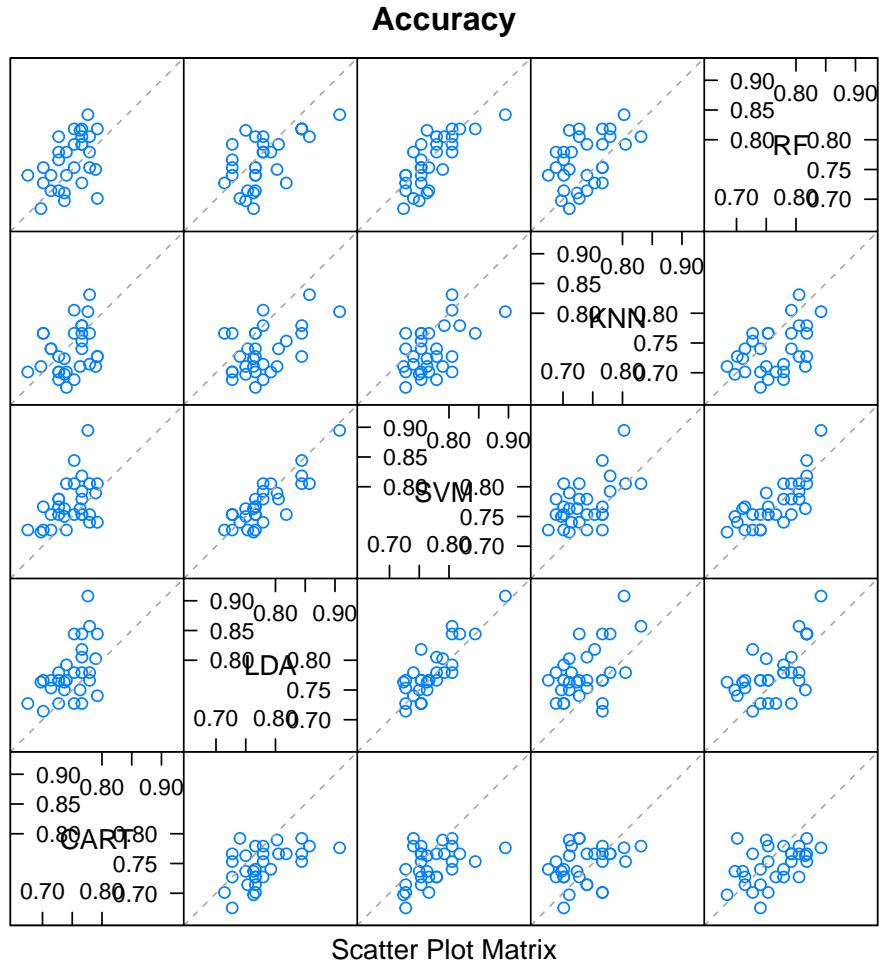
```
# dot plots of accuracy
scales <- list(x=list(relation="free"), y=list(relation="free"))
dotplot(results, scales=scales)
```



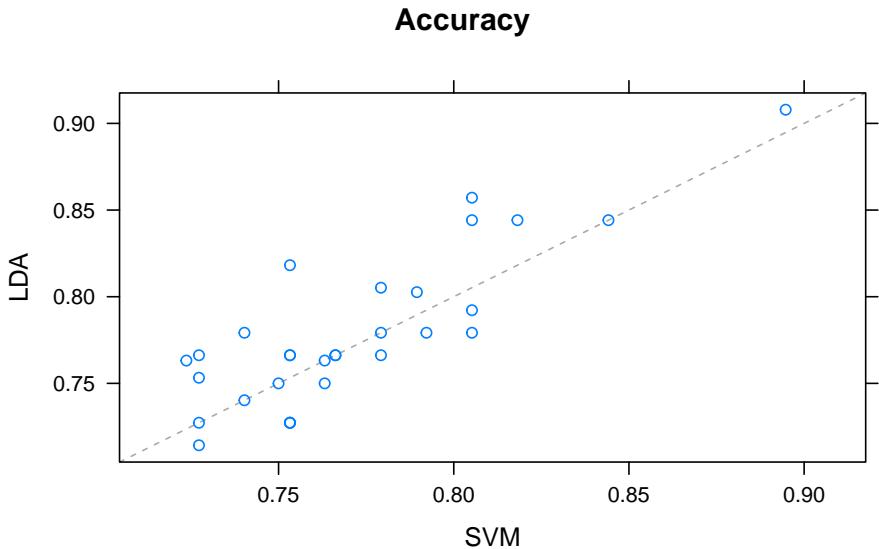
```
# parallel plots to compare models
parallelplot(results)
```



```
# pairwise scatter plots of predictions to compare models
splom(results)
```



```
# xyplot plots to compare models
xyplot(results, models=c("LDA", "SVM"))
```



```
# difference in model predictions
diffs <- diff(results)
# summarize p-values for pairwise comparisons
```

```
summary(diffs)
#>
#> Call:
#> summary.diff.resamples(object = diffs)
#>
#> p-value adjustment: bonferroni
#> Upper diagonal: estimates of the difference
#> Lower diagonal: p-value for H0: difference = 0
#>
#> Accuracy
#>      CART      LDA      SVM      KNN      RF
#> CART -0.03214 -0.02432  0.01002 -0.01688
#> LDA  0.001186          0.00781  0.04216  0.01525
#> SVM  0.011640  0.915689          0.03434  0.00744
#> KNN  1.000000  6.68e-05  0.000294          -0.02690
#> RF   0.272754  0.449062  1.000000  0.018379
#>
#> Kappa
#>      CART      LDA      SVM      KNN      RF
#> CART -0.071016 -0.046972  0.016687 -0.047894
#> LDA  0.000809          0.024044  0.087703  0.023122
#> SVM  0.025808  0.356273          0.063659 -0.000922
#> KNN  1.000000  0.000386  0.004082          -0.064581
#> RF   0.021176  1.000000  1.000000  0.015897
```


Chapter 3

Multiclass classification comparison. Diabetes dataset. (*LDA, CART, KNN, SVM, RF*)

3.1 `iris` dataset

3.2 Introduction

These are the algorithms used:

1. LDA
2. CART
3. KNN
4. SVM
5. RF

```
# load the caret package
library(caret)
#> Loading required package: lattice
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures    rlang
#>   c.quosures    rlang
#>   print.quosures rlang
# attach the iris dataset to the environment
data(iris)
# rename the dataset
dataset <- iris
```

3.3 Workflow

1. Load dataset
2. Create train and test datasets, 80/20

3. Inspect dataset
4. Visualize features
5. Set the train control to
 - 10 cross-validations
 - Metric: accuracy
6. Train the models
7. Compare accuracy of models
8. Visual comparison
9. Make predictions on validation set

We will split the loaded dataset into two, 80% of which we will use to train our models and 20% that we will hold back as a validation dataset.

```
# create a list of 80% of the rows in the original dataset we can use for training
validationIndex <- createDataPartition(dataset$Species, p=0.80, list=FALSE)
# select 20% of the data for validation
validation <- dataset[-validationIndex,]

# use the remaining 80% of data to training and testing the models
dataset <- dataset[validationIndex,]

# dimensions of dataset
dim(dataset)
#> [1] 120   5

# list types for each attribute
sapply(dataset, class)
#> Sepal.Length Sepal.Width Petal.Length Petal.Width      Species
#> "numeric"     "numeric"    "numeric"      "numeric"      "factor"
```

3.4 Peek at the dataset

```
# take a peek at the first 5 rows of the data
head(dataset)
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1       5.1      3.5       1.4      0.2  setosa
#> 2       4.9      3.0       1.4      0.2  setosa
#> 3       4.7      3.2       1.3      0.2  setosa
#> 4       4.6      3.1       1.5      0.2  setosa
#> 5       5.0      3.6       1.4      0.2  setosa
#> 6       5.4      3.9       1.7      0.4  setosa

library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
```

```

glimpse(dataset)
#> Observations: 120
#> Variables: 5
#> $ Sepal.Length <dbl> 5.1, 4.9, 4.7, 4.6, 5.0, 5.4, 4.6, 5.0, 4.4, 4.9, ...
#> $ Sepal.Width <dbl> 3.5, 3.0, 3.2, 3.1, 3.6, 3.9, 3.4, 3.4, 2.9, 3.1, ...
#> $ Petal.Length <dbl> 1.4, 1.4, 1.3, 1.5, 1.4, 1.7, 1.4, 1.5, 1.4, 1.5, ...
#> $ Petal.Width <dbl> 0.2, 0.2, 0.2, 0.2, 0.2, 0.4, 0.3, 0.2, 0.2, 0.1, ...
#> $ Species      <fct> setosa, setosa, setosa, setosa, setosa, setosa, s...
library(skimr)
#>
#> Attaching package: 'skimr'
#> The following object is masked from 'package:stats':
#>
#>     filter

skim(dataset)
#> Skim summary statistics
#> n obs: 120
#> n variables: 5
#>
#> -- Variable type:factor -----
#> variable missing complete   n n_unique          top_counts
#> Species      0       120 120           3 set: 40, ver: 40, vir: 40, NA: 0
#> ordered
#> FALSE
#>
#> -- Variable type:numeric -----
#>     variable missing complete   n mean    sd   p0   p25   p50   p75   p100
#> Petal.Length  0       120 3.76 1.78 1  1.58 4.35 5.1   6.9
#> Petal.Width   0       120 1.2  0.76 0.1 0.3  1.3  1.8   2.5
#> Sepal.Length  0       120 5.86 0.84 4.3 5.1  5.8  6.4   7.9
#> Sepal.Width   0       120 3.06 0.44 2  2.8   3  3.32 4.4
#>     hist
#>
#>
#>
#>
```

3.5 Levels of the class

```

# list the levels for the class
levels(dataset$Species)
#> [1] "setosa"      "versicolor"  "virginica"
```

3.6 class distribution

```

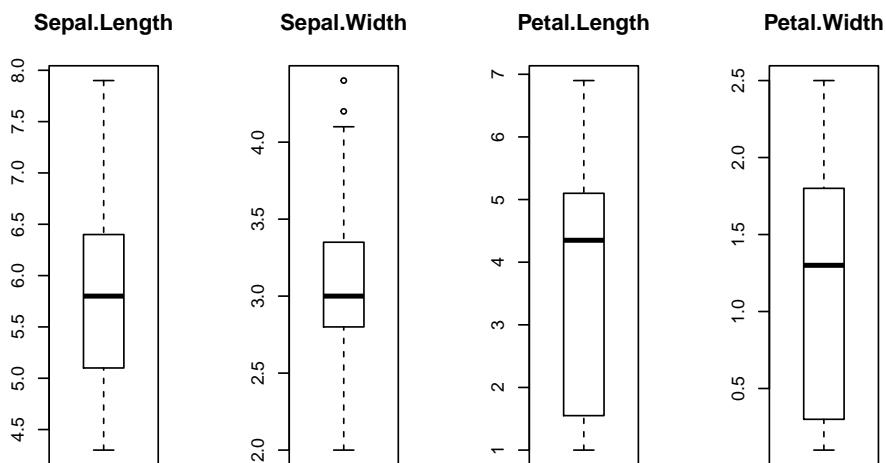
# summarize the class distribution
percentage <- prop.table(table(dataset$Species)) * 100
```

```
cbind(freq=table(dataset$Species), percentage=percentage)
#>           freq   percentage
#> setosa      40     33.3
#> versicolor  40     33.3
#> virginica   40     33.3
```

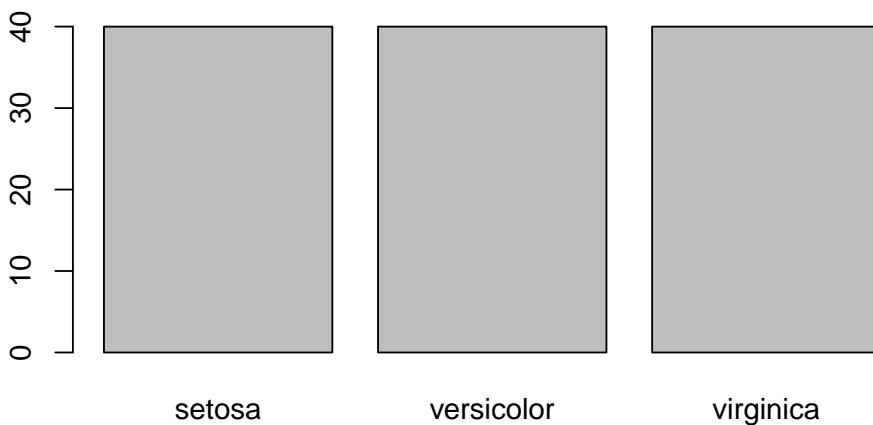
3.7 Visualize the dataset

```
# split input and output
x <- dataset[,1:4]
y <- dataset[,5]

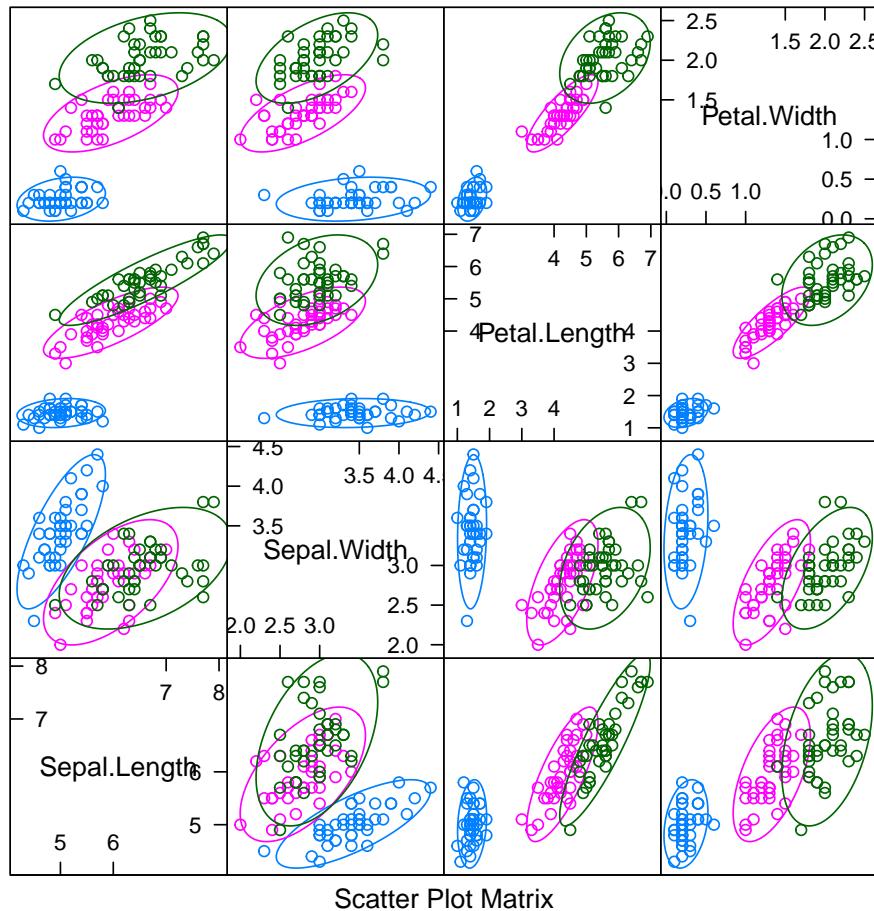
# boxplot for each attribute on one image
par(mfrow=c(1,4))
for(i in 1:4) {
  boxplot(x[,i], main=names(dataset)[i])
}
```



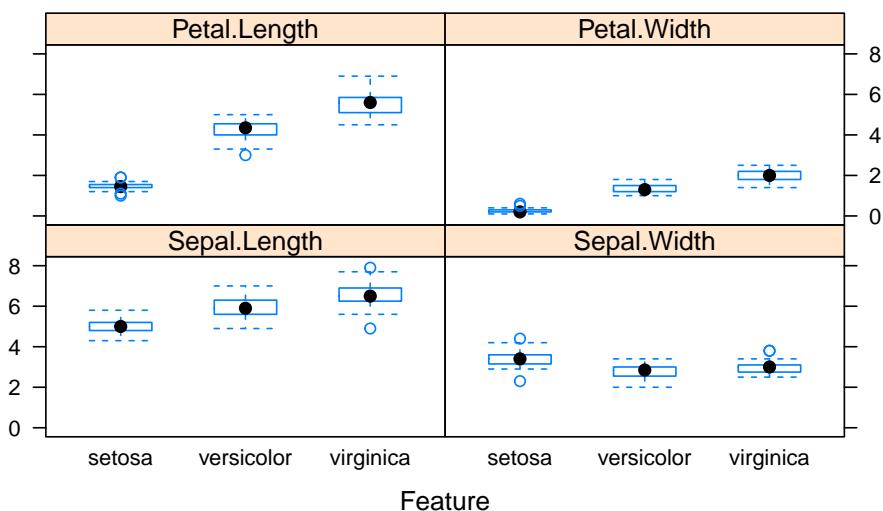
```
# barplot for class breakdown
plot(y)
```



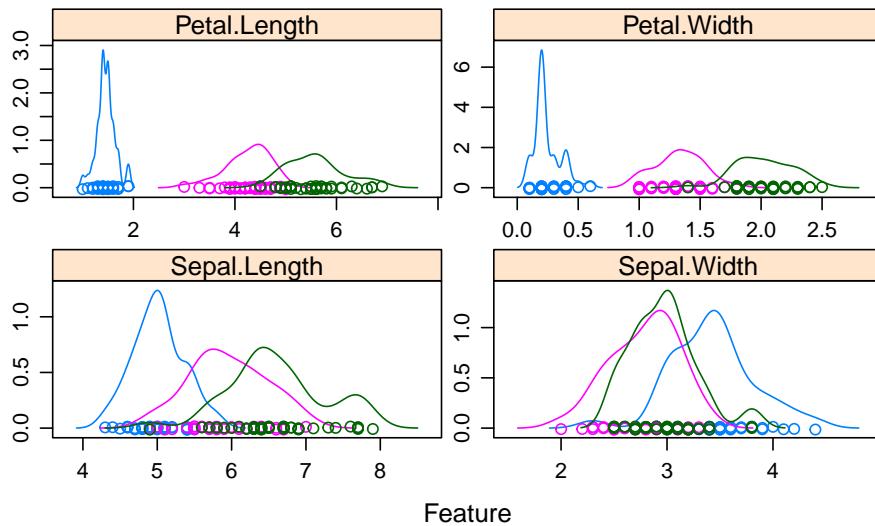
```
# scatter plot matrix
featurePlot(x=x, y=y, plot="ellipse")
```



```
# box and whisker plots for each attribute
featurePlot(x=x, y=y, plot="box")
```



```
# density plots for each attribute by class value
scales <- list(x=list(relation="free"), y=list(relation="free"))
featurePlot(x=x, y=y, plot="density", scales=scales)
```



3.8 Evaluate algorithms

3.8.1 split and metrics

```
# Run algorithms using 10-fold cross-validation
trainControl <- trainControl(method="cv", number=10)
metric <- "Accuracy"
```

3.8.2 build models

```
# LDA
set.seed(7)
fit.lda <- train(Species~, data=dataset, method = "lda",
                  metric=metric, trControl=trainControl)

# CART
set.seed(7)
fit.cart <- train(Species~, data=dataset, method = "rpart",
                   metric=metric, trControl=trainControl)

# KNN
set.seed(7)
fit.knn <- train(Species~, data=dataset, method = "knn",
                   metric=metric, trControl=trainControl)

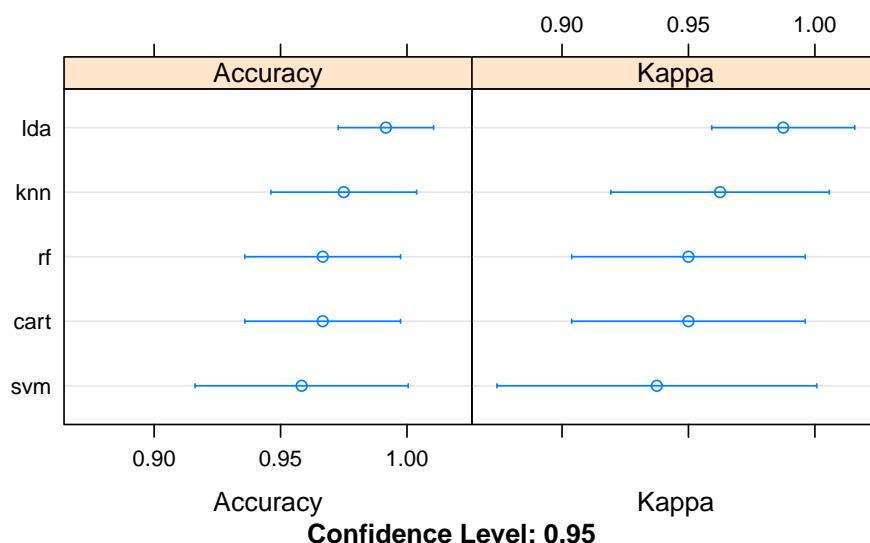
# SVM
set.seed(7)
fit.svm <- train(Species~, data=dataset, method = "svmRadial",
                   metric=metric, trControl=trainControl)

# Random Forest
set.seed(7)
fit.rf <- train(Species~, data=dataset, method = "rf",
                 metric=metric, trControl=trainControl)
```

3.8.3 compare

```
#summarize accuracy of models
results <- resamples(list(lda = fit.lda,
                           cart = fit.cart,
                           knn = fit.knn,
                           svm = fit.svm,
                           rf = fit.rf))
summary(results)
#>
#> Call:
#> summary.resamples(object = results)
#>
#> Models: lda, cart, knn, svm, rf
#> Number of resamples: 10
#>
#> Accuracy
#>      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> lda  0.917  1.000   1 0.992    1    1    0
#> cart 0.917  0.917   1 0.967    1    1    0
#> knn  0.917  0.938   1 0.975    1    1    0
#> svm  0.833  0.917   1 0.958    1    1    0
#> rf   0.917  0.917   1 0.967    1    1    0
#>
#> Kappa
#>      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> lda  0.875  1.000   1 0.987    1    1    0
#> cart 0.875  0.875   1 0.950    1    1    0
#> knn  0.875  0.906   1 0.962    1    1    0
#> svm  0.750  0.875   1 0.937    1    1    0
#> rf   0.875  0.875   1 0.950    1    1    0

# compare accuracy of models
dotplot(results)
```



```
# summarize Best Model
print(fit.lda)
#> Linear Discriminant Analysis
#>
#> 120 samples
#> 4 predictor
#> 3 classes: 'setosa', 'versicolor', 'virginica'
#>
#> No pre-processing
#> Resampling: Cross-Validated (10 fold)
#> Summary of sample sizes: 108, 108, 108, 108, 108, ...
#> Resampling results:
#>
#> Accuracy Kappa
#> 0.992    0.987
```

3.9 Make predictions

```
# estimate skill of LDA on the validation dataset
predictions <- predict(fit.lda, validation)
confusionMatrix(predictions, validation$Species)
#> Confusion Matrix and Statistics
#>
#>           Reference
#> Prediction   setosa versicolor virginica
#>   setosa       10      0      0
#>   versicolor    0      9      1
#>   virginica     0      1      9
#>
#> Overall Statistics
#>
#>           Accuracy : 0.933
#>             95% CI : (0.779, 0.992)
#>   No Information Rate : 0.333
#>   P-Value [Acc > NIR] : 8.75e-12
#>
#>           Kappa : 0.9
#>
#> Mcnemar's Test P-Value : NA
#>
#> Statistics by Class:
#>
#>           Class: setosa Class: versicolor Class: virginica
#> Sensitivity            1.000      0.900      0.900
#> Specificity             1.000      0.950      0.950
#> Pos Pred Value          1.000      0.900      0.900
#> Neg Pred Value          1.000      0.950      0.950
#> Prevalence              0.333      0.333      0.333
#> Detection Rate           0.333      0.300      0.300
#> Detection Prevalence     0.333      0.333      0.333
#> Balanced Accuracy        1.000      0.925      0.925
```

Chapter 4

Regression algorithms comparison. Boston dataset. (*LM, GKM,* *GLMNET, SVM, CART, KNN*)

4.1 Boston dataset

- Comparison of various algorithms.

4.2 Introduction

These are the algorithms used:

1. LM
2. GLM
3. GLMNET
4. SVM
5. CART
6. KNN

```
# load packages
library(mlbench)
library(caret)
#> Loading required package: lattice
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method           from
#>   [.quosures      rlang
#>   c.quosures      rlang
#>   print.quosures rlang
library(corrplot)
#> corrplot 0.84 loaded

# attach the BostonHousing dataset
data(BostonHousing)
```

```
dplyr::glimpse(BostonHousing)
#> Observations: 506
#> Variables: 14
#> $ crim    <dbl> 0.00632, 0.02731, 0.02729, 0.03237, 0.06905, 0.02985, ...
#> $ zn      <dbl> 18.0, 0.0, 0.0, 0.0, 0.0, 0.0, 12.5, 12.5, 12.5, ...
#> $ indus   <dbl> 2.31, 7.07, 7.07, 2.18, 2.18, 2.18, 7.87, 7.87, 7.87, ...
#> $ chas    <fct> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
#> $ nox     <dbl> 0.538, 0.469, 0.469, 0.458, 0.458, 0.458, 0.524, 0.524...
#> $ rm      <dbl> 6.58, 6.42, 7.18, 7.00, 7.15, 6.43, 6.01, 6.17, 5.63, ...
#> $ age     <dbl> 65.2, 78.9, 61.1, 45.8, 54.2, 58.7, 66.6, 96.1, 100.0, ...
#> $ dis     <dbl> 4.09, 4.97, 4.97, 6.06, 6.06, 6.06, 5.56, 5.95, 6.08, ...
#> $ rad     <dbl> 1, 2, 2, 3, 3, 5, 5, 5, 5, 5, 4, 4, 4, 4, 4, ...
#> $ tax     <dbl> 296, 242, 242, 222, 222, 311, 311, 311, 311, 311, ...
#> $ ptratio <dbl> 15.3, 17.8, 17.8, 18.7, 18.7, 18.7, 15.2, 15.2, 15.2, ...
#> $ b       <dbl> 397, 397, 393, 395, 397, 394, 396, 397, 387, 387, 393, ...
#> $ lstat   <dbl> 4.98, 9.14, 4.03, 2.94, 5.33, 5.21, 12.43, 19.15, 29.9...
#> $ medu   <dbl> 24.0, 21.6, 34.7, 33.4, 36.2, 28.7, 22.9, 27.1, 16.5, ...

tibble::as_tibble(BostonHousing)
#> # A tibble: 506 x 14
#>   crim      zn indus chas      nox      rm      age      dis      rad      tax      ptratio
#>   <dbl> <dbl> <dbl> <fct> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 0.00632    18  2.31  0     0.538   6.58   65.2   4.09    1    296   15.3
#> 2 0.0273     0   7.07  0     0.469   6.42   78.9   4.97    2    242   17.8
#> 3 0.0273     0   7.07  0     0.469   7.18   61.1   4.97    2    242   17.8
#> 4 0.0324     0   2.18  0     0.458   7.00   45.8   6.06    3    222   18.7
#> 5 0.0690     0   2.18  0     0.458   7.15   54.2   6.06    3    222   18.7
#> 6 0.0298     0   2.18  0     0.458   6.43   58.7   6.06    3    222   18.7
#> # ... with 500 more rows, and 3 more variables: b <dbl>, lstat <dbl>,
#> #   medu <dbl>
```

4.3 Workflow

1. Load dataset
2. Create train and test datasets, 80/20
3. Inspect dataset:
 - Dimension
 - classes
 - `skimr`
4. Analyze features
 - correlation
5. Visualize features
 - histograms
 - density plots
 - pairwise
 - correlogram
5. Train as-is

- Set the train control to
 - 10 cross-validations
 - 3 repetitions
 - Metric: RMSE
 - Train the models
 - Compare accuracy of models
 - Visual comparison
 - dot plot
6. Train with Feature selection
- Feature selection
 - `findCorrelation`
 - generate new dataset
 - Train models again
- Compare RMSE again
 - Visual comparison
 - dot plot
7. Train with dataset transformation
- data transformation
 - Center
 - Scale
 - BoxCox
 - Train models
 - Compare RMSE
 - Visual comparison
 - dot plot
8. Tune the best model
- Set the train control to
 - 10 cross-validations
 - 3 repetitions
 - Metric: RMSE
 - Train the models
 - Radial SVM
 - Sigma vector
 - .C
 - BoxCox 9, Ensembling
 - Select the algorithms
 - Random Forest
 - Stochastic Gradient Boosting
 - Cubist
 - Numeric comparison
 - resample
 - summary
 - Visual comparison
 - dot plot
10. Tune the best model: Cubist
- Set the train control to
 - 10 cross-validations
 - 3 repetitions

- Metric: RMSE
- Train the models
 - Cubist
 - .committees
 - .neighbors
 - BoxCox
- Evaluate the tuning parameters
 - Numeric comparison
 - * print tuned model
 - Visual comparison
 - * scatter plot

11. Finalize the model

- Back transformation
- Summary

12. Apply model to validation set

- Transform the dataset
- Make prediction
- Calculate the RMSE

```
# Split out validation dataset
# create a list of 80% of the rows in the original dataset we can use for training
set.seed(7)
validationIndex <- createDataPartition(BostonHousing$medv,
                                         p=0.80, list=FALSE)

# select 20% of the data for validation
validation <- BostonHousing[-validationIndex,]

# use the remaining 80% of data to training and testing the models
dataset <- BostonHousing[validationIndex,]

# dimensions of dataset
dim(validation)
#> [1] 99 14
dim(dataset)
#> [1] 407 14

# list types for each attribute
sapply(dataset, class)
#>      crim       zn     indus      chas      nox       rm       age
#> "numeric" "numeric" "numeric" "factor" "numeric" "numeric" "numeric"
#>      dis       rad      tax   ptratio       b     lstat     medv
#> "numeric" "numeric" "numeric" "numeric" "numeric" "numeric" "numeric"

# take a peek at the first 20 rows of the data
head(dataset, n=20)
#>      crim     zn  indus  chas    nox     rm     age   dis   rad tax ptratio     b lstat
#> 1  0.00632 18.0 2.31 0 0.538 6.58 65.2 4.09 1 296 15.3 397 4.98
#> 2  0.02731  0.0 7.07 0 0.469 6.42 78.9 4.97 2 242 17.8 397 9.14
#> 3  0.02729  0.0 7.07 0 0.469 7.18 61.1 4.97 2 242 17.8 393 4.03
#> 4  0.03237  0.0 2.18 0 0.458 7.00 45.8 6.06 3 222 18.7 395 2.94
#> 5  0.06905  0.0 2.18 0 0.458 7.15 54.2 6.06 3 222 18.7 397 5.33
#> 6  0.02985  0.0 2.18 0 0.458 6.43 58.7 6.06 3 222 18.7 394 5.21
#> 7  0.08829 12.5 7.87 0 0.524 6.01 66.6 5.56 5 311 15.2 396 12.43
```

```

#> 10 0.17004 12.5 7.87 0 0.524 6.00 85.9 6.59 5 311 15.2 387 17.10
#> 11 0.22489 12.5 7.87 0 0.524 6.38 94.3 6.35 5 311 15.2 393 20.45
#> 12 0.11747 12.5 7.87 0 0.524 6.01 82.9 6.23 5 311 15.2 397 13.27
#> 13 0.09378 12.5 7.87 0 0.524 5.89 39.0 5.45 5 311 15.2 390 15.71
#> 14 0.62976 0.0 8.14 0 0.538 5.95 61.8 4.71 4 307 21.0 397 8.26
#> 15 0.63796 0.0 8.14 0 0.538 6.10 84.5 4.46 4 307 21.0 380 10.26
#> 17 1.05393 0.0 8.14 0 0.538 5.93 29.3 4.50 4 307 21.0 387 6.58
#> 20 0.72580 0.0 8.14 0 0.538 5.73 69.5 3.80 4 307 21.0 391 11.28
#> 21 1.25179 0.0 8.14 0 0.538 5.57 98.1 3.80 4 307 21.0 377 21.02
#> 22 0.85204 0.0 8.14 0 0.538 5.96 89.2 4.01 4 307 21.0 393 13.83
#> 23 1.23247 0.0 8.14 0 0.538 6.14 91.7 3.98 4 307 21.0 397 18.72
#> 24 0.98843 0.0 8.14 0 0.538 5.81 100.0 4.10 4 307 21.0 395 19.88
#> 25 0.75026 0.0 8.14 0 0.538 5.92 94.1 4.40 4 307 21.0 394 16.30
#> medv
#> 1 24.0
#> 2 21.6
#> 3 34.7
#> 4 33.4
#> 5 36.2
#> 6 28.7
#> 7 22.9
#> 10 18.9
#> 11 15.0
#> 12 18.9
#> 13 21.7
#> 14 20.4
#> 15 18.2
#> 17 23.1
#> 20 18.2
#> 21 13.6
#> 22 19.6
#> 23 15.2
#> 24 14.5
#> 25 15.6

```

```

library(skimr)
#>
#> Attaching package: 'skimr'
#> The following object is masked from 'package:stats':
#>
#>   filter
skim_with(numeric = list(hist = NULL))
skim(dataset)
#> Skim summary statistics
#> n obs: 407
#> n variables: 14
#>
#> -- Variable type:factor --
#> variable missing complete n n_unique top_counts ordered
#>   chas     0    407 407      2 0: 378, 1: 29, NA: 0 FALSE
#>
#> -- Variable type:numeric --
#> variable missing complete n mean sd p0 p25 p50 p75
#>   age     0    407 407 68.38 28.16 6.2 42.7 77.3 94.2

```

```

#>      b      0      407 407 357.19  89.67   0.32   373.81 391.27 396.02
#>      crim    0      407 407   3.64   8.8   0.0063   0.08   0.27   3.69
#>      dis      0      407 407   3.82   2.12   1.13   2.11   3.15   5.21
#>      indus    0      407 407    11   6.87   0.74   4.93   8.56  18.1
#>      lstat     0      407 407 12.56   7.03   1.92   7.06  11.32 16.45
#>      medu     0      407 407 22.52   8.96   5     17.05 21.2   25
#>      nox      0      407 407   0.55   0.12   0.39   0.45   0.54   0.63
#>      ptratio   0      407 407 18.42   2.18  12.6   17     19   20.2
#>      rad       0      407 407   9.59   8.77   1     4     5   24
#>      rm        0      407 407   6.29   0.7   3.56   5.89  6.21  6.62
#>      tax       0      407 407 408.75 168.72 187   280.5 334   666
#>      zn        0      407 407 11.92  24.19   0     0     0   15
#>
#>      p100
#> 100
#> 396.9
#> 88.98
#> 12.13
#> 27.74
#> 37.97
#> 50
#> 0.87
#> 22
#> 24
#> 8.78
#> 711
#> 100

dataset[,4] <- as.numeric(as.character(dataset[,4]))

skim(dataset)
#> Skim summary statistics
#> n obs: 407
#> n variables: 14
#>
#> -- Variable type:numeric --
#> variable missing complete n mean sd p0 p25 p50
#> age      0      407 407 68.38 28.16 6.2 42.7 77.3
#> b        0      407 407 357.19 89.67 0.32 373.81 391.27
#> chas     0      407 407 0.071 0.26 0     0     0
#> crim     0      407 407   3.64   8.8   0.0063 0.08   0.27
#> dis      0      407 407   3.82   2.12   1.13   2.11   3.15
#> indus    0      407 407    11   6.87   0.74   4.93   8.56
#> lstat    0      407 407 12.56   7.03   1.92   7.06  11.32
#> medu     0      407 407 22.52   8.96   5     17.05 21.2
#> nox      0      407 407   0.55   0.12   0.39   0.45   0.54
#> ptratio   0      407 407 18.42   2.18  12.6   17     19
#> rad       0      407 407   9.59   8.77   1     4     5
#> rm        0      407 407   6.29   0.7   3.56   5.89  6.21
#> tax       0      407 407 408.75 168.72 187   280.5 334
#> zn        0      407 407 11.92  24.19   0     0     0
#>
#> p75      p100
#> 94.2 100
#> 396.02 396.9
#> 0      1

```

```
#>   3.69  88.98
#>   5.21  12.13
#>  18.1   27.74
#> 16.45  37.97
#>  25     50
#>   0.63   0.87
#> 20.2    22
#>  24     24
#>   6.62   8.78
#> 666    711
#>  15     100
```

no more factors or character variables

```
# find correlation between variables
cor(dataset[,1:13])
#>      crim      zn    indus    chas    nox      rm      age      dis
#> crim  1.0000 -0.1996  0.4076 -0.05507  0.4099 -0.194  0.3524 -0.376
#> zn    -0.1996  1.0000 -0.5314 -0.02987 -0.5202  0.311 -0.5845  0.680
#> indus  0.4076 -0.5314  1.0000  0.06583  0.7733 -0.383  0.6512 -0.711
#> chas   -0.0551 -0.0299  0.0658  1.00000  0.0934  0.127  0.0735 -0.099
#> nox    0.4099 -0.5202  0.7733  0.09340  1.0000 -0.296  0.7338 -0.769
#> rm     -0.1940  0.3111 -0.3826  0.12677 -0.2961  1.000 -0.2262  0.207
#> age    0.3524 -0.5845  0.6512  0.07350  0.7338 -0.226  1.0000 -0.749
#> dis    -0.3756  0.6799 -0.7113 -0.09905 -0.7693  0.207 -0.7492  1.000
#> rad    0.6083 -0.3227  0.6200 -0.00245  0.6276 -0.221  0.4690 -0.504
#> tax    0.5711 -0.3184  0.7185 -0.03064  0.6758 -0.295  0.5058 -0.526
#> ptratio 0.2897 -0.3888  0.3782 -0.12283  0.1888 -0.365  0.2709 -0.228
#> b      -0.3442  0.1747 -0.3644  0.03782 -0.3684  0.126 -0.2742  0.284
#> lstat   0.4229 -0.4219  0.6136 -0.08430  0.5839 -0.612  0.6066 -0.501
#>      rad      tax ptratio      b      lstat
#> crim   0.60834  0.5711  0.290 -0.3442  0.4229
#> zn     -0.32273 -0.3184 -0.389  0.1747 -0.4219
#> indus   0.61998  0.7185  0.378 -0.3644  0.6136
#> chas   -0.00245 -0.0306 -0.123  0.0378 -0.0843
#> nox    0.62760  0.6758  0.189 -0.3684  0.5839
#> rm     -0.22126 -0.2953 -0.365  0.1260 -0.6120
#> age    0.46896  0.5058  0.271 -0.2742  0.6066
#> dis    -0.50372 -0.5264 -0.228  0.2843 -0.5013
#> rad    1.00000  0.9201  0.480 -0.4231  0.5025
#> tax    0.92005  1.0000  0.469 -0.4303  0.5538
#> ptratio 0.47971  0.4691  1.000 -0.1700  0.4093
#> b      -0.42314 -0.4303 -0.170  1.0000 -0.3509
#> lstat   0.50251  0.5538  0.409 -0.3509  1.0000
```

```
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
```

```

m <- cor(dataset[,1:13])
diag(m) <- 0

# select variables with correlation 0.7 and above
threshold <- 0.7
ok <- apply(abs(m) >= threshold, 1, any)
m[, ok, ok]

#>      indus    nox    age    dis    rad    tax
#> indus 0.000  0.773  0.651 -0.711  0.620  0.719
#> nox   0.773  0.000  0.734 -0.769  0.628  0.676
#> age    0.651  0.734  0.000 -0.749  0.469  0.506
#> dis   -0.711 -0.769 -0.749  0.000 -0.504 -0.526
#> rad    0.620  0.628  0.469 -0.504  0.000  0.920
#> tax    0.719  0.676  0.506 -0.526  0.920  0.000

# values of correlation >= 0.7
ind <- sapply(1:13, function(x) abs(m[, x]) > 0.7)
m[ind]
#> [1] 0.773 -0.711  0.719  0.773  0.734 -0.769  0.734 -0.749 -0.711 -0.769
#> [11] -0.749  0.920  0.719  0.920

# defining a index for selecting if the condition is met
cind <- apply(m, 2, function(x) any(abs(x) > 0.7))
cm <- m[, cind] # since col6 only has values less than 0.5 it is not taken
cm

#>      indus    nox    age    dis    rad    tax
#> crim   0.4076  0.4099  0.3524 -0.376  0.60834  0.5711
#> zn     -0.5314 -0.5202 -0.5845  0.680 -0.32273 -0.3184
#> indus   0.0000  0.7733  0.6512 -0.711  0.61998  0.7185
#> chas    0.0658  0.0934  0.0735 -0.099 -0.00245 -0.0306
#> nox    0.7733  0.0000  0.7338 -0.769  0.62760  0.6758
#> rm     -0.3826 -0.2961 -0.2262  0.207 -0.22126 -0.2953
#> age    0.6512  0.7338  0.0000 -0.749  0.46896  0.5058
#> dis   -0.7113 -0.7693 -0.7492  0.000 -0.50372 -0.5264
#> rad    0.6200  0.6276  0.4690 -0.504  0.00000  0.9201
#> tax    0.7185  0.6758  0.5058 -0.526  0.92005  0.0000
#> ptratio 0.3782  0.1888  0.2709 -0.228  0.47971  0.4691
#> b      -0.3644 -0.3684 -0.2742  0.284 -0.42314 -0.4303
#> lstat   0.6136  0.5839  0.6066 -0.501  0.50251  0.5538

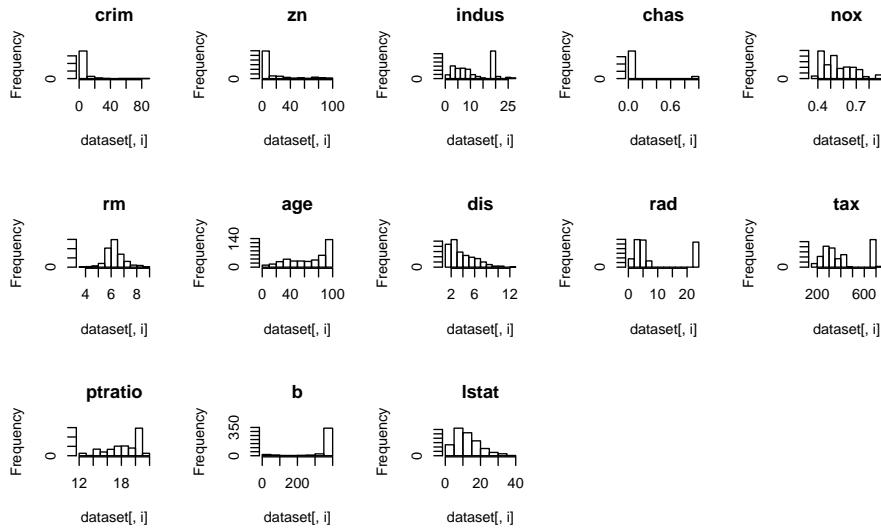
rind <- apply(cm, 1, function(x) any(abs(x) > 0.7))
rm <- cm[rind, ]
rm

#>      indus    nox    age    dis    rad    tax
#> indus 0.000  0.773  0.651 -0.711  0.620  0.719
#> nox   0.773  0.000  0.734 -0.769  0.628  0.676
#> age    0.651  0.734  0.000 -0.749  0.469  0.506
#> dis   -0.711 -0.769 -0.749  0.000 -0.504 -0.526
#> rad    0.620  0.628  0.469 -0.504  0.000  0.920
#> tax    0.719  0.676  0.506 -0.526  0.920  0.000

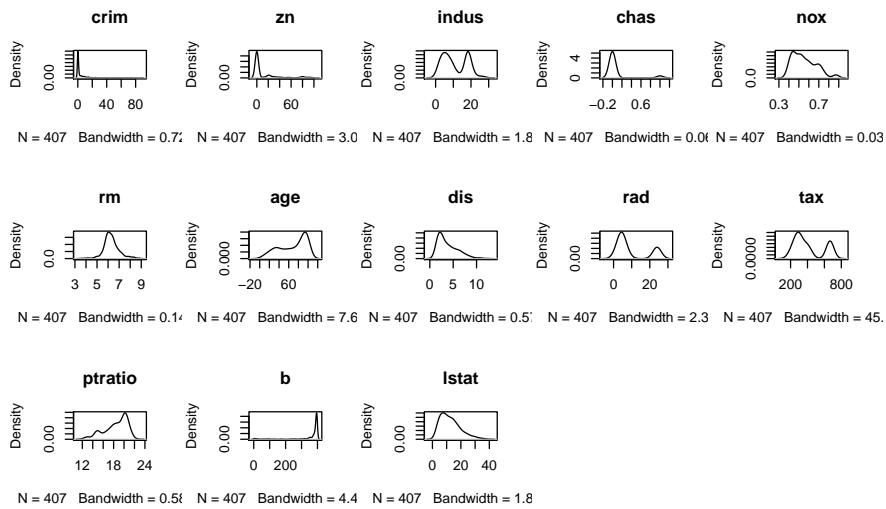
# histograms for each attribute
par(mfrow=c(3,5))
for(i in 1:13) {
  hist(dataset[,i], main=names(dataset)[i])
}

```

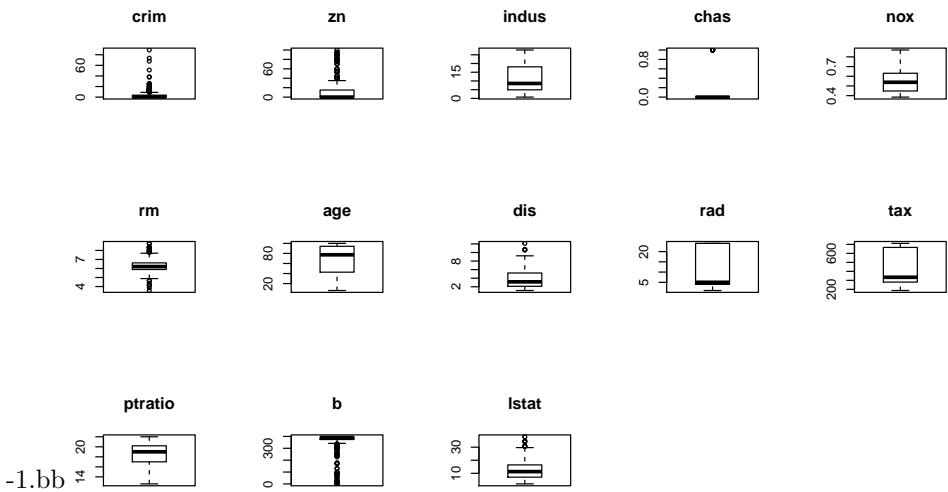
}



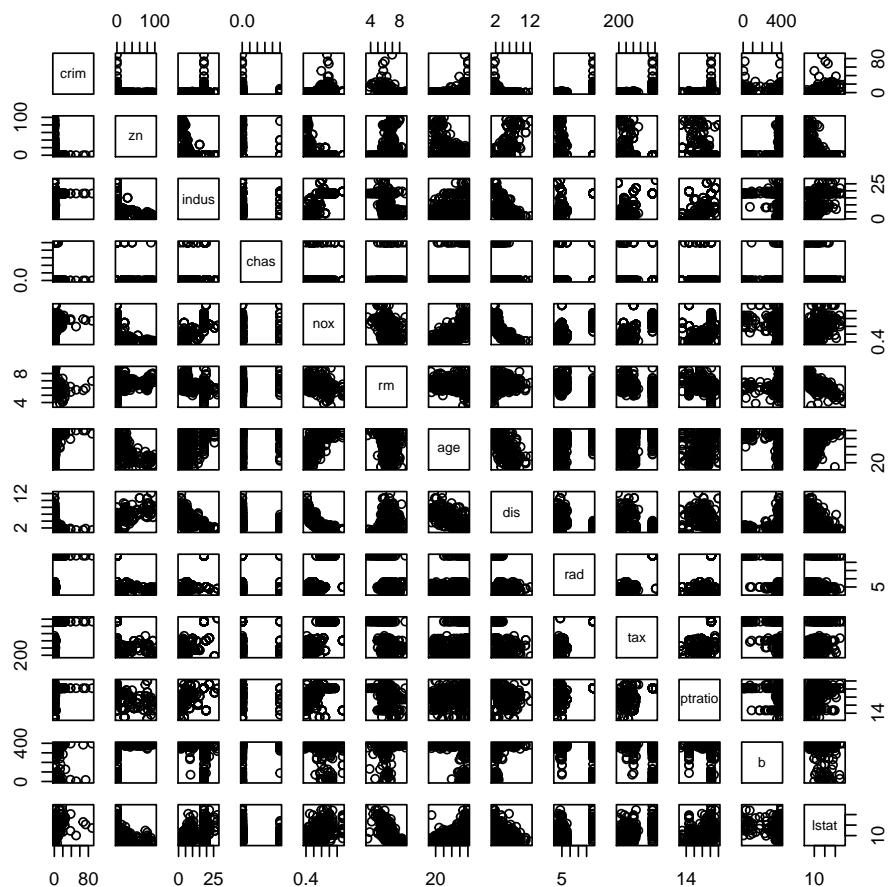
```
# density plot for each attribute
par(mfrow=c(3,5))
for(i in 1:13) {
  plot(density(dataset[,i]), main=names(dataset)[i])
}
```



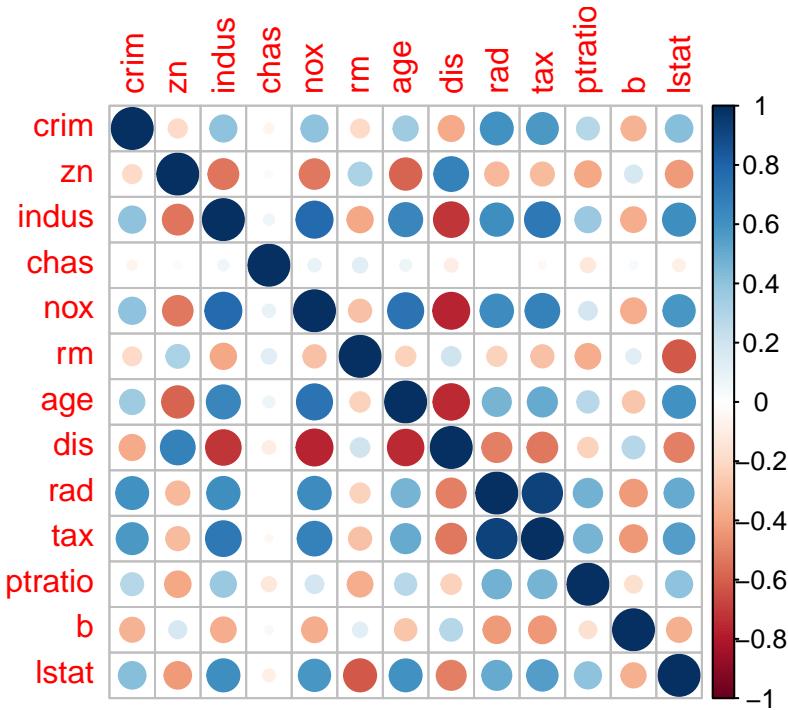
```
# boxplots for each attribute
par(mfrow=c(3,5))
for(i in 1:13) {
  boxplot(dataset[,i], main=names(dataset)[i])
}
```



```
# scatter plot matrix
pairs(dataset[,1:13])
```



```
# correlation plot
correlations <- cor(dataset[,1:13])
corrplot(correlations, method="circle")
```



4.4 Evaluation

```

# Run algorithms using 10-fold cross-validation
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
metric <- "RMSE"

# LM
set.seed(7)
fit.lm <- train(medv~., data=dataset, method="lm",
                  metric=metric, preProc=c("center", "scale"),
                  trControl=trainControl)

# GLM
set.seed(7)
fit.glm <- train(medv~., data=dataset, method="glm",
                  metric=metric, preProc=c("center", "scale"),
                  trControl=trainControl)

# GLMNET
set.seed(7)
fit.glmnet <- train(medv~., data=dataset, method="glmnet",
                     metric=metric,
                     preProc=c("center", "scale"),
                     trControl=trainControl)

# SVM
set.seed(7)
fit.svm <- train(medv~., data=dataset, method="svmRadial",
                  metric=metric,
                  preProc=c("center", "scale"),
                  trControl=trainControl)

# CART

```

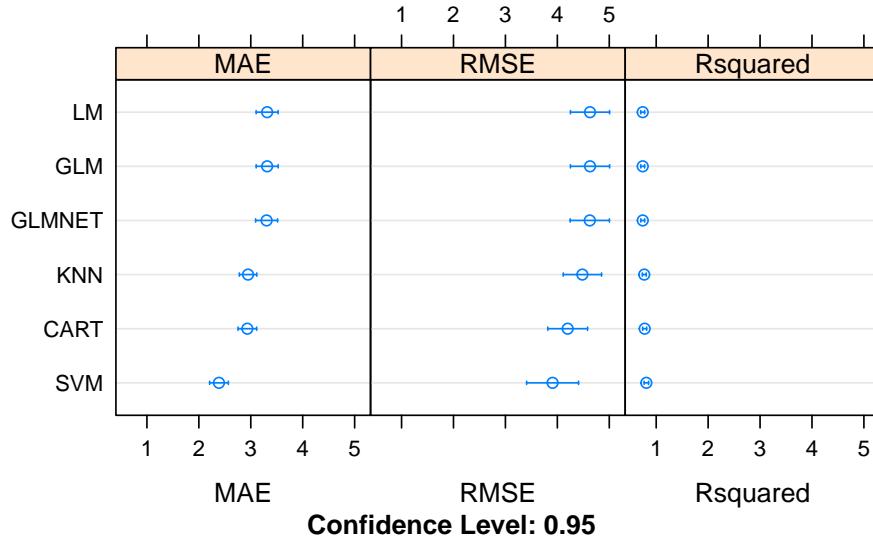
```

set.seed(7)
grid <- expand.grid(.cp=c(0, 0.05, 0.1))
fit.cart <- train(medv~., data=dataset, method="rpart",
                  metric=metric, tuneGrid=grid,
                  preProc=c("center", "scale"),
                  trControl=trainControl)

# KNN
set.seed(7)
fit.knn <- train(medv~., data=dataset, method="knn",
                  metric=metric, preProc=c("center", "scale"),
                  trControl=trainControl)

# Compare algorithms
results <- resamples(list(LM      = fit.lm,
                           GLM     = fit.glm,
                           GLMNET = fit.glmnet,
                           SVM    = fit.svm,
                           CART   = fit.cart,
                           KNN    = fit.knn))
summary(results)
#>
#> Call:
#> summary.resamples(object = results)
#>
#> Models: LM, GLM, GLMNET, SVM, CART, KNN
#> Number of resamples: 30
#>
#> MAE
#>      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> LM    2.30  2.90  3.37 3.32   3.70 4.64   0
#> GLM   2.30  2.90  3.37 3.32   3.70 4.64   0
#> GLMNET 2.30  2.88  3.34 3.30   3.70 4.63   0
#> SVM   1.42  1.99  2.52 2.39   2.65 3.35   0
#> CART   2.22  2.62  2.88 2.93   3.08 4.16   0
#> KNN   1.98  2.69  2.87 2.95   3.24 4.00   0
#>
#> RMSE
#>      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> LM    2.99  3.87  4.63 4.63   5.32 6.69   0
#> GLM   2.99  3.87  4.63 4.63   5.32 6.69   0
#> GLMNET 2.99  3.88  4.62 4.62   5.32 6.69   0
#> SVM   2.05  2.95  3.81 3.91   4.46 6.98   0
#> CART   2.77  3.38  4.00 4.20   4.60 7.09   0
#> KNN   2.65  3.74  4.42 4.48   5.06 6.98   0
#>
#> Rsquared
#>      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> LM    0.505  0.674  0.747 0.740   0.813 0.900   0
#> GLM   0.505  0.674  0.747 0.740   0.813 0.900   0
#> GLMNET 0.503  0.673  0.747 0.741   0.816 0.904   0
#> SVM   0.519  0.762  0.845 0.810   0.896 0.970   0
#> CART   0.514  0.737  0.816 0.778   0.842 0.899   0
#> KNN   0.519  0.748  0.804 0.770   0.829 0.931   0
dotplot(results)

```



4.5 Feature selection

```

# remove correlated attributes
# find attributes that are highly correlated
set.seed(7)
cutoff <- 0.70
correlations <- cor(dataset[,1:13])
highlyCorrelated <- findCorrelation(correlations, cutoff=cutoff)

for (value in highlyCorrelated) {
  print(names(dataset)[value])
}
#> [1] "indus"
#> [1] "nox"
#> [1] "tax"
#> [1] "dis"

# create a new dataset without highly correlated features
datasetFeatures <- dataset[,-highlyCorrelated]
dim(datasetFeatures)
#> [1] 407 10

# Run algorithms using 10-fold cross-validation
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
metric <- "RMSE"

# LM
set.seed(7)
fit.lm <- train(medv~., data=dataset, method="lm",
                 metric=metric, preProc=c("center", "scale"),
                 trControl=trainControl)

# GLM
set.seed(7)
fit.glm <- train(medv~., data=dataset, method="glm",

```

```

        metric=metric, preProc=c("center", "scale"),
        trControl=trainControl)

# GLMNET
set.seed(7)
fit.glmnet <- train(medv~., data=dataset, method="glmnet",
                     metric=metric,
                     preProc=c("center", "scale"),
                     trControl=trainControl)

# SVM
set.seed(7)
fit.svm <- train(medv~., data=dataset, method="svmRadial",
                   metric=metric,
                   preProc=c("center", "scale"),
                   trControl=trainControl)

# CART
set.seed(7)
grid <- expand.grid(.cp=c(0, 0.05, 0.1))
fit.cart <- train(medv~., data=dataset, method="rpart",
                   metric=metric, tuneGrid=grid,
                   preProc=c("center", "scale"),
                   trControl=trainControl)

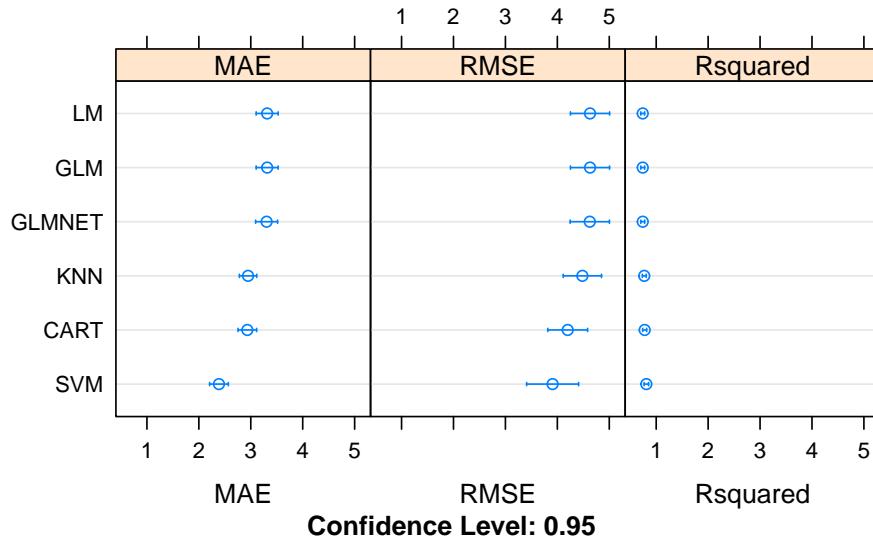
# KNN
set.seed(7)
fit.knn <- train(medv~., data=dataset, method="knn",
                  metric=metric, preProc=c("center", "scale"),
                  trControl=trainControl)

# Compare algorithms
feature_results <- resamples(list(LM      = fit.lm,
                                     GLM     = fit.glm,
                                     GLMNET = fit.glmnet,
                                     SVM    = fit.svm,
                                     CART   = fit.cart,
                                     KNN    = fit.knn))

summary(feature_results)
#>
#> Call:
#> summary.resamples(object = feature_results)
#>
#> Models: LM, GLM, GLMNET, SVM, CART, KNN
#> Number of resamples: 30
#>
#> MAE
#>      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> LM    2.30  2.90  3.37 3.32   3.70 4.64    0
#> GLM   2.30  2.90  3.37 3.32   3.70 4.64    0
#> GLMNET 2.30  2.88  3.34 3.30   3.70 4.63    0
#> SVM   1.42  1.99  2.52 2.39   2.65 3.35    0
#> CART   2.22  2.62  2.88 2.93   3.08 4.16    0
#> KNN   1.98  2.69  2.87 2.95   3.24 4.00    0
#>
#> RMSE
#>      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's

```

```
#> LM      2.99   3.87   4.63 4.63    5.32 6.69    0
#> GLM     2.99   3.87   4.63 4.63    5.32 6.69    0
#> GLMNET  2.99   3.88   4.62 4.62    5.32 6.69    0
#> SVM     2.05   2.95   3.81 3.91    4.46 6.98    0
#> CART    2.77   3.38   4.00 4.20    4.60 7.09    0
#> KNN     2.65   3.74   4.42 4.48    5.06 6.98    0
#>
#> Rsquared
#>           Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> LM      0.505 0.674 0.747 0.740 0.813 0.900 0
#> GLM     0.505 0.674 0.747 0.740 0.813 0.900 0
#> GLMNET  0.503 0.673 0.747 0.741 0.816 0.904 0
#> SVM     0.519 0.762 0.845 0.810 0.896 0.970 0
#> CART    0.514 0.737 0.816 0.778 0.842 0.899 0
#> KNN     0.519 0.748 0.804 0.770 0.829 0.931 0
dotplot(feature_results)
```



Comparing the results, we can see that this has made the RMSE worse for the linear and the nonlinear algorithms. The correlated attributes we removed are contributing to the accuracy of the models.

4.6 Evaluate Algorithms: Box-Cox Transform

We know that some of the attributes have a skew and others perhaps have an exponential distribution. One option would be to explore squaring and log transforms respectively (you could try this!). Another approach would be to use a power transform and let it figure out the amount to correct each attribute. One example is the Box-Cox power transform. Let's try using this transform to rescale the original data and evaluate the effect on the same 6 algorithms. We will also leave in the centering and scaling for the benefit of the instance-based methods.

```
# Run algorithms using 10-fold cross-validation
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
metric <- "RMSE"

# lm
set.seed(7)
```

```

fit.lm <- train(medv~, data=dataset, method="lm", metric=metric,
                 preProc=c("center", "scale", "BoxCox"),
                 trControl=trainControl)

# GLM
set.seed(7)
fit.glm <- train(medv~, data=dataset, method="glm", metric=metric,
                  preProc=c("center", "scale", "BoxCox"),
                  trControl=trainControl)

# GLMNET
set.seed(7)
fit.glmnet <- train(medv~, data=dataset, method="glmnet", metric=metric,
                     preProc=c("center", "scale", "BoxCox"),
                     trControl=trainControl)

# SVM
set.seed(7)
fit.svm <- train(medv~, data=dataset, method="svmRadial", metric=metric,
                  preProc=c("center", "scale", "BoxCox"),
                  trControl=trainControl)

# CART
set.seed(7)
grid <- expand.grid(.cp=c(0, 0.05, 0.1))
fit.cart <- train(medv~, data=dataset, method="rpart", metric=metric,
                   tuneGrid=grid,
                   preProc=c("center", "scale", "BoxCox"),
                   trControl=trainControl)

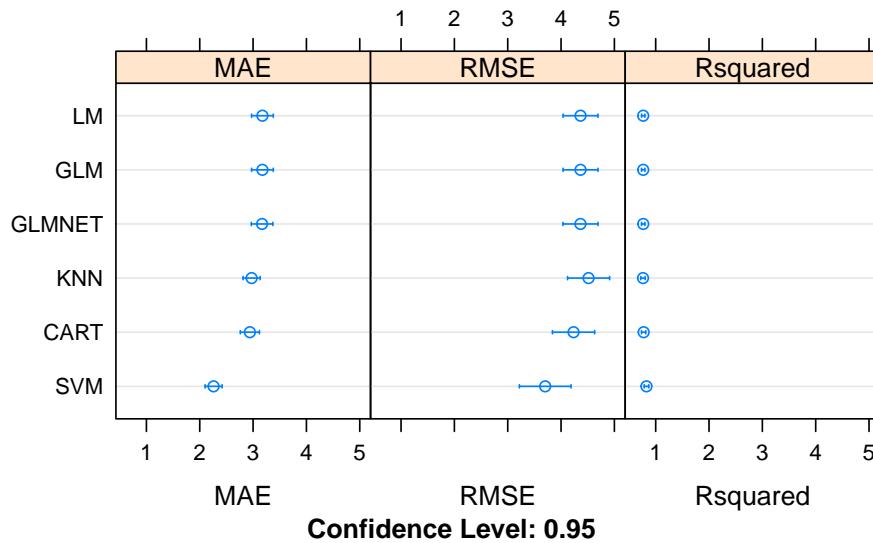
# KNN
set.seed(7)
fit.knn <- train(medv~, data=dataset, method="knn", metric=metric,
                  preProc=c("center", "scale", "BoxCox"),
                  trControl=trainControl)

# Compare algorithms
transformResults <- resamples(list(LM      = fit.lm,
                                      GLM     = fit.glm,
                                      GLMNET = fit.glmnet,
                                      SVM    = fit.svm,
                                      CART   = fit.cart,
                                      KNN    = fit.knn))

summary(transformResults)
#>
#> Call:
#> summary.resamples(object = transformResults)
#>
#> Models: LM, GLM, GLMNET, SVM, CART, KNN
#> Number of resamples: 30
#>
#> MAE
#>      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> LM     2.10   2.80   3.21 3.18   3.45 4.44   0
#> GLM    2.10   2.80   3.21 3.18   3.45 4.44   0
#> GLMNET 2.11   2.80   3.21 3.17   3.45 4.43   0
#> SVM    1.30   1.95   2.25 2.26   2.48 3.19   0
#> CART   2.22   2.62   2.89 2.94   3.11 4.16   0

```

```
#> KNN      2.33    2.66    2.82 2.97    3.27 3.96    0
#>
#> RMSE
#>          Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> LM       2.82    3.81    4.43 4.37    5.00 6.16    0
#> GLM      2.82    3.81    4.43 4.37    5.00 6.16    0
#> GLMNET   2.83    3.79    4.42 4.36    5.00 6.18    0
#> SVM      1.80    2.73    3.41 3.70    4.24 6.73    0
#> CART     2.77    3.38    4.00 4.23    4.83 7.09    0
#> KNN      3.01    3.73    4.37 4.52    5.02 7.30    0
#>
#> Rsquared
#>          Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> LM       0.560   0.712   0.775 0.766   0.825 0.910   0
#> GLM      0.560   0.712   0.775 0.766   0.825 0.910   0
#> GLMNET   0.556   0.713   0.775 0.766   0.826 0.910   0
#> SVM      0.524   0.778   0.854 0.827   0.907 0.979   0
#> CART     0.514   0.727   0.816 0.774   0.843 0.899   0
#> KNN      0.492   0.723   0.792 0.762   0.842 0.937   0
dotplot(transformResults)
```



4.7 Tune SVM

```
print(fit.svm)
#> Support Vector Machines with Radial Basis Function Kernel
#>
#> 407 samples
#> 13 predictor
#>
#> Pre-processing: centered (13), scaled (13), Box-Cox transformation (11)
#> Resampling: Cross-Validated (10 fold, repeated 3 times)
#> Summary of sample sizes: 365, 366, 366, 367, 366, 366, ...
#> Resampling results across tuning parameters:
#>
```

```

#>      C      RMSE   Rsquared   MAE
#> 0.25  4.54  0.772     2.73
#> 0.50  4.07  0.802     2.46
#> 1.00  3.70  0.827     2.26
#>
#> Tuning parameter 'sigma' was held constant at a value of 0.116
#> RMSE was used to select the optimal model using the smallest value.
#> The final values used for the model were sigma = 0.116 and C = 1.

```

Let's design a grid search around a C value of 1. We might see a small trend of decreasing RMSE with increasing C, so let's try all integer C values between 1 and 10. Another parameter that caret let us tune is the sigma parameter. This is a smoothing parameter. Good sigma values often start around 0.1, so we will try numbers before and after.

```

# tune SVM sigma and C parametres
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
metric <- "RMSE"
set.seed(7)

grid <- expand.grid(.sigma = c(0.025, 0.05, 0.1, 0.15),
                     .C = seq(1, 10, by=1))

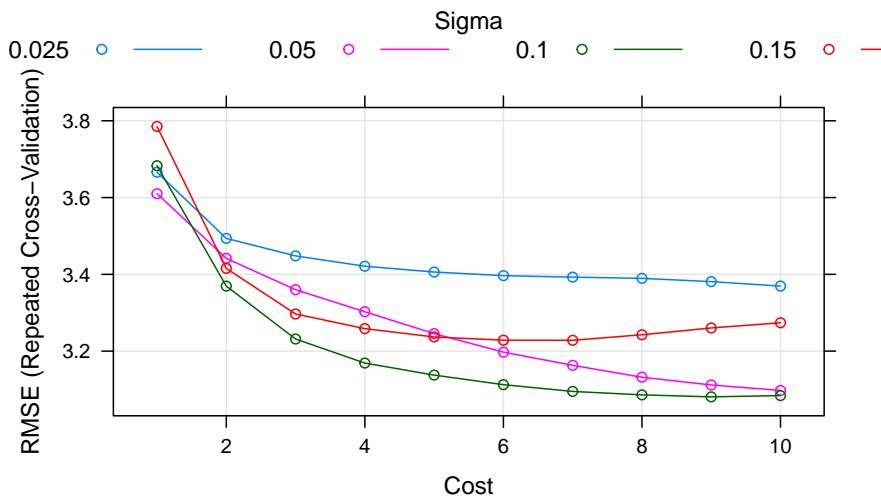
fit.svm <- train(medv~., data=dataset, method="svmRadial", metric=metric,
                  tuneGrid=grid,
                  preProc=c("BoxCox"), trControl=trainControl)
print(fit.svm)
#> Support Vector Machines with Radial Basis Function Kernel
#>
#> 407 samples
#> 13 predictor
#>
#> Pre-processing: Box-Cox transformation (11)
#> Resampling: Cross-Validated (10 fold, repeated 3 times)
#> Summary of sample sizes: 365, 366, 366, 367, 366, 366, ...
#> Resampling results across tuning parameters:
#>
#>      sigma  C      RMSE   Rsquared   MAE
#> 0.025    1  3.67  0.830     2.34
#> 0.025    2  3.49  0.840     2.21
#> 0.025    3  3.45  0.842     2.17
#> 0.025    4  3.42  0.844     2.14
#> 0.025    5  3.41  0.845     2.13
#> 0.025    6  3.40  0.846     2.12
#> 0.025    7  3.39  0.846     2.11
#> 0.025    8  3.39  0.846     2.11
#> 0.025    9  3.38  0.846     2.11
#> 0.025   10  3.37  0.847     2.10
#> 0.050    1  3.61  0.833     2.25
#> 0.050    2  3.44  0.843     2.17
#> 0.050    3  3.36  0.848     2.11
#> 0.050    4  3.30  0.852     2.08
#> 0.050    5  3.25  0.856     2.05
#> 0.050    6  3.20  0.860     2.03
#> 0.050    7  3.16  0.862     2.02

```

```

#> 0.050 8 3.13 0.865 2.02
#> 0.050 9 3.11 0.866 2.01
#> 0.050 10 3.10 0.867 2.01
#> 0.100 1 3.68 0.829 2.26
#> 0.100 2 3.37 0.848 2.12
#> 0.100 3 3.23 0.858 2.06
#> 0.100 4 3.17 0.862 2.04
#> 0.100 5 3.14 0.865 2.04
#> 0.100 6 3.11 0.866 2.04
#> 0.100 7 3.09 0.868 2.04
#> 0.100 8 3.09 0.868 2.04
#> 0.100 9 3.08 0.868 2.04
#> 0.100 10 3.08 0.868 2.05
#> 0.150 1 3.79 0.822 2.30
#> 0.150 2 3.42 0.846 2.14
#> 0.150 3 3.30 0.854 2.09
#> 0.150 4 3.26 0.857 2.09
#> 0.150 5 3.24 0.858 2.09
#> 0.150 6 3.23 0.858 2.10
#> 0.150 7 3.23 0.857 2.12
#> 0.150 8 3.24 0.856 2.13
#> 0.150 9 3.26 0.855 2.15
#> 0.150 10 3.27 0.854 2.17
#>
#> RMSE was used to select the optimal model using the smallest value.
#> The final values used for the model were sigma = 0.1 and C = 9.
plot(fit.svm)

```



4.8 Ensembling

We can try some ensemble methods on the problem and see if we can get a further decrease in our RMSE.

- Random Forest, bagging (RF).
- Gradient Boosting Machines (GBM).
- Cubist, boosting (CUBIST).

```

# try ensembles
seed <- 7
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
metric <- "RMSE"

# Random Forest
set.seed(seed)
fit.rf <- train(medv~., data=dataset, method="rf", metric=metric,
                 preProc=c("BoxCox"),
                 trControl=trainControl)

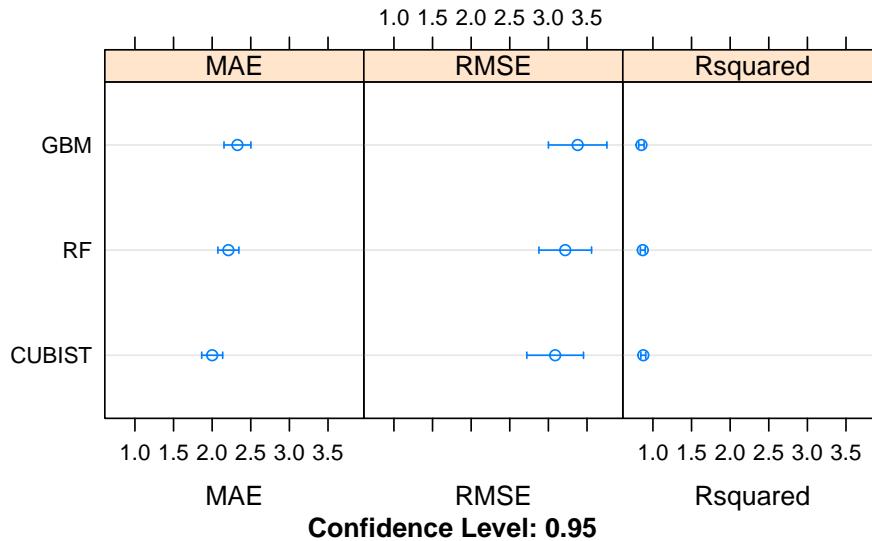
# Stochastic Gradient Boosting
set.seed(seed)
fit.gbm <- train(medv~., data=dataset, method="gbm", metric=metric,
                  preProc=c("BoxCox"),
                  trControl=trainControl, verbose=FALSE)

# Cubist
set.seed(seed)
fit.cubist <- train(medv~., data=dataset, method="cubist", metric=metric,
                     preProc=c("BoxCox"), trControl=trainControl)

# Compare algorithms
ensembleResults <- resamples(list(RF = fit.rf,
                                     GBM = fit.gbm,
                                     CUBIST = fit.cubist))

summary(ensembleResults)
#>
#> Call:
#> summary.resamples(object = ensembleResults)
#>
#> Models: RF, GBM, CUBIST
#> Number of resamples: 30
#>
#> MAE
#>      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> RF     1.64   1.98   2.20 2.21    2.30 3.22    0
#> GBM    1.65   1.98   2.27 2.33    2.55 3.75    0
#> CUBIST 1.31   1.75   1.95 2.00    2.17 2.89    0
#>
#> RMSE
#>      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> RF     2.11   2.58   3.08 3.22    3.70 6.52    0
#> GBM    1.92   2.54   3.31 3.38    3.67 6.85    0
#> CUBIST 1.79   2.38   2.74 3.09    3.78 5.79    0
#>
#> Rsquared
#>      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> RF     0.597  0.852  0.900 0.869   0.919 0.972    0
#> GBM    0.558  0.815  0.889 0.851   0.912 0.964    0
#> CUBIST 0.681  0.818  0.909 0.875   0.932 0.970    0
dotplot(ensembleResults)

```



Let's dive deeper into Cubist and see if we can tune it further and get more skill out of it. Cubist has two parameters that are tunable with caret: committees which is the number of boosting operations and neighbors which is used during prediction and is the number of instances used to correct the rule-based prediction (although the documentation is perhaps a little ambiguous on this).

```
# look at parameters used for Cubist
print(fit.cubist)
#> Cubist
#>
#> 407 samples
#> 13 predictor
#>
#> Pre-processing: Box-Cox transformation (11)
#> Resampling: Cross-Validated (10 fold, repeated 3 times)
#> Summary of sample sizes: 365, 366, 366, 367, 366, 366, ...
#> Resampling results across tuning parameters:
#>
#>   committees  neighbors  RMSE  Rsquared  MAE
#>   1            0          3.94  0.805    2.50
#>   1            5          3.66  0.828    2.24
#>   1            9          3.69  0.825    2.26
#>   10           0          3.45  0.848    2.29
#>   10           5          3.19  0.868    2.04
#>   10           9          3.23  0.864    2.07
#>   20           0          3.34  0.858    2.25
#>   20           5          3.09  0.875    2.00
#>   20           9          3.12  0.872    2.03
#>
#> RMSE was used to select the optimal model using the smallest value.
#> The final values used for the model were committees = 20 and neighbors = 5.
```

Let's use a grid search to tune around those values. We'll try all committees between 15 and 25 and spot-check a neighbors value above and below 5.

```
library(Cubist)
# Tune the Cubist algorithm
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
```

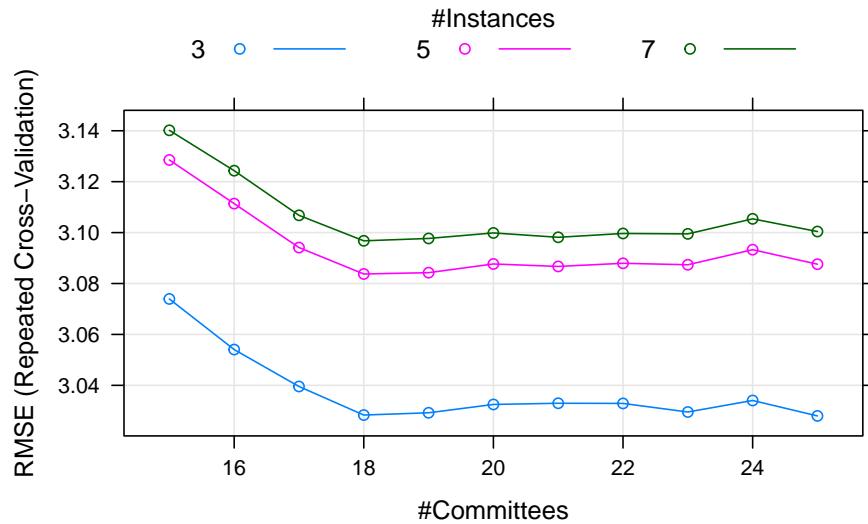
```

metric <- "RMSE"
set.seed(7)
grid <- expand.grid(.committees = seq(15, 25, by=1),
                     .neighbors = c(3, 5, 7))

tune.cubist <- train(medv~., data=dataset, method = "cubist", metric=metric,
                      preProc=c("BoxCox"),
                      tuneGrid=grid, trControl=trainControl)
print(tune.cubist)
#> Cubist
#>
#> 407 samples
#> 13 predictor
#>
#> Pre-processing: Box-Cox transformation (11)
#> Resampling: Cross-Validated (10 fold, repeated 3 times)
#> Summary of sample sizes: 365, 366, 366, 367, 366, 366, ...
#> Resampling results across tuning parameters:
#>
#>   committees  neighbors  RMSE  Rsquared  MAE
#>   15          3          3.07  0.877    2.00
#>   15          5          3.13  0.873    2.02
#>   15          7          3.14  0.871    2.03
#>   16          3          3.05  0.878    1.99
#>   16          5          3.11  0.874    2.01
#>   16          7          3.12  0.872    2.02
#>   17          3          3.04  0.879    1.98
#>   17          5          3.09  0.875    2.00
#>   17          7          3.11  0.873    2.01
#>   18          3          3.03  0.880    1.97
#>   18          5          3.08  0.876    2.00
#>   18          7          3.10  0.874    2.01
#>   19          3          3.03  0.880    1.97
#>   19          5          3.08  0.876    1.99
#>   19          7          3.10  0.874    2.01
#>   20          3          3.03  0.879    1.98
#>   20          5          3.09  0.875    2.00
#>   20          7          3.10  0.874    2.01
#>   21          3          3.03  0.879    1.98
#>   21          5          3.09  0.876    2.00
#>   21          7          3.10  0.874    2.02
#>   22          3          3.03  0.879    1.98
#>   22          5          3.09  0.875    2.00
#>   22          7          3.10  0.874    2.02
#>   23          3          3.03  0.880    1.98
#>   23          5          3.09  0.876    2.01
#>   23          7          3.10  0.874    2.02
#>   24          3          3.03  0.879    1.98
#>   24          5          3.09  0.875    2.01
#>   24          7          3.11  0.873    2.02
#>   25          3          3.03  0.880    1.98
#>   25          5          3.09  0.876    2.01
#>   25          7          3.10  0.874    2.02

```

```
#>
#> RMSE was used to select the optimal model using the smallest value.
#> The final values used for the model were committees = 25 and neighbors = 3.
plot(tune.cubist)
```



We can see that we have achieved a more accurate model again with an RMSE of 2.822 using committees = 18 and neighbors = 3.

It looks like the results for the Cubist algorithm are the most accurate. Let's finalize it by creating a new standalone Cubist model with the parameters above trained using the whole dataset. We must also use the Box-Cox power transform.

4.9 Finalize the model

```
# prepare the data transform using training data
set.seed(7)
x <- dataset[,1:13]
y <- dataset[,14]

# transform
preprocessParams <- preProcess(x, method=c("BoxCox"))
transX <- predict(preprocessParams, x)

# train the final model
finalModel <- cubist(x = transX, y=y, committees=18)
summary(finalModel)
#>
#> Call:
#> cubist.default(x = transX, y = y, committees = 18)
#>
#>
#> Cubist [Release 2.07 GPL Edition]  Fri Sep 20 14:28:05 2019
#> -----
#>
#>      Target attribute `outcome'
```

```

#>
#> Read 407 cases (14 attributes) from undefined.data
#>
#> Model 1:
#>
#> Rule 1/1: [84 cases, mean 14.29, range 5 to 27.5, est err 1.97]
#>
#> if
#> nox > -0.4864544
#> then
#> outcome = 35.08 - 2.45 crim - 4.31 lstat + 2.1e-05 b
#>
#> Rule 1/2: [163 cases, mean 19.37, range 7 to 31, est err 2.10]
#>
#> if
#> nox <= -0.4864544
#> lstat > 2.848535
#> then
#> outcome = 186.8 - 2.34 lstat - 3.3 dis - 88 tax + 2 rad + 4.4 rm
#> - 0.033 ptratio - 0.0116 age + 3.3e-05 b
#>
#> Rule 1/3: [24 cases, mean 21.65, range 18.2 to 25.3, est err 1.19]
#>
#> if
#> rm <= 3.326479
#> dis > 1.345056
#> lstat <= 2.848535
#> then
#> outcome = 43.83 + 14.5 rm - 2.29 lstat - 3.8 dis - 30 tax
#> - 0.014 ptratio - 1.4 nox + 0.017 zn + 0.4 rad + 0.15 crim
#> - 0.0025 age + 8e-06 b
#>
#> Rule 1/4: [7 cases, mean 27.66, range 20.7 to 50, est err 7.89]
#>
#> if
#> rm > 3.326479
#> ptratio > 193.545
#> lstat <= 2.848535
#> then
#> outcome = 19.64 + 7.8 rm - 3.4 dis - 1.62 lstat + 0.27 crim - 0.006 age
#> + 0.023 zn - 7 tax - 0.003 ptratio
#>
#> Rule 1/5: [141 cases, mean 30.60, range 15 to 50, est err 2.09]
#>
#> if
#> rm > 3.326479
#> ptratio <= 193.545
#> then
#> outcome = 137.95 + 21.7 rm - 3.43 lstat - 4.9 dis - 87 tax - 0.0162 age
#> - 0.039 ptratio + 0.06 crim + 0.005 zn
#>
#> Rule 1/6: [8 cases, mean 32.16, range 22.1 to 50, est err 8.67]
#>

```

```

#>      if
#> rm <= 3.326479
#> dis <= 1.345056
#> lstat <= 2.848535
#>      then
#> outcome = -19.71 + 18.58 lstat - 15.9 dis + 5.6 rm
#>
#> Model 2:
#>
#> Rule 2/1: [23 cases, mean 10.57, range 5 to 15, est err 3.06]
#>
#>      if
#> crim > 2.086391
#> dis <= 0.6604174
#> b > 67032.41
#>      then
#> outcome = 37.22 - 4.83 crim - 7 dis - 1.9 lstat - 1.9e-05 b - 0.7 rm
#>
#> Rule 2/2: [70 cases, mean 14.82, range 5 to 50, est err 3.90]
#>
#>      if
#> rm <= 3.620525
#> dis <= 0.6604174
#>      then
#> outcome = 74.6 - 21 dis - 5.09 lstat - 15 tax - 0.0017 age + 6e-06 b
#>
#> Rule 2/3: [18 cases, mean 18.03, range 7.5 to 50, est err 6.81]
#>
#>      if
#> crim > 2.086391
#> dis <= 0.6604174
#> b <= 67032.41
#>      then
#> outcome = 94.95 - 40.1 dis - 8.15 crim - 7.14 lstat - 3.5e-05 b - 1.3 rm
#>
#> Rule 2/4: [258 cases, mean 20.74, range 9.5 to 36.2, est err 1.92]
#>
#>      if
#> rm <= 3.620525
#> dis > 0.6604174
#> lstat > 1.805082
#>      then
#> outcome = 61.89 - 2.56 lstat + 5.5 rm - 2.8 dis + 7.3e-05 b - 0.0132 age
#>           - 26 tax - 0.11 indus - 0.004 ptratio + 0.05 crim
#>
#> Rule 2/5: [37 cases, mean 31.66, range 10.4 to 50, est err 3.70]
#>
#>      if
#> rm > 3.620525
#> lstat > 1.805082
#>      then
#> outcome = 370.03 - 180 tax - 2.19 lstat - 1.7 dis + 2.6 rm
#>           - 0.016 ptratio - 0.25 indus + 0.12 crim - 0.0021 age

```

```

#>           + 9e-06 b - 0.5 nox
#>
#> Rule 2/6: [42 cases, mean 38.23, range 22.8 to 50, est err 3.70]
#>
#>   if
#>   lstat <= 1.805082
#>   then
#>   outcome = -73.87 + 32.4 rm - 9.4e-05 b - 1.8 dis + 0.028 zn
#>           - 0.013 ptratio
#>
#> Rule 2/7: [4 cases, mean 40.20, range 37.6 to 42.8, est err 7.33]
#>
#>   if
#>   rm > 4.151791
#>   dis > 1.114486
#>   then
#>   outcome = 35.8
#>
#> Rule 2/8: [8 cases, mean 47.45, range 41.3 to 50, est err 10.01]
#>
#>   if
#>   dis <= 1.114486
#>   lstat <= 1.805082
#>   then
#>   outcome = 48.96 + 7.53 crim - 4.1e-05 b - 0.8 dis + 1.2 rm + 0.008 zn
#>
#> Model 3:
#>
#> Rule 3/1: [81 cases, mean 13.93, range 5 to 23.2, est err 2.24]
#>
#>   if
#>   nox > -0.4864544
#>   lstat > 2.848535
#>   then
#>   outcome = 55.03 - 0.0631 age - 2.11 crim + 12 nox - 4.16 lstat
#>           + 3.2e-05 b
#>
#> Rule 3/2: [163 cases, mean 19.37, range 7 to 31, est err 2.29]
#>
#>   if
#>   nox <= -0.4864544
#>   lstat > 2.848535
#>   then
#>   outcome = 77.73 - 0.059 ptratio + 5.8 rm - 3.2 dis - 0.0139 age
#>           - 1.15 lstat - 30 tax - 1.1 nox + 0.4 rad
#>
#> Rule 3/3: [62 cases, mean 24.01, range 18.2 to 50, est err 3.56]
#>
#>   if
#>   rm <= 3.448196
#>   lstat <= 2.848535
#>   then
#>   outcome = 94.86 + 18.2 rm + 0.63 crim - 68 tax - 2.3 dis - 3 nox

```

```

#>           - 0.0098 age - 0.41 indus - 0.011 ptratio
#>
#> Rule 3/4: [143 cases, mean 28.76, range 16.5 to 50, est err 2.53]
#>
#>   if
#>   dis > 0.9547035
#>   lstat <= 2.848535
#>   then
#>   outcome = 269.46 + 17.9 rm - 6.1 dis - 153 tax + 0.96 crim - 0.0217 age
#>           - 5.5 nox - 0.62 indus - 0.028 ptratio - 0.89 lstat + 0.4 rad
#>           + 0.004 zn
#>
#> Rule 3/5: [10 cases, mean 35.13, range 21.9 to 50, est err 9.31]
#>
#>   if
#>   dis <= 0.6492998
#>   lstat <= 2.848535
#>   then
#>   outcome = 58.69 - 56.8 dis - 8.4 nox
#>
#> Rule 3/6: [10 cases, mean 41.67, range 22 to 50, est err 9.89]
#>
#>   if
#>   dis > 0.6492998
#>   dis <= 0.9547035
#>   lstat <= 2.848535
#>   then
#>   outcome = 47.93
#>
#> Model 4:
#>
#> Rule 4/1: [69 cases, mean 12.69, range 5 to 27.5, est err 2.55]
#>
#>   if
#>   dis <= 0.719156
#>   lstat > 3.508535
#>   then
#>   outcome = 180.13 - 7.2 dis + 0.039 age - 3.78 lstat - 83 tax
#>
#> Rule 4/2: [164 cases, mean 19.42, range 12 to 31, est err 1.96]
#>
#>   if
#>   dis > 0.719156
#>   lstat > 2.848535
#>   then
#>   outcome = 52.75 + 7.1 rm - 2.05 lstat - 3.6 dis + 8.2e-05 b - 0.0152 age
#>           - 25 tax + 0.5 rad - 1.2 nox - 0.008 ptratio
#>
#> Rule 4/3: [11 cases, mean 20.39, range 15 to 27.9, est err 3.51]
#>
#>   if
#>   dis <= 0.719156
#>   lstat > 2.848535

```

```

#> lstat <= 3.508535
#>   then
#> outcome = 21.69
#>
#> Rule 4/4: [63 cases, mean 23.22, range 16.5 to 31.5, est err 1.67]
#>
#>   if
#> rm <= 3.483629
#> dis > 0.9731624
#> lstat <= 2.848535
#>   then
#> outcome = 59.35 - 3.96 lstat - 3.1 dis + 1 rm - 14 tax + 0.3 rad
#>           - 0.7 nox - 0.005 ptratio + 6e-06 b
#>
#> Rule 4/5: [8 cases, mean 33.08, range 22 to 50, est err 23.91]
#>
#>   if
#> rm > 3.369183
#> dis <= 0.9731624
#> lstat > 2.254579
#> lstat <= 2.848535
#>   then
#> outcome = -322.28 + 64.9 lstat + 56.8 rm - 30.2 dis
#>
#> Rule 4/6: [7 cases, mean 33.87, range 22.1 to 50, est err 13.21]
#>
#>   if
#> rm <= 3.369183
#> dis <= 0.9731624
#> lstat <= 2.848535
#>   then
#> outcome = -52.11 + 43.45 lstat - 30.8 dis
#>
#> Rule 4/7: [91 cases, mean 34.43, range 21.9 to 50, est err 3.32]
#>
#>   if
#> rm > 3.483629
#> lstat <= 2.848535
#>   then
#> outcome = -33.09 + 22 rm - 5.02 lstat - 0.038 ptratio - 0.9 dis
#>           + 0.005 zn
#>
#> Rule 4/8: [22 cases, mean 36.99, range 21.9 to 50, est err 13.21]
#>
#>   if
#> dis <= 0.9731624
#> lstat <= 2.848535
#>   then
#> outcome = 80.3 - 17.43 lstat - 0.134 ptratio + 2.5 rm - 1.2 dis
#>           + 0.008 zn
#>
#> Model 5:
#>

```

```

#> Rule 5/1: [84 cases, mean 14.29, range 5 to 27.5, est err 2.81]
#>
#>   if
#>   nox > -0.4864544
#>   then
#>   outcome = 56.48 + 28.5 nox - 0.0875 age - 3.58 crim - 5.9 dis
#>           - 2.96 lstat + 0.073 ptratio + 1.7e-05 b
#>
#> Rule 5/2: [163 cases, mean 19.37, range 7 to 31, est err 2.38]
#>
#>   if
#>   nox <= -0.4864544
#>   lstat > 2.848535
#>   then
#>   outcome = 61.59 - 0.064 ptratio + 5.9 rm - 3.1 dis - 0.0142 age
#>           - 0.77 lstat - 21 tax
#>
#> Rule 5/3: [163 cases, mean 29.94, range 16.5 to 50, est err 3.65]
#>
#>   if
#>   lstat <= 2.848535
#>   then
#>   outcome = 264.17 + 21.9 rm - 8 dis - 155 tax - 0.0317 age
#>           - 0.032 ptratio + 0.29 crim - 1.6 nox - 0.25 indus
#>
#> Rule 5/4: [10 cases, mean 35.13, range 21.9 to 50, est err 11.79]
#>
#>   if
#>   dis <= 0.6492998
#>   lstat <= 2.848535
#>   then
#>   outcome = 68.19 - 73.4 dis + 1.1 rm + 0.11 crim - 0.6 nox - 0.1 indus
#>           - 0.0017 age - 0.12 lstat
#>
#> Model 6:
#>
#> Rule 6/1: [71 cases, mean 15.57, range 5 to 50, est err 4.42]
#>
#>   if
#>   dis <= 0.6443245
#>   lstat > 1.793385
#>   then
#>   outcome = 45.7 - 20.6 dis - 5.38 lstat
#>
#> Rule 6/2: [159 cases, mean 19.53, range 8.3 to 36.2, est err 2.08]
#>
#>   if
#>   rm <= 3.329365
#>   dis > 0.6443245
#>   then
#>   outcome = 24.33 + 8.8 rm + 0.000118 b - 0.0146 age - 2.5 dis
#>           - 0.95 lstat + 0.37 crim - 0.32 indus + 0.02 zn - 16 tax
#>           + 0.2 rad - 0.5 nox - 0.004 ptratio

```

```

#>
#>   Rule 6/3: [175 cases, mean 27.80, range 9.5 to 50, est err 2.95]
#>
#>     if
#>     rm > 3.329365
#>     dis > 0.6443245
#>     then
#>     outcome = 0.11 + 18.7 rm - 3.11 lstat + 8.1e-05 b - 1.1 dis + 0.19 crim
#>             - 20 tax - 0.19 indus + 0.3 rad - 0.7 nox - 0.005 ptratio
#>             + 0.006 zn
#>
#>   Rule 6/4: [8 cases, mean 32.50, range 21.9 to 50, est err 10.34]
#>
#>     if
#>     dis <= 0.6443245
#>     lstat > 1.793385
#>     lstat <= 2.894121
#>     then
#>     outcome = 69.38 - 71.2 dis - 0.14 lstat
#>
#>   Rule 6/5: [34 cases, mean 37.55, range 22.8 to 50, est err 3.55]
#>
#>     if
#>     rm <= 4.151791
#>     lstat <= 1.793385
#>     then
#>     outcome = -125.14 + 41.7 rm + 4.3 rad + 1.48 indus - 0.014 ptratio
#>
#>   Rule 6/6: [7 cases, mean 43.66, range 37.6 to 50, est err 3.12]
#>
#>     if
#>     rm > 4.151791
#>     lstat <= 1.793385
#>     then
#>     outcome = -137.67 + 44.6 rm - 0.064 ptratio
#>
#> Model 7:
#>
#>   Rule 7/1: [84 cases, mean 14.29, range 5 to 27.5, est err 2.91]
#>
#>     if
#>     nox > -0.4864544
#>     then
#>     outcome = 46.85 - 3.45 crim - 0.0621 age + 14.2 nox + 4.4 dis
#>             - 2.01 lstat + 2.5e-05 b
#>
#>   Rule 7/2: [323 cases, mean 24.66, range 7 to 50, est err 3.68]
#>
#>     if
#>     nox <= -0.4864544
#>     then
#>     outcome = 57.59 - 0.065 ptratio - 4.4 dis + 6.8 rm - 0.0143 age
#>             - 1.36 lstat - 19 tax - 0.8 nox - 0.12 crim + 0.09 indus

```

```

#>
#>   Rule 7/3: [132 cases, mean 28.24, range 16.5 to 50, est err 2.55]
#>
#>     if
#>     dis > 1.063503
#>     lstat <= 2.848535
#>     then
#>     outcome = 270.92 + 24.5 rm - 0.0418 age - 165 tax - 5.7 dis
#>             - 0.028 ptratio + 0.26 crim + 0.017 zn
#>
#>   Rule 7/4: [7 cases, mean 36.01, range 23.3 to 50, est err 3.87]
#>
#>     if
#>     dis <= 0.6002641
#>     lstat <= 2.848535
#>     then
#>     outcome = 57.18 - 69.5 dis - 6.5 nox + 1.9 rm - 0.015 ptratio
#>
#>   Rule 7/5: [24 cases, mean 37.55, range 21.9 to 50, est err 8.66]
#>
#>     if
#>     dis > 0.6002641
#>     dis <= 1.063503
#>     lstat <= 2.848535
#>     then
#>     outcome = -3.76 - 14.8 dis - 2.93 crim - 0.16 ptratio + 17.5 rm - 15 nox
#>
#> Model 8:
#>
#>   Rule 8/1: [80 cases, mean 13.75, range 5 to 27.9, est err 3.51]
#>
#>     if
#>     dis <= 0.719156
#>     lstat > 2.848535
#>     then
#>     outcome = 123.46 - 11.3 dis - 5.06 lstat - 45 tax + 0.9 rad + 1.7e-05 b
#>
#>   Rule 8/2: [164 cases, mean 19.42, range 12 to 31, est err 2.05]
#>
#>     if
#>     dis > 0.719156
#>     lstat > 2.848535
#>     then
#>     outcome = 227.11 - 120 tax + 6.4 rm + 9.3e-05 b - 3.3 dis + 2 rad
#>             - 0.0183 age - 0.93 lstat + 0.05 crim - 0.3 nox
#>
#>   Rule 8/3: [163 cases, mean 29.94, range 16.5 to 50, est err 3.54]
#>
#>     if
#>     lstat <= 2.848535
#>     then
#>     outcome = 158.14 - 5.73 lstat + 10.8 rm - 4 dis - 83 tax - 4.1 nox
#>             + 0.61 crim - 0.54 indus + 1 rad + 3.6e-05 b

```

```

#>
#>   Rule 8/4: [7 cases, mean 36.01, range 23.3 to 50, est err 11.44]
#>
#>     if
#>     dis <= 0.6002641
#>     lstat <= 2.848535
#>     then
#>     outcome = 72.89 - 87.2 dis + 0.6 rm - 0.13 lstat
#>
#>   Rule 8/5: [47 cases, mean 38.44, range 15 to 50, est err 5.71]
#>
#>     if
#>     rm > 3.726352
#>     then
#>     outcome = 602.95 - 10.4 lstat + 21 rm - 326 tax - 0.093 ptratio
#>
#> Model 9:
#>
#>   Rule 9/1: [81 cases, mean 13.93, range 5 to 23.2, est err 2.91]
#>
#>     if
#>     nox > -0.4864544
#>     lstat > 2.848535
#>     then
#>     outcome = 41.11 - 3.98 crim - 4.42 lstat + 6.7 nox
#>
#>   Rule 9/2: [163 cases, mean 19.37, range 7 to 31, est err 2.49]
#>
#>     if
#>     nox <= -0.4864544
#>     lstat > 2.848535
#>     then
#>     outcome = 44.98 - 0.068 ptratio - 4.4 dis + 6.6 rm - 1.25 lstat
#>             - 0.0118 age - 0.9 nox - 12 tax - 0.08 crim + 0.06 indus
#>
#>   Rule 9/3: [132 cases, mean 28.24, range 16.5 to 50, est err 2.35]
#>
#>     if
#>     dis > 1.063503
#>     lstat <= 2.848535
#>     then
#>     outcome = 157.67 + 22.2 rm - 0.0383 age - 104 tax - 0.033 ptratio
#>             - 2.2 dis
#>
#>   Rule 9/4: [7 cases, mean 30.76, range 21.9 to 50, est err 6.77]
#>
#>     if
#>     dis <= 1.063503
#>     b <= 66469.73
#>     lstat <= 2.848535
#>     then
#>     outcome = 48.52 - 56.1 dis - 12.9 nox - 0.032 ptratio + 2.7 rm
#>

```

```

#>   Rule 9/5: [24 cases, mean 39.09, range 22 to 50, est err 6.20]
#>
#>     if
#>     dis <= 1.063503
#>     b > 66469.73
#>     lstat <= 2.848535
#>     then
#>     outcome = -5.49 - 34.8 dis - 20.7 nox + 18.2 rm - 0.051 ptratio
#>
#> Model 10:
#>
#>   Rule 10/1: [327 cases, mean 19.45, range 5 to 50, est err 2.77]
#>
#>     if
#>     rm <= 3.617282
#>     lstat > 1.805082
#>     then
#>     outcome = 270.78 - 4.09 lstat - 131 tax + 2.9 rad + 5.3e-05 b - 0.6 dis
#>             - 0.16 indus + 0.7 rm - 0.3 nox
#>
#>   Rule 10/2: [38 cases, mean 31.57, range 10.4 to 50, est err 4.71]
#>
#>     if
#>     rm > 3.617282
#>     lstat > 1.805082
#>     then
#>     outcome = 308.44 - 150 tax - 2.63 lstat + 1.6 rad - 1.9 dis - 0.49 indus
#>             + 2.5 rm + 3e-05 b - 1.2 nox + 0.14 crim - 0.005 ptratio
#>
#>   Rule 10/3: [35 cases, mean 37.15, range 22.8 to 50, est err 2.76]
#>
#>     if
#>     rm <= 4.151791
#>     lstat <= 1.805082
#>     then
#>     outcome = -71.65 + 33.4 rm - 0.017 ptratio - 0.34 lstat + 0.2 rad
#>             - 0.3 dis - 7 tax - 0.4 nox
#>
#>   Rule 10/4: [10 cases, mean 42.63, range 21.9 to 50, est err 7.11]
#>
#>     if
#>     rm > 4.151791
#>     then
#>     outcome = -92.51 + 32.8 rm - 0.03 ptratio
#>
#> Model 11:
#>
#>   Rule 11/1: [84 cases, mean 14.29, range 5 to 27.5, est err 4.13]
#>
#>     if
#>     nox > -0.4864544
#>     then
#>     outcome = 42.75 - 4.12 crim + 18.1 nox - 0.045 age + 6.8 dis

```

```

#>           - 1.86 lstat
#>
#> Rule 11/2: [244 cases, mean 17.56, range 5 to 31, est err 4.29]
#>
#>   if
#>   lstat > 2.848535
#>   then
#>   outcome = 34.83 - 5.2 dis - 0.058 ptratio - 0.0228 age + 5.8 rm
#>           - 0.56 lstat - 0.07 crim - 0.4 nox - 5 tax
#>
#> Rule 11/3: [163 cases, mean 29.94, range 16.5 to 50, est err 3.49]
#>
#>   if
#>   lstat <= 2.848535
#>   then
#>   outcome = 151.5 + 23.3 rm - 5.5 dis + 1.01 crim - 0.0211 age
#>           - 0.052 ptratio - 98 tax + 0.031 zn
#>
#> Rule 11/4: [10 cases, mean 35.13, range 21.9 to 50, est err 25.19]
#>
#>   if
#>   dis <= 0.6492998
#>   lstat <= 2.848535
#>   then
#>   outcome = 130.87 - 157.1 dis - 15.76 crim
#>
#> Model 12:
#>
#> Rule 12/1: [80 cases, mean 13.75, range 5 to 27.9, est err 4.76]
#>
#>   if
#>   dis <= 0.719156
#>   lstat > 2.894121
#>   then
#>   outcome = 182.68 - 6.03 lstat - 7.6 dis - 76 tax + 1.3 rad - 0.52 indus
#>           + 2.6e-05 b
#>
#> Rule 12/2: [300 cases, mean 19.10, range 5 to 50, est err 2.76]
#>
#>   if
#>   rm <= 3.50716
#>   lstat > 1.793385
#>   then
#>   outcome = 83.61 - 3 lstat + 9.6e-05 b - 0.0072 age - 33 tax + 0.7 rad
#>           + 0.32 indus
#>
#> Rule 12/3: [10 cases, mean 24.25, range 15.7 to 36.2, est err 13.88]
#>
#>   if
#>   rm <= 3.50716
#>   tax <= 1.865769
#>   then
#>   outcome = 35.46

```

```

#>
#>   Rule 12/4: [10 cases, mean 32.66, range 21.9 to 50, est err 6.28]
#>
#>     if
#>     dis <= 0.719156
#>     lstat > 1.793385
#>     lstat <= 2.894121
#>     then
#>     outcome = 82.78 - 69.5 dis - 3.66 indus
#>
#>   Rule 12/5: [89 cases, mean 32.75, range 13.4 to 50, est err 3.39]
#>
#>     if
#>     rm > 3.50716
#>     dis > 0.719156
#>     then
#>     outcome = 313.22 + 13.7 rm - 174 tax - 3.06 lstat + 4.8e-05 b - 1.5 dis
#>           - 0.41 indus + 0.7 rad - 0.0055 age + 0.22 crim
#>
#>   Rule 12/6: [34 cases, mean 37.55, range 22.8 to 50, est err 3.25]
#>
#>     if
#>     rm <= 4.151791
#>     lstat <= 1.793385
#>     then
#>     outcome = -86.8 + 36 rm - 0.3 lstat - 5 tax
#>
#>   Rule 12/7: [7 cases, mean 43.66, range 37.6 to 50, est err 5.79]
#>
#>     if
#>     rm > 4.151791
#>     lstat <= 1.793385
#>     then
#>     outcome = -158.68 + 47.4 rm - 0.02 ptratio
#>
#> Model 13:
#>
#>   Rule 13/1: [84 cases, mean 14.29, range 5 to 27.5, est err 2.87]
#>
#>     if
#>     nox > -0.4864544
#>     then
#>     outcome = 54.69 - 3.79 crim - 0.0644 age + 11.4 nox - 2.53 lstat
#>
#>   Rule 13/2: [8 cases, mean 17.76, range 7 to 27.9, est err 13.69]
#>
#>     if
#>     nox <= -0.4864544
#>     age > 296.3423
#>     b <= 60875.57
#>     then
#>     outcome = -899.55 + 3.0551 age
#>

```

```

#>   Rule 13/3: [31 cases, mean 17.94, range 7 to 27.9, est err 5.15]
#>
#>     if
#>     nox <= -0.4864544
#>     b <= 60875.57
#>     lstat > 2.848535
#>     then
#>     outcome = 44.43 - 3.51 lstat - 0.054 ptratio - 1.4 dis - 0.26 crim
#>             - 0.0042 age - 0.21 indus + 0.9 rm
#>
#>   Rule 13/4: [163 cases, mean 19.37, range 7 to 31, est err 3.37]
#>
#>     if
#>     nox <= -0.4864544
#>     lstat > 2.848535
#>     then
#>     outcome = -5.76 + 0.000242 b + 8.9 rm - 5.2 dis - 0.0209 age
#>             - 0.042 ptratio - 0.63 indus
#>
#>   Rule 13/5: [163 cases, mean 29.94, range 16.5 to 50, est err 3.45]
#>
#>     if
#>     lstat <= 2.848535
#>     then
#>     outcome = 178.84 + 23.8 rm - 0.0343 age - 4.5 dis - 114 tax + 0.88 crim
#>             - 0.048 ptratio + 0.026 zn
#>
#>   Rule 13/6: [7 cases, mean 36.01, range 23.3 to 50, est err 14.09]
#>
#>     if
#>     dis <= 0.6002641
#>     lstat <= 2.848535
#>     then
#>     outcome = 45.82 - 70.3 dis - 9.9 nox + 5.1 rm + 1.5 rad
#>
#>   Rule 13/7: [31 cases, mean 37.21, range 21.9 to 50, est err 7.73]
#>
#>     if
#>     dis <= 1.063503
#>     lstat <= 2.848535
#>     then
#>     outcome = 95.05 - 4.52 lstat - 7.5 dis + 8.8 rm - 0.064 ptratio
#>             - 6.2 nox - 36 tax
#>
#>   Model 14:
#>
#>   Rule 14/1: [49 cases, mean 16.06, range 8.4 to 22.7, est err 3.17]
#>
#>     if
#>     nox > -0.4205732
#>     lstat > 2.848535
#>     then
#>     outcome = 12.83 + 42.3 nox - 4.77 lstat + 9.7 rm + 7.8e-05 b

```

```

#>
#>   Rule 14/2: [78 cases, mean 16.36, range 5 to 50, est err 5.17]
#>
#>     if
#>     dis <= 0.6604174
#>     then
#>       outcome = 110.6 - 10.4 dis - 4.85 lstat + 0.0446 age - 46 tax + 0.8 rad
#>
#>   Rule 14/3: [57 cases, mean 18.40, range 9.5 to 31, est err 2.43]
#>
#>     if
#>     nox > -0.9365134
#>     nox <= -0.4205732
#>     age > 245.2507
#>     dis > 0.6604174
#>     lstat > 2.848535
#>     then
#>       outcome = 206.69 - 0.1012 age - 7.05 lstat + 12.2 nox - 67 tax + 0.3 rad
#>           + 0.5 rm - 0.3 dis
#>
#>   Rule 14/4: [230 cases, mean 20.19, range 9.5 to 36.2, est err 2.09]
#>
#>     if
#>     rm <= 3.483629
#>     dis > 0.6492998
#>     then
#>       outcome = 119.15 - 2.61 lstat + 5.2 rm - 57 tax - 1.8 dis - 2.4 nox
#>           + 0.7 rad + 0.24 crim + 0.003 age - 0.007 ptratio + 9e-06 b
#>
#>   Rule 14/5: [48 cases, mean 20.28, range 10.2 to 24.5, est err 2.13]
#>
#>     if
#>     nox > -0.9365134
#>     nox <= -0.4205732
#>     age <= 245.2507
#>     dis > 0.6604174
#>     lstat > 2.848535
#>     then
#>       outcome = 19.4 - 1.91 lstat + 1.02 indus - 0.013 age + 2.7 rm + 2.6 nox
#>           - 0.009 ptratio
#>
#>   Rule 14/6: [44 cases, mean 20.69, range 14.4 to 29.6, est err 2.26]
#>
#>     if
#>     nox <= -0.9365134
#>     lstat > 2.848535
#>     then
#>       outcome = 87.55 - 0.000315 b - 6.5 dis + 2.6 rad - 0.59 lstat - 18 tax
#>
#>   Rule 14/7: [102 cases, mean 32.44, range 13.4 to 50, est err 3.35]
#>
#>     if
#>     rm > 3.483629

```

```

#> dis > 0.6492998
#>   then
#> outcome = 126.92 + 22.7 rm - 4.68 lstat - 85 tax - 0.036 ptratio
#>           - 1.1 dis + 0.007 zn
#>
#> Rule 14/8: [84 cases, mean 33.40, range 21 to 50, est err 2.44]
#>
#>   if
#> rm > 3.483629
#> tax <= 1.896025
#>   then
#> outcome = 347.12 + 25.2 rm - 213 tax - 3.5 lstat - 0.013 ptratio
#>
#> Rule 14/9: [10 cases, mean 35.13, range 21.9 to 50, est err 12.13]
#>
#>   if
#> dis <= 0.6492998
#> lstat <= 2.848535
#>   then
#> outcome = 72.65 - 77.8 dis
#>
#> Model 15:
#>
#> Rule 15/1: [28 cases, mean 12.35, range 5 to 27.9, est err 4.09]
#>
#>   if
#> crim > 2.405809
#> b > 16084.5
#>   then
#> outcome = 53.45 - 7.8 crim - 3.5 lstat - 0.0189 age
#>
#> Rule 15/2: [11 cases, mean 13.56, range 8.3 to 27.5, est err 5.99]
#>
#>   if
#> crim > 2.405809
#> b <= 16084.5
#>   then
#> outcome = 8.73 + 0.001756 b
#>
#> Rule 15/3: [244 cases, mean 17.56, range 5 to 31, est err 2.73]
#>
#>   if
#> lstat > 2.848535
#>   then
#> outcome = 103.02 - 0.0251 age - 2.37 lstat - 3.5 dis + 6.8e-05 b + 4 rm
#>           - 0.035 ptratio - 41 tax - 0.25 crim
#>
#> Rule 15/4: [131 cases, mean 28.22, range 16.5 to 50, est err 2.59]
#>
#>   if
#> dis > 1.086337
#> lstat <= 2.848535
#>   then

```

```

#> outcome = 267.07 + 17.7 rm - 0.0421 age - 150 tax - 5.5 dis + 0.88 crim
#>           - 0.035 ptratio + 0.031 zn - 0.12 lstat - 0.3 nox
#>
#> Rule 15/5: [13 cases, mean 33.08, range 22 to 50, est err 4.44]
#>
#>     if
#>     nox <= -0.7229691
#>     dis <= 1.086337
#>     lstat <= 2.848535
#>     then
#>     outcome = 148.52 - 0.002365 b - 85.9 nox - 1 dis + 0.16 crim + 0.8 rm
#>           + 0.007 zn - 0.0016 age - 7 tax - 0.003 ptratio
#>
#> Rule 15/6: [7 cases, mean 36.01, range 23.3 to 50, est err 7.00]
#>
#>     if
#>     dis <= 0.6002641
#>     lstat <= 2.848535
#>     then
#>     outcome = 50.55 - 68.1 dis - 11.4 nox + 0.00012 b + 1 rm - 0.008 ptratio
#>
#> Rule 15/7: [12 cases, mean 41.77, range 21.9 to 50, est err 9.73]
#>
#>     if
#>     nox > -0.7229691
#>     dis > 0.6002641
#>     lstat <= 2.848535
#>     then
#>     outcome = 13.74 - 92 nox - 40.5 dis - 0.023 ptratio + 2.6 rm
#>
#> Model 16:
#>
#> Rule 16/1: [60 cases, mean 15.95, range 7.2 to 27.5, est err 3.16]
#>
#>     if
#>     nox > -0.4344906
#>     then
#>     outcome = 46.98 - 6.53 lstat - 6.9 dis - 1.1 rm
#>
#> Rule 16/2: [45 cases, mean 16.89, range 5 to 50, est err 5.45]
#>
#>     if
#>     nox <= -0.4344906
#>     dis <= 0.6557049
#>     then
#>     outcome = 35.33 - 37 dis - 51.7 nox - 7.38 lstat - 0.4 rm
#>
#> Rule 16/3: [128 cases, mean 19.97, range 9.5 to 36.2, est err 2.52]
#>
#>     if
#>     rm <= 3.626081
#>     dis > 0.6557049
#>     dis <= 1.298828

```

```

#> lstat > 2.133251
#>   then
#>   outcome = 61.65 - 3.35 lstat + 4.9 dis + 1.6 rm - 1.3 nox - 22 tax
#>           + 0.5 rad + 1.8e-05 b + 0.09 crim - 0.004 ptratio
#>
#> Rule 16/4: [140 cases, mean 21.93, range 12.7 to 35.1, est err 2.19]
#>
#>   if
#>   rm <= 3.626081
#>   dis > 1.298828
#>   then
#>   outcome = 54.16 - 3.58 lstat + 2.2 rad - 1.6 dis - 1.9 nox + 1.8 rm
#>           - 17 tax + 1.3e-05 b + 0.06 crim - 0.003 ptratio
#>
#> Rule 16/5: [30 cases, mean 21.97, range 14.4 to 29.1, est err 2.41]
#>
#>   if
#>   rm <= 3.626081
#>   dis > 1.298828
#>   tax <= 1.879832
#>   lstat > 2.133251
#>   then
#>   outcome = -1065.35 + 566 tax + 8.7 rm - 0.13 lstat - 0.2 dis - 0.3 nox
#>
#> Rule 16/6: [22 cases, mean 30.88, range 10.4 to 50, est err 4.51]
#>
#>   if
#>   rm > 3.626081
#>   lstat > 2.133251
#>   then
#>   outcome = 42.24 + 18.7 rm - 1.5 indus - 1.84 lstat - 2.5 nox - 1.6 dis
#>           - 39 tax + 0.7 rad - 0.012 ptratio + 0.0035 age + 1.2e-05 b
#>           + 0.11 crim
#>
#> Rule 16/7: [73 cases, mean 34.52, range 20.6 to 50, est err 3.36]
#>
#>   if
#>   lstat <= 2.133251
#>   then
#>   outcome = 50.6 + 19.6 rm - 2.77 lstat - 3.2 nox - 1.7 dis - 45 tax
#>           + 1 rad + 0.007 age - 0.014 ptratio
#>
#> Model 17:
#>
#> Rule 17/1: [116 cases, mean 15.37, range 5 to 27.9, est err 2.55]
#>
#>   if
#>   crim > 0.4779842
#>   lstat > 2.944963
#>   then
#>   outcome = 35.96 - 3.68 crim - 3.41 lstat + 0.3 nox
#>
#> Rule 17/2: [112 cases, mean 19.13, range 7 to 31, est err 2.14]

```

```

#>
#>   if
#>   crim <= 0.4779842
#>   lstat > 2.944963
#>   then
#>   outcome = 184.65 - 0.0365 age + 9 rm - 4.1 dis - 97 tax + 8.4e-05 b
#>           - 0.024 ptratio
#>
#> Rule 17/3: [9 cases, mean 28.37, range 15 to 50, est err 11.17]
#>
#>   if
#>   dis <= 0.9547035
#>   b <= 66469.73
#>   lstat <= 2.944963
#>   then
#>   outcome = -1.12 + 0.000454 b
#>
#> Rule 17/4: [179 cases, mean 29.28, range 15 to 50, est err 3.35]
#>
#>   if
#>   lstat <= 2.944963
#>   then
#>   outcome = 278.16 + 20 rm - 7.4 dis - 0.0356 age - 161 tax + 0.051 zn
#>           - 0.61 lstat + 0.17 crim - 0.008 ptratio
#>
#> Rule 17/5: [23 cases, mean 36.10, range 15 to 50, est err 10.83]
#>
#>   if
#>   dis <= 0.9547035
#>   lstat <= 2.944963
#>   then
#>   outcome = 233.74 - 8.5 dis + 12.1 rm + 1.15 crim - 2.42 lstat - 113 tax
#>           - 0.0221 age + 0.068 zn - 0.031 ptratio
#>
#> Model 18:
#>
#> Rule 18/1: [84 cases, mean 14.29, range 5 to 27.5, est err 2.44]
#>
#>   if
#>   nox > -0.4864544
#>   then
#>   outcome = 41.55 - 6.2 lstat + 14.6 nox + 3.8e-05 b
#>
#> Rule 18/2: [163 cases, mean 19.37, range 7 to 31, est err 2.44]
#>
#>   if
#>   nox <= -0.4864544
#>   lstat > 2.848535
#>   then
#>   outcome = 172.79 - 3.67 lstat + 3.1 rad - 3.5 dis - 72 tax - 0.72 indus
#>           - 0.033 ptratio - 1.2 nox + 0.0027 age + 0.6 rm + 0.05 crim
#>           + 5e-06 b
#>

```

```

#>   Rule 18/3: [106 cases, mean 25.41, range 16.5 to 50, est err 2.76]
#>
#>     if
#>     rm <= 3.626081
#>     lstat <= 2.848535
#>     then
#>     outcome = 10.71 - 4.6 dis - 2.21 lstat + 2.3 rad + 5.5 rm - 5.3 nox
#>             - 0.83 indus - 0.003 ptratio
#>
#>   Rule 18/4: [4 cases, mean 33.47, range 30.1 to 36.2, est err 5.61]
#>
#>     if
#>     rm <= 3.626081
#>     tax <= 1.863917
#>     lstat <= 2.848535
#>     then
#>     outcome = 36.84
#>
#>   Rule 18/5: [10 cases, mean 35.13, range 21.9 to 50, est err 17.40]
#>
#>     if
#>     dis <= 0.6492998
#>     lstat <= 2.848535
#>     then
#>     outcome = 84.58 - 94.7 dis - 0.15 lstat
#>
#>   Rule 18/6: [57 cases, mean 38.38, range 21.9 to 50, est err 3.97]
#>
#>     if
#>     rm > 3.626081
#>     lstat <= 2.848535
#>     then
#>     outcome = 100.34 + 22.3 rm - 5.79 lstat - 0.062 ptratio - 69 tax
#>             + 0.3 rad - 0.5 nox - 0.3 dis + 0.0011 age
#>
#>
#> Evaluation on training data (407 cases):
#>
#>     Average |error|           1.72
#>     Relative |error|          0.26
#>     Correlation coefficient  0.96
#>
#>
#> Attribute usage:
#>     Conds Model
#>
#>     72%    84%    lstat
#>     38%    85%    dis
#>     35%    80%    rm
#>     27%    55%    nox
#>     4%     58%    crim
#>     2%     49%    b
#>     2%     68%    ptratio

```

```
#>      1%    78%   tax
#>      1%    67%   age
#>      41%   rad
#>      36%   indus
#>      20%   zn
#>
#>
#> Time: 0.2 secs
```

We can now use this model to evaluate our held-out validation dataset. Again, we must prepare the input data using the same Box-Cox transform.

```
# transform the validation dataset
set.seed(7)
valX <- validation[,1:13]
trans_valX <- predict(preprocessParams, valX)
valY <- validation[,14]

# use final model to make predictions on the validation dataset
predictions <- predict(finalModel, newdata = trans_valX, neighbors=3)

# calculate RMSE
rmse <- RMSE(predictions, valY)
r2 <- R2(predictions, valY)
print(rmse)
#> [1] 3.24
```

We can see that the estimated RMSE on this unseen data is about 2.666, lower but not too dissimilar from our expected RMSE of 2.822.

Chapter 5

Classification algorithms comparison. Breast cancer dataset, (*LG, LDA, GLMNET, KNN, CARTM NB, SVM*)

5.1 BreastCancer dataset

5.2 Introduction

In this classification problem we apply these algorithms:

- Linear
 - 1. LG (logistic regression)
 - 2. LDA (linear discriminant analysis)
 - 3. GLMNET (Regularized logistic regression)
- Non-linear
 - 4. KNN (k-Nearest Neighbors)
 - 5. CART (Classification and Regression Trees)
 - 6. NB (Naive Bayes)
 - 7. SVM (Support Vector Machines)

```
# load packages
library(mlbench)
library(caret)
#> Loading required package: lattice
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures    rlang
#>   c.quosures    rlang
#>   print.quosures rlang
library(tictoc)

# Load data
```

```
data(BreastCancer)
```

5.3 Workflow

1. Load dataset
2. Create train and validation datasets, 80/20
3. Inspect dataset:

- dimension
 - class of variables
 - `skimr`
4. Clean up features
 - Convert character to numeric
 - Frequency table on class
 - remove NAs
 5. Visualize features
 - histograms (loop on variables)
 - density plots (loop)
 - boxplot (loop)
 - Pairwise jittered plot
 - Barplots for all features (loop)

6. Train as-is
 - Set the train control to
 - 10 cross-validations
 - 3 repetitions
 - Metric: Accuracy
 - Train the models
 - Numeric comparison of model results
 - Visual comparison
 - dot plot
7. Train with data transformation
 - data transformation
 - BoxCox
 - Train models
 - Numeric comparison
 - Visual comparison
 - dot plot

8. Tune the best model: SVM
 - Set the train control to
 - 10 cross-validations
 - 3 repetitions
 - Metric: Accuracy
 - Train the models
 - Radial SVM
 - Sigma vector
 - `.C`

- BoxCox
 - Evaluate tuning parameters
9. Tune the best model: KNN
- Set the train control to
 - 10 cross-validations
 - 3 repetitions
 - Metric: Accuracy
 - Train the models
 - .k
 - BoxCox
 - Evaluate tuning parameters
 - Scatter plot 10, Ensembling
 - Select the algorithms
 - Bagged CART
 - Random Forest
 - Stochastic Gradient Boosting
 - C5.0
 - Numeric comparison
 - resamples
 - summary
 - Visual comparison
 - dot plot
11. Finalize the model
- Back transformation
 - preProcess
 - predict
12. Apply model to validation set
- Prepare validation set
 - Transform the dataset
 - Make prediction
 - knn3Train
 - Calculate accuracy
 - Confusion Matrix

5.4 Inspect the dataset

```
dplyr::glimpse(BreastCancer)
#> Observations: 699
#> Variables: 11
#> $ Id              <chr> "1000025", "1002945", "1015425", "1016277", "1...
#> $ Cl.thickness    <ord> 5, 5, 3, 6, 4, 8, 1, 2, 2, 4, 1, 2, 5, 1, 8, 7...
#> $ Cell.size        <ord> 1, 4, 1, 8, 1, 10, 1, 1, 2, 1, 1, 3, 1, 7, ...
#> $ Cell.shape       <ord> 1, 4, 1, 8, 1, 10, 1, 2, 1, 1, 1, 1, 3, 1, 5, ...
#> $ Marg.adhesion   <ord> 1, 5, 1, 1, 3, 8, 1, 1, 1, 1, 1, 1, 3, 1, 10, ...
#> $ Epith.c.size    <ord> 2, 7, 2, 3, 2, 7, 2, 2, 2, 2, 1, 2, 2, 2, 7, 6...
#> $ Bare.nuclei     <fct> 1, 10, 2, 4, 1, 10, 10, 1, 1, 1, 1, 3, 3, 9...
#> $ Bl.cromatin     <fct> 3, 3, 3, 3, 9, 3, 3, 1, 2, 3, 2, 4, 3, 5, 4...
#> $ Normal.nucleoli <fct> 1, 2, 1, 7, 1, 1, 1, 1, 1, 4, 1, 5, 3...
```

```
#> $ Mitoses      <fct> 1, 1, 1, 1, 1, 1, 1, 1, 5, 1, 1, 1, 1, 1, 4, 1...
#> $ Class        <fct> benign, benign, benign, benign, benign, malign...
```

```
tibble::as_tibble(BreastCancer)
#> # A tibble: 699 x 11
#>   Id    Cl.thickness Cell.size Cell.shape Marg.adhesion Epith.c.size
#>   <chr> <ord>       <ord>     <ord>      <ord>           <ord>
#> 1 1000~ 5          1         1         1          2
#> 2 1002~ 5          4         4         5          7
#> 3 1015~ 3          1         1         1          2
#> 4 1016~ 6          8         8         1          3
#> 5 1017~ 4          1         1         3          2
#> 6 1017~ 8          10        10        8          7
#> # ... with 693 more rows, and 5 more variables: Bare.nuclei <fct>,
#> #   Bl.cromatin <fct>, Normal.nucleoli <fct>, Mitoses <fct>, Class <fct>
```

```
# Split out validation dataset
# create a list of 80% of the rows in the original dataset we can use for training
set.seed(7)
validationIndex <- createDataPartition(BreastCancer$Class,
                                         p=0.80,
                                         list=FALSE)

# select 20% of the data for validation
validation <- BreastCancer[-validationIndex,]
# use the remaining 80% of data to training and testing the models
dataset <- BreastCancer[validationIndex,]
```

```
# dimensions of dataset
dim(validation)
#> [1] 139 11
dim(dataset)
#> [1] 560 11
```

```
# peek
head(dataset, n=20)
#>   Id    Cl.thickness Cell.size Cell.shape Marg.adhesion Epith.c.size
#> 1 1000025          5         1         1          1          2
#> 2 1002945          5         4         4          5          7
#> 3 1015425          3         1         1          1          2
#> 5 1017023          4         1         1          3          2
#> 6 1017122          8         10        10          8          7
#> 7 1018099          1         1         1          1          2
#> 8 1018561          2         1         2          1          2
#> 9 1033078          2         1         1          1          2
#> 10 1033078         4         2         1          1          2
#> 11 1035283         1         1         1          1          1
#> 13 1041801         5         3         3          3          2
#> 14 1043999         1         1         1          1          2
#> 15 1044572         8         7         5          10          7
#> 16 1047630         7         4         6          4          6
#> 18 1049815         4         1         1          1          2
#> 19 1050670         10        7         7          6          4
#> 21 1054590         7         3         2          10          5
#> 22 1054593         10        5         5          3          6
```

```
#> 23 1056784      3      1      1      1      2
#> 24 1057013      8      4      5      1      2
#>   Bare.nuclei Bl.cromatin Normal.nucleoli Mitoses   Class
#> 1      1          3          1          1      benign
#> 2      10         3          2          1      benign
#> 3      2          3          1          1      benign
#> 5      1          3          1          1      benign
#> 6      10         9          7          1 malignant
#> 7      10         3          1          1      benign
#> 8      1          3          1          1      benign
#> 9      1          1          1          5      benign
#> 10     1          2          1          1      benign
#> 11     1          3          1          1      benign
#> 13     3          4          4          1 malignant
#> 14     3          3          1          1      benign
#> 15     9          5          5          4 malignant
#> 16     1          4          3          1 malignant
#> 18     1          3          1          1      benign
#> 19     10         4          1          2 malignant
#> 21     10         5          4          4 malignant
#> 22     7          7          10         1 malignant
#> 23     1          2          1          1      benign
#> 24     <NA>        7          3          1 malignant
```

```
library(skimr)
#>
#> Attaching package: 'skimr'
#> The following object is masked from 'package:stats':
#>
#>   filter
skim(dataset)
#> Skim summary statistics
#> n obs: 560
#> n variables: 11
#>
#> -- Variable type:character -----
#>   variable missing complete   n min max empty n_unique
#>     Id      0      560 560    5   8     0      523
#>
#> -- Variable type:factor -----
#>   variable missing complete   n n_unique
#>     Bare.nuclei    12      548 560    10
#>     Bl.cromatin    0      560 560    10
#>     Cell.shape     0      560 560    10
#>     Cell.size      0      560 560    10
#>     Cl.thickness   0      560 560    10
#>     Class          0      560 560     2
#>     Epith.c.size   0      560 560    10
#>     Marg.adhesion  0      560 560    10
#>     Mitoses        0      560 560     9
#>     Normal.nucleoli 0      560 560    10
#>                               top_counts ordered
#> 1: 327, 10: 102, 2: 28, 5: 24 FALSE
#> 3: 136, 2: 134, 1: 118, 7: 64 FALSE
```

```

#>   1: 281, 10: 48, 3: 47, 2: 46    TRUE
#>   1: 307, 10: 55, 3: 43, 2: 36    TRUE
#>   1: 113, 3: 94, 5: 94, 4: 62    TRUE
#>     ben: 367, mal: 193, NA: 0    FALSE
#>   2: 304, 3: 60, 4: 40, 1: 38    TRUE
#>   1: 331, 2: 44, 10: 44, 3: 42    TRUE
#>   1: 466, 3: 29, 2: 23, 4: 10    FALSE
#>   1: 347, 10: 50, 3: 36, 2: 29    FALSE

# types
sapply(dataset, class)
#> $Id
#> [1] "character"
#>
#> $Cl.thickness
#> [1] "ordered" "factor"
#>
#> $Cell.size
#> [1] "ordered" "factor"
#>
#> $Cell.shape
#> [1] "ordered" "factor"
#>
#> $Marg.adhesion
#> [1] "ordered" "factor"
#>
#> $Epith.c.size
#> [1] "ordered" "factor"
#>
#> $Bare.nuclei
#> [1] "factor"
#>
#> $Bl.cromatin
#> [1] "factor"
#>
#> $Normal.nucleoli
#> [1] "factor"
#>
#> $Mitoses
#> [1] "factor"
#>
#> $Class
#> [1] "factor"

```

We can see that besides the Id, the attributes are factors. This makes sense. I think for modeling it may be more useful to work with the data as numbers than factors. Factors might make things easier for decision tree algorithms (or not). Given that there is an ordinal relationship between the levels we can expose that structure to other algorithms better if we work directly with the integer numbers.

5.5 clean up

```

# Remove redundant variable Id
dataset <- dataset[,-1]

# convert input values to numeric
for(i in 1:9) {
    dataset[,i] <- as.numeric(as.character(dataset[,i]))
}

# summary
summary(dataset)
#>   Cl.thickness      Cell.size      Cell.shape      Marg.adhesion
#> Min. : 1.00      Min. : 1.00      Min. : 1.00      Min. : 1.00
#> 1st Qu.: 2.00    1st Qu.: 1.00    1st Qu.: 1.00    1st Qu.: 1.00
#> Median : 4.00    Median : 1.00    Median : 1.00    Median : 1.00
#> Mean   : 4.44    Mean   : 3.14    Mean   : 3.21    Mean   : 2.79
#> 3rd Qu.: 6.00    3rd Qu.: 5.00    3rd Qu.: 5.00    3rd Qu.: 4.00
#> Max.   :10.00    Max.   :10.00    Max.   :10.00    Max.   :10.00
#>
#>   Epith.c.size      Bare.nuclei     Bl.cromatin     Normal.nucleoli
#> Min. : 1.00      Min. : 1.00      Min. : 1.00      Min. : 1.00
#> 1st Qu.: 2.00    1st Qu.: 1.00    1st Qu.: 2.00    1st Qu.: 1.00
#> Median : 2.00    Median : 1.00    Median : 3.00    Median : 1.00
#> Mean   : 3.23    Mean   : 3.48    Mean   : 3.45    Mean   : 2.94
#> 3rd Qu.: 4.00    3rd Qu.: 5.25    3rd Qu.: 5.00    3rd Qu.: 4.00
#> Max.   :10.00    Max.   :10.00    Max.   :10.00    Max.   :10.00
#> NA's   :12
#>   Mitoses          Class
#> Min.   : 1.00    benign  :367
#> 1st Qu.: 1.00    malignant:193
#> Median : 1.00
#> Mean   : 1.59
#> 3rd Qu.: 1.00
#> Max.   :10.00
#>

skim(dataset)
#> Skim summary statistics
#> n obs: 560
#> n variables: 10
#>
#> -- Variable type:factor --
#>   variable missing complete   n n_unique          top_counts ordered
#>     Class       0      560 560           2 ben: 367, mal: 193, NA: 0 FALSE
#>
#> -- Variable type:numeric --
#>   variable missing complete   n mean   sd p0 p25 p50 p75 p100
#>     Bare.nuclei   12     548 560 3.48 3.62  1   1   1 5.25  10
#>     Bl.cromatin   0      560 560 3.45 2.43  1   2   3 5    10
#>     Cell.shape    0      560 560 3.21 2.97  1   1   1 5    10
#>     Cell.size     0      560 560 3.14 3.07  1   1   1 5    10
#>     Cl.thickness  0      560 560 4.44 2.83  1   2   4 6    10
#>     Epith.c.size 0      560 560 3.23 2.22  1   2   2 4    10

```

```
#>   Marg.adhesion      0      560 560 2.79 2.85  1   1   1 4      10
#>   Mitoses            0      560 560 1.59 1.7   1   1   1 1      10
#>   Normal.nucleoli   0      560 560 2.94 3.08  1   1   1 4      10
#>   hist
#>
#>
#>
#>
#>
#>
#>
```

we can see we have 13 NA values for the Bare.nuclei attribute. This suggests we may need to remove the records (or impute values) with NA values for some analysis and modeling techniques.

5.6 Analyze the class variable

```
# class distribution
cbind(freq = table(dataset$Class),
      percentage = prop.table(table(dataset$Class))*100)
#>   freq percentage
#> benign    367      65.5
#> malignant 193      34.5
```

There is indeed a 65% to 35% split for benign-malignant in the class values which is imbalanced, but not so much that we need to be thinking about rebalancing the dataset, at least not yet.

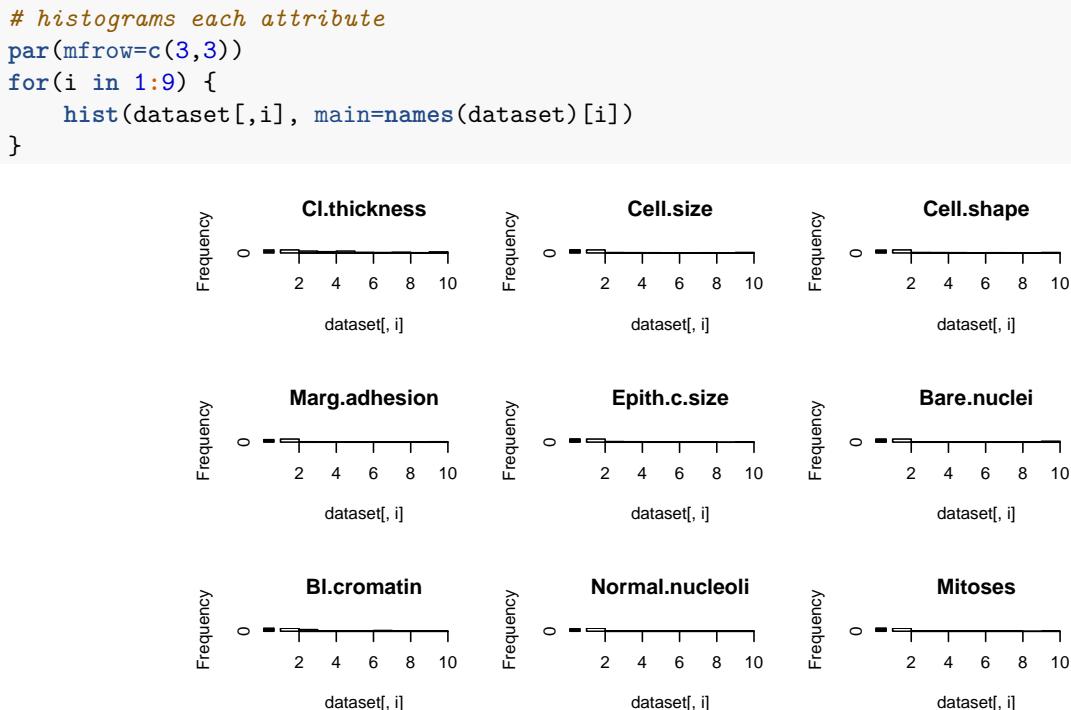
5.6.1 remove NAs

```
# summarize correlations between input variables
complete_cases <- complete.cases(dataset)
cor(dataset[complete_cases,1:9])
#>   Cl.thickness Cell.size Cell.shape Marg.adhesion
#> Cl.thickness     1.000    0.665    0.667    0.486
#> Cell.size        0.665    1.000    0.904    0.722
#> Cell.shape       0.667    0.904    1.000    0.694
#> Marg.adhesion   0.486    0.722    0.694    1.000
#> Epith.c.size    0.543    0.773    0.739    0.643
#> Bare.nuclei     0.598    0.700    0.721    0.669
#> Bl.cromatin     0.565    0.752    0.739    0.692
#> Normal.nucleoli 0.570    0.737    0.741    0.644
#> Mitoses          0.347    0.453    0.432    0.419
#>   Epith.c.size Bare.nuclei Bl.cromatin Normal.nucleoli
#> Cl.thickness     0.543    0.598    0.565    0.570
#> Cell.size         0.773    0.700    0.752    0.737
#> Cell.shape        0.739    0.721    0.739    0.741
#> Marg.adhesion   0.643    0.669    0.692    0.644
#> Epith.c.size     1.000    0.614    0.628    0.642
#> Bare.nuclei      0.614    1.000    0.685    0.605
```

```
#> Bl.cromatin      0.628      0.685      1.000      0.692
#> Normal.nucleoli 0.642      0.605      0.692      1.000
#> Mitoses          0.485      0.351      0.356      0.432
#>                   Mitoses
#> Cl.thickness     0.347
#> Cell.size         0.453
#> Cell.shape        0.432
#> Marg.adhesion    0.419
#> Epith.c.size     0.485
#> Bare.nuclei      0.351
#> Bl.cromatin      0.356
#> Normal.nucleoli  0.432
#> Mitoses          1.000
```

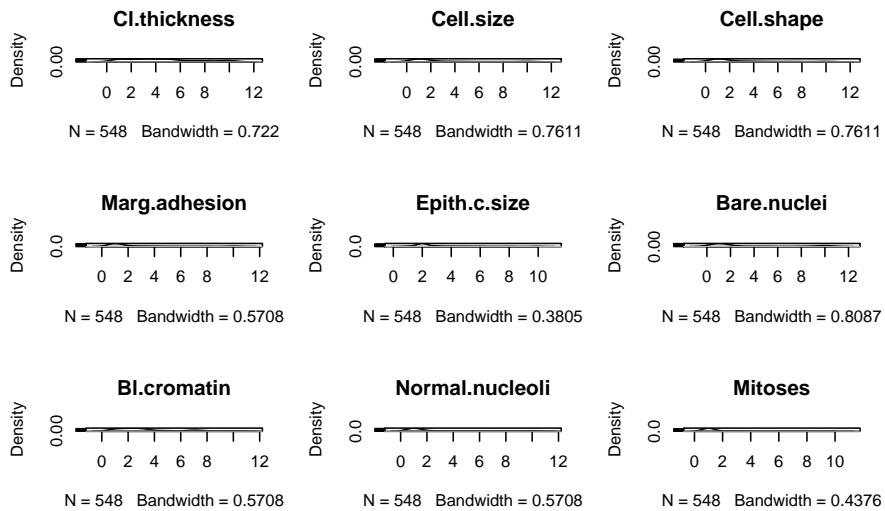
We can see some modest to high correlation between some of the attributes. For example between cell shape and cell size at 0.90 correlation.

5.7 Unimodal visualization



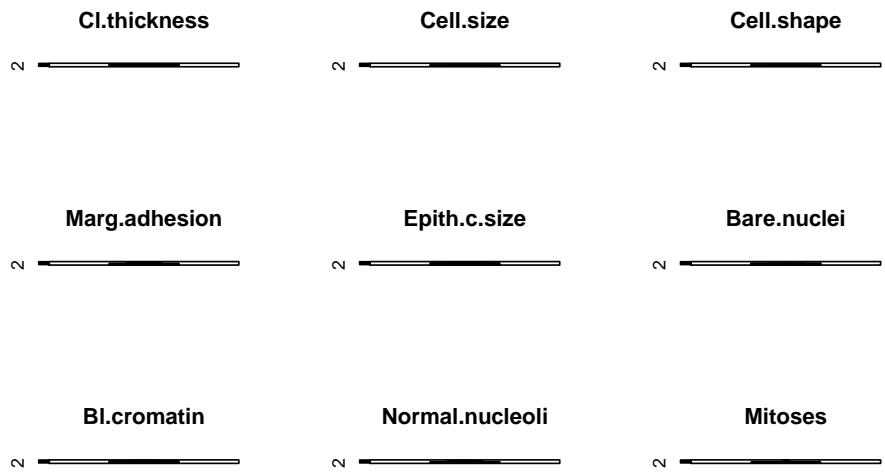
We can see that almost all of the distributions have an exponential or bimodal shape to them.

```
# density plot for each attribute
par(mfrow=c(3,3))
complete_cases <- complete.cases(dataset)
for(i in 1:9) {
  plot(density(dataset[complete_cases,i]), main=names(dataset)[i])
}
```



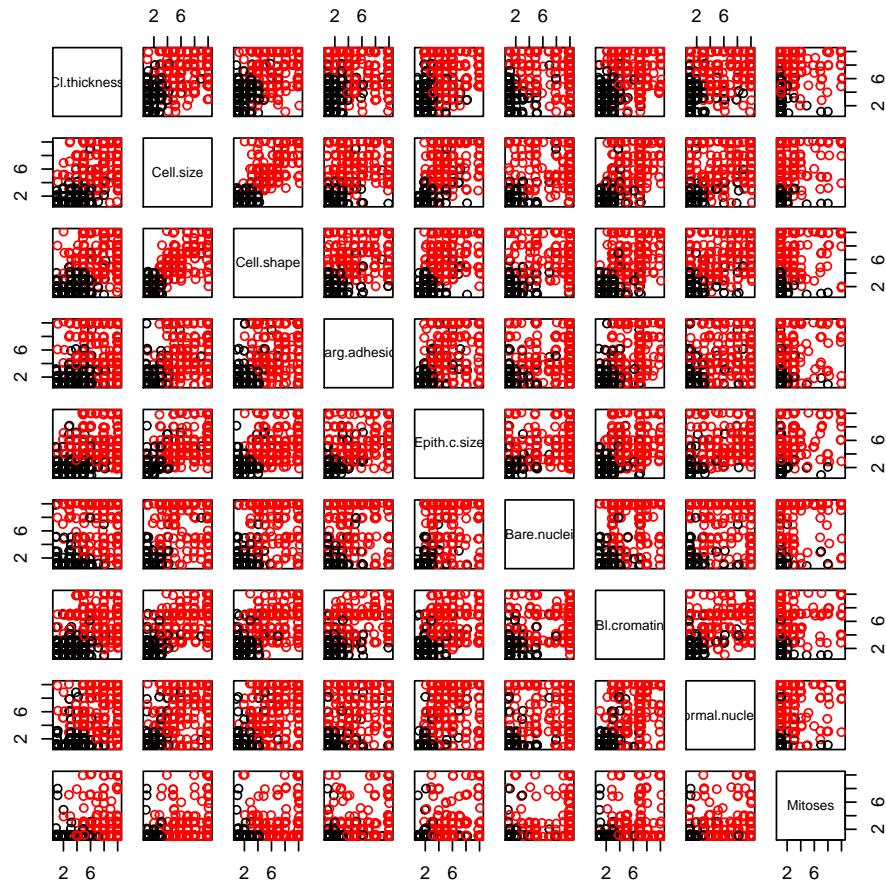
These plots add more support to our initial ideas. We can see bimodal distributions (two bumps) and exponential-looking distributions.

```
# boxplots for each attribute
par(mfrow=c(3,3))
for(i in 1:9) {
  boxplot(dataset[,i], main=names(dataset)[i])
}
```



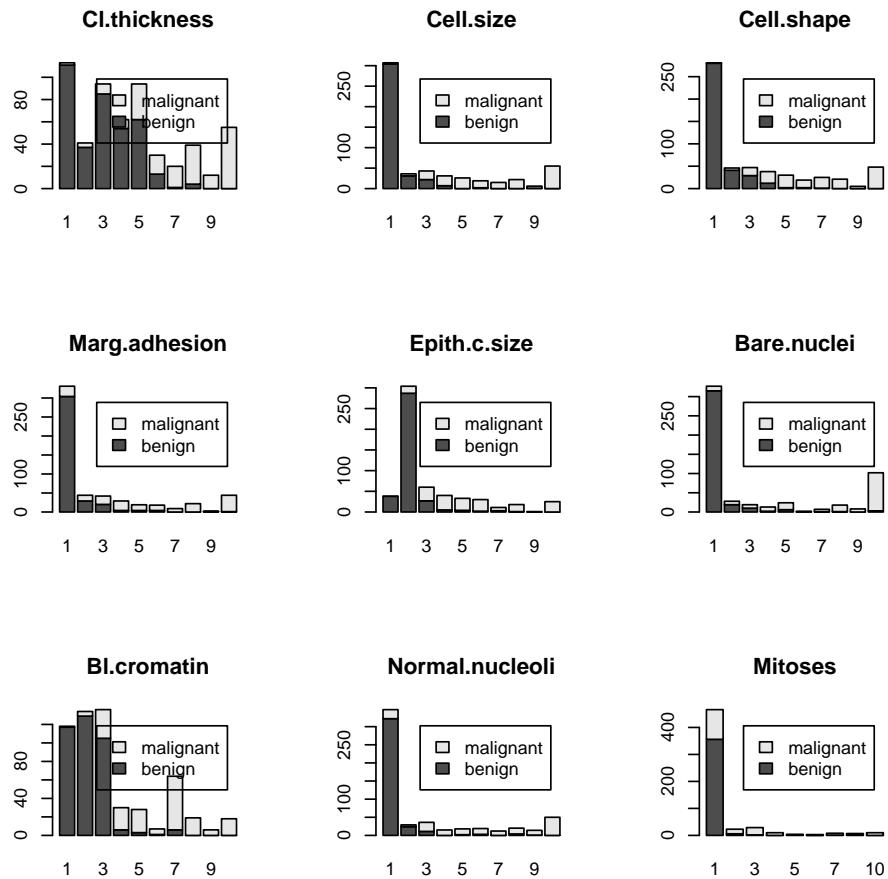
5.8 Multimodal visualization

```
# scatter plot matrix
jittered_x <- sapply(dataset[,1:9], jitter)
pairs(jittered_x, names(dataset[,1:9]), col=dataset$Class)
```



We can see that the black (benign) a part to be clustered around the bottom-right corner (smaller values) and red (malignant) are all over the place.

```
# bar plots of each variable by class
par(mfrow=c(3,3))
for(i in 1:9) {
  barplot(table(dataset$Class,dataset[,i]), main=names(dataset)[i],
         legend.text=unique(dataset$Class))
}
```



5.9 Algorithms Evaluation

- Linear Algorithms: Logistic Regression (LG), Linear Discriminate Analysis (LDA) and Regularized Logistic Regression (GLMNET).
- Nonlinear Algorithms: k-Nearest Neighbors (KNN), Classification and Regression Trees (CART), Naive Bayes (NB) and Support Vector Machines with Radial Basis Functions (SVM).

For simplicity, we will use Accuracy and Kappa metrics. Given that it is a medical test, we could have gone with the Area Under ROC Curve (AUC) and looked at the sensitivity and specificity to select the best algorithms.

```
# 10-fold cross-validation with 3 repeats
trainControl <- trainControl(method = "repeatedcv", number=10, repeats=3)
metric <- "Accuracy"

tic()
# LG
set.seed(7)
fit.glm <- train(Class~., data=dataset, method="glm", metric=metric,
                   trControl=trainControl, na.action=na.omit)

# LDA
set.seed(7)
fit lda <- train(Class~., data=dataset, method="lda", metric=metric,
                   trControl=trainControl, na.action=na.omit)

# GLMNET
```

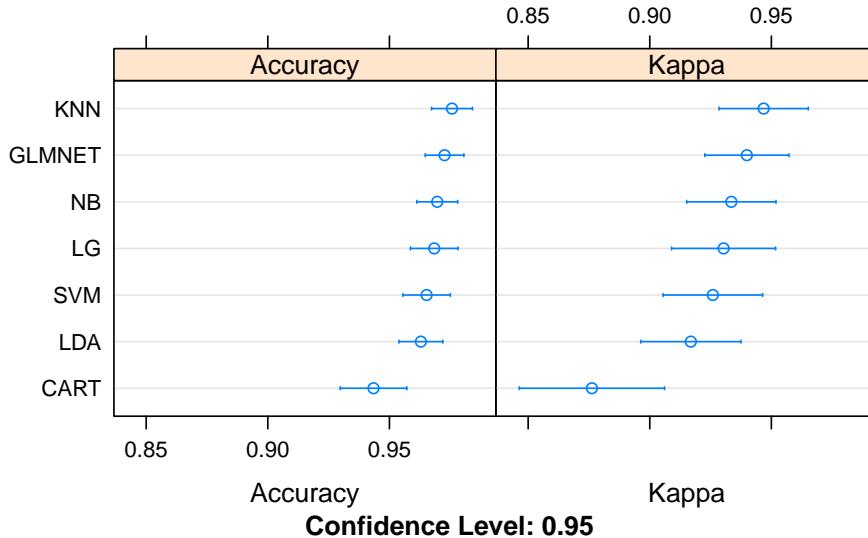
```

set.seed(7)
fit.glmnet <- train(Class~., data=dataset, method="glmnet", metric=metric,
                     trControl=trainControl, na.action=na.omit)
# KNN
set.seed(7)
fit.knn <- train(Class~., data=dataset, method="knn", metric=metric,
                   trControl=trainControl, na.action=na.omit)
# CART
set.seed(7)
fit.cart <- train(Class~., data=dataset, method="rpart", metric=metric,
                    trControl=trainControl, na.action=na.omit)
# Naive Bayes
set.seed(7)
fit.nb <- train(Class~., data=dataset, method="nb", metric=metric,
                  trControl=trainControl, na.action=na.omit)
# SVM
set.seed(7)
fit.svm <- train(Class~., data=dataset, method="svmRadial", metric=metric,
                   trControl=trainControl, na.action=na.omit)

# Compare algorithms
results <- resamples(list(LG      = fit.glm,
                           LDA     = fit.lda,
                           GLMNET = fit.glmnet,
                           KNN    = fit.knn,
                           CART   = fit.cart,
                           NB     = fit.nb,
                           SVM    = fit.svm))
toc()
#> 13.846 sec elapsed
summary(results)
#>
#> Call:
#> summary.resamples(object = results)
#>
#> Models: LG, LDA, GLMNET, KNN, CART, NB, SVM
#> Number of resamples: 30
#>
#> Accuracy
#>          Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> LG      0.909  0.945  0.964  0.968  0.995    1    0
#> LDA     0.907  0.945  0.964  0.963  0.982    1    0
#> GLMNET 0.927  0.964  0.964  0.973  0.995    1    0
#> KNN    0.927  0.964  0.982  0.976  1.000    1    0
#> CART   0.833  0.927  0.945  0.943  0.964    1    0
#> NB     0.927  0.963  0.981  0.970  0.982    1    0
#> SVM    0.907  0.945  0.964  0.965  0.982    1    0
#>
#> Kappa
#>          Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> LG      0.806  0.880  0.921  0.930  0.990    1    0
#> LDA     0.786  0.880  0.918  0.917  0.959    1    0
#> GLMNET 0.843  0.918  0.922  0.940  0.990    1    0

```

```
#> KNN  0.843  0.920  0.959 0.947  1.000    1    0
#> CART 0.630  0.840  0.879 0.876  0.920    1    0
#> NB   0.835  0.918  0.959 0.934  0.960    1    0
#> SVM  0.804  0.883  0.922 0.926  0.960    1    0
dotplot(results)
```



We can see good accuracy across the board. All algorithms have a mean accuracy above 90%, well above the baseline of 65% if we just predicted benign. The problem is learnable. We can see that KNN (97.08%) and logistic regression (NB was 96.2% and GLMNET was 96.4%) had the highest accuracy on the problem.

5.10 Data transform

We know we have some skewed distributions. There are transform methods that we can use to adjust and normalize these distributions. A favorite for positive input attributes (which we have in this case) is the Box-Cox transform.

```
# 10-fold cross-validation with 3 repeats
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
metric <- "Accuracy"

# LG
set.seed(7)
fit.glm <- train(Class~, data=dataset, method="glm", metric=metric,
                   preProc=c("BoxCox"), trControl=trainControl, na.action=na.omit)

# LDA
set.seed(7)
fit.lda <- train(Class~, data=dataset, method="lda", metric=metric,
                   preProc=c("BoxCox"), trControl=trainControl, na.action=na.omit)

# GLMNET
set.seed(7)
fit.glmnet <- train(Class~, data=dataset, method="glmnet", metric=metric,
                      preProc=c("BoxCox"), trControl=trainControl,
                      na.action=na.omit)

# KNN
```

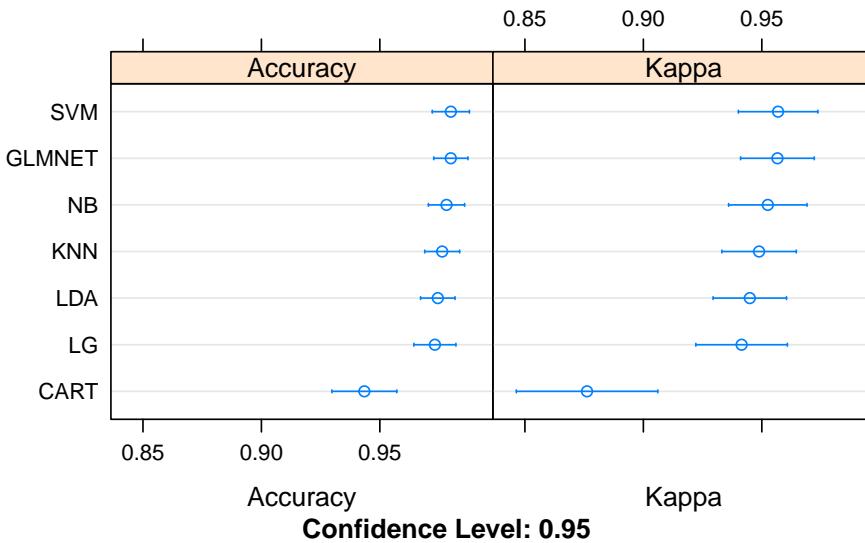
```
set.seed(7)
fit.knn <- train(Class~., data=dataset, method="knn", metric=metric,
                  preProc=c("BoxCox"), trControl=trainControl, na.action=na.omit)
# CART
set.seed(7)
fit.cart <- train(Class~., data=dataset, method="rpart", metric=metric,
                   preProc=c("BoxCox"), trControl=trainControl,
                   na.action=na.omit)
# Naive Bayes
set.seed(7)
fit.nb <- train(Class~., data=dataset, method="nb", metric=metric,
                  preProc=c("BoxCox"), trControl=trainControl, na.action=na.omit)
#> Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
#> observation 1
#> Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
#> observation 24
#> Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
#> observation 28
#> Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
#> observation 20
#> Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
#> observation 11
#> Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
#> observation 18
#> Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
#> observation 54
#> Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
#> observation 3
#> Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
#> observation 23
#> Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
#> observation 21
#> Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
#> observation 27
#> Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
#> observation 53
#> Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
#> observation 12
#> Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
#> observation 9
#> Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
#> observation 2
#> Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
#> observation 17
#> Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
#> observation 9
#> Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
#> observation 55
#> Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
#> observation 23
#> Warning in FUN(X[[i]], ...): Numerical 0 probability for all classes with
#> observation 32
# SVM
```

```

set.seed(7)
fit.svm <- train(Class~., data=dataset, method="svmRadial", metric=metric,
                  preProc=c("BoxCox"), trControl=trainControl, na.action=na.omit)

# Compare algorithms
transformResults <- resamples(list(LG      = fit.glm,
                                    LDA     = fit.lda,
                                    GLMNET = fit.glmnet,
                                    KNN    = fit.knn,
                                    CART   = fit.cart,
                                    NB     = fit.nb,
                                    SVM    = fit.svm))
summary(transformResults)
#>
#> Call:
#> summary.resamples(object = transformResults)
#>
#> Models: LG, LDA, GLMNET, KNN, CART, NB, SVM
#> Number of resamples: 30
#>
#> Accuracy
#>           Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> LG      0.909 0.963 0.982 0.973 0.996 1 0
#> LDA     0.927 0.964 0.981 0.974 0.982 1 0
#> GLMNET 0.944 0.964 0.982 0.980 1.000 1 0
#> KNN    0.909 0.964 0.981 0.976 0.982 1 0
#> CART   0.833 0.927 0.945 0.943 0.964 1 0
#> NB     0.927 0.964 0.982 0.978 1.000 1 0
#> SVM    0.927 0.964 0.982 0.980 1.000 1 0
#>
#> Kappa
#>           Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> LG      0.806 0.919 0.959 0.941 0.990 1 0
#> LDA     0.847 0.921 0.959 0.945 0.961 1 0
#> GLMNET 0.878 0.922 0.960 0.957 1.000 1 0
#> KNN    0.806 0.922 0.959 0.949 0.961 1 0
#> CART   0.630 0.840 0.879 0.876 0.920 1 0
#> NB     0.843 0.922 0.960 0.953 1.000 1 0
#> SVM    0.847 0.922 0.960 0.957 1.000 1 0
dotplot(transformResults)

```



We can see that the accuracy of the previous best algorithm KNN was elevated to 97.14%. We have a new ranking, showing SVM with the most accurate mean accuracy at 97.20%.

5.11 Tuning SVM

```
# 10-fold cross-validation with 3 repeats
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
metric <- "Accuracy"
set.seed(7)

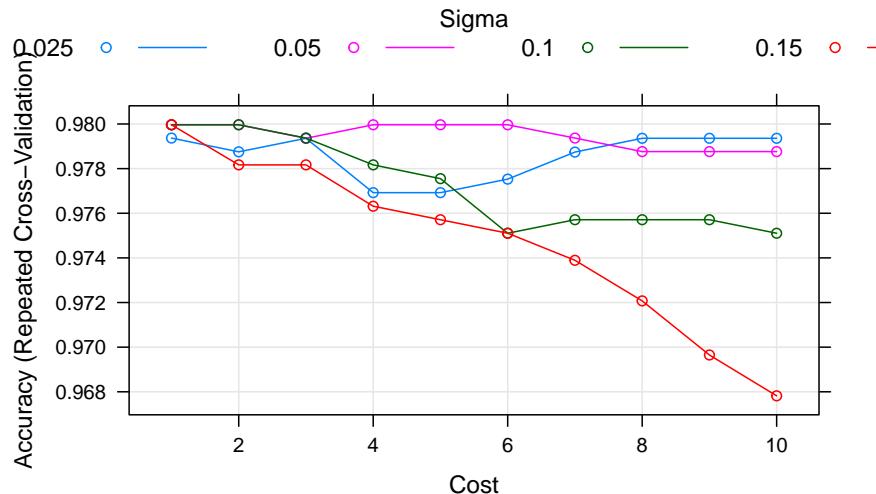
grid <- expand.grid(.sigma = c(0.025, 0.05, 0.1, 0.15),
                     .C = seq(1, 10, by=1))

fit.svm <- train(Class~, data=dataset, method="svmRadial", metric=metric,
                   tuneGrid=grid,
                   preProc=c("BoxCox"), trControl=trainControl,
                   na.action=na.omit)
print(fit.svm)
#> Support Vector Machines with Radial Basis Function Kernel
#>
#> 560 samples
#> 9 predictor
#> 2 classes: 'benign', 'malignant'
#>
#> Pre-processing: Box-Cox transformation (9)
#> Resampling: Cross-Validated (10 fold, repeated 3 times)
#> Summary of sample sizes: 493, 492, 493, 493, 493, 494, ...
#> Resampling results across tuning parameters:
#>
#>   sigma  C  Accuracy  Kappa
#>   0.025  1  0.979    0.956
#>   0.025  2  0.979    0.954
#>   0.025  3  0.979    0.956
#>   0.025  4  0.977    0.950
```

```

#> 0.025 5 0.977 0.950
#> 0.025 6 0.978 0.951
#> 0.025 7 0.979 0.954
#> 0.025 8 0.979 0.956
#> 0.025 9 0.979 0.956
#> 0.025 10 0.979 0.956
#> 0.050 1 0.980 0.957
#> 0.050 2 0.980 0.957
#> 0.050 3 0.979 0.956
#> 0.050 4 0.980 0.957
#> 0.050 5 0.980 0.957
#> 0.050 6 0.980 0.957
#> 0.050 7 0.979 0.956
#> 0.050 8 0.979 0.954
#> 0.050 9 0.979 0.954
#> 0.050 10 0.979 0.954
#> 0.100 1 0.980 0.957
#> 0.100 2 0.980 0.957
#> 0.100 3 0.979 0.956
#> 0.100 4 0.978 0.953
#> 0.100 5 0.978 0.952
#> 0.100 6 0.975 0.946
#> 0.100 7 0.976 0.948
#> 0.100 8 0.976 0.948
#> 0.100 9 0.976 0.948
#> 0.100 10 0.975 0.946
#> 0.150 1 0.980 0.957
#> 0.150 2 0.978 0.953
#> 0.150 3 0.978 0.953
#> 0.150 4 0.976 0.949
#> 0.150 5 0.976 0.948
#> 0.150 6 0.975 0.946
#> 0.150 7 0.974 0.944
#> 0.150 8 0.972 0.939
#> 0.150 9 0.970 0.934
#> 0.150 10 0.968 0.930
#>
## Accuracy was used to select the optimal model using the largest value.
## The final values used for the model were sigma = 0.15 and C = 1.
plot(fit.svm)

```



We can see that we have made very little difference to the results. The most accurate model had a score of 97.31% (the same as our previously rounded score of 97.20%) using a sigma = 0.1 and C = 1. We could tune further, but I don't expect a payoff.

5.12 Tuning KNN

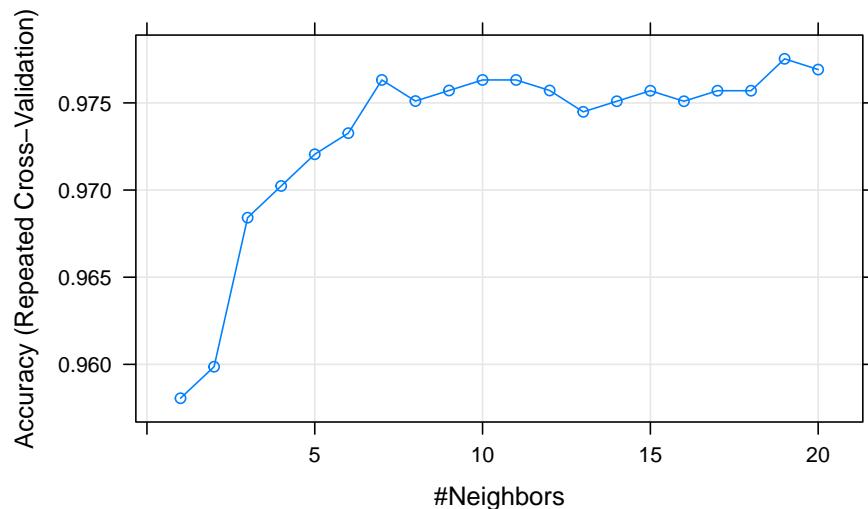
```
# 10-fold cross-validation with 3 repeats
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
metric <- "Accuracy"
set.seed(7)

grid <- expand.grid(k = seq(1,20, by=1))
fit.knn <- train(Class~., data=dataset, method="knn", metric=metric,
                  tuneGrid=grid,
                  preProc=c("BoxCox"), trControl=trainControl,
                  na.action=na.omit)
print(fit.knn)
#> k-Nearest Neighbors
#>
#> #> 560 samples
#> #> 9 predictor
#> #> 2 classes: 'benign', 'malignant'
#>
#> #> Pre-processing: Box-Cox transformation (9)
#> #> Resampling: Cross-Validated (10 fold, repeated 3 times)
#> #> Summary of sample sizes: 493, 492, 493, 493, 493, 494, ...
#> #> Resampling results across tuning parameters:
#>
#> #>   k    Accuracy   Kappa
#> #>   1    0.958     0.908
#> #>   2    0.960     0.912
#> #>   3    0.968     0.931
#> #>   4    0.970     0.935
#> #>   5    0.972     0.939
#> #>   6    0.973     0.942
#> #>   7    0.976     0.949
```

```

#>    8  0.975    0.946
#>    9  0.976    0.947
#>   10  0.976    0.949
#>   11  0.976    0.949
#>   12  0.976    0.947
#>   13  0.974    0.945
#>   14  0.975    0.946
#>   15  0.976    0.947
#>   16  0.975    0.946
#>   17  0.976    0.947
#>   18  0.976    0.947
#>   19  0.978    0.951
#>   20  0.977    0.950
#>
#> Accuracy was used to select the optimal model using the largest value.
#> The final value used for the model was k = 19.
plot(fit.knn)

```



We can see again that tuning has made little difference, settling on a value of $k = 7$ with an accuracy of 97.19%. This is higher than the previous 97.14%, but very similar (or perhaps identical!) to the result achieved by the tuned SVM.

5.13 Ensemble

```

# 10-fold cross-validation with 3 repeats
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
metric <- "Accuracy"

# Bagged CART
set.seed(7)
fit.treebag <- train(Class~, data=dataset, method="treebag", metric=metric,
                      trControl=trainControl, na.action=na.omit)

# Random Forest
set.seed(7)

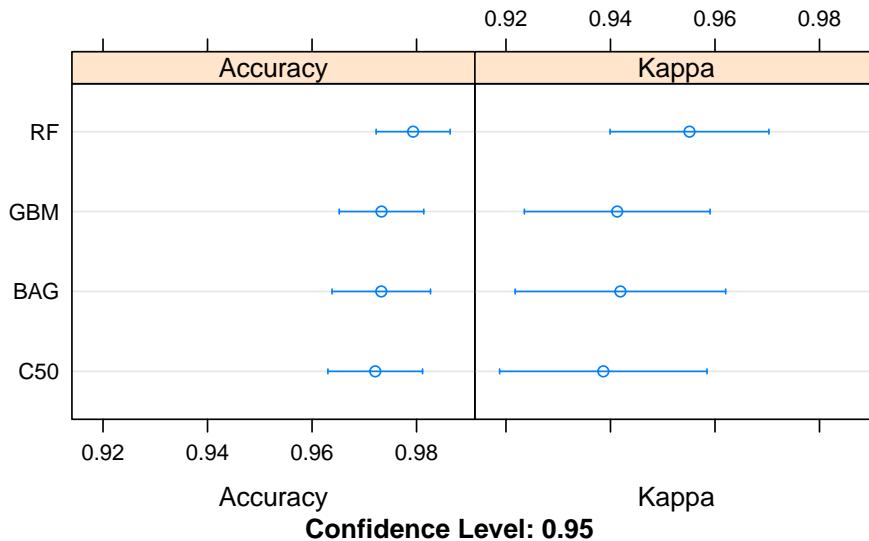
```

```
fit.rf <- train(Class~, data=dataset, method="rf", metric=metric,
                 preProc=c("BoxCox"), trControl=trainControl, na.action=na.omit)

# Stochastic Gradient Boosting
set.seed(7)
fit.gbm <- train(Class~, data=dataset, method="gbm", metric=metric,
                  preProc=c("BoxCox"), trControl=trainControl, verbose=FALSE, na.action=na.omit)

# C5.0
set.seed(7)
fit.c50 <- train(Class~, data=dataset, method="C5.0", metric=metric,
                  preProc=c("BoxCox"), trControl=trainControl, na.action=na.omit)

# Compare results
ensembleResults <- resamples(list(BAG = fit.treebag,
                                    RF   = fit.rf,
                                    GBM  = fit.gbm,
                                    C50  = fit.c50))
summary(ensembleResults)
#>
#> Call:
#> summary.resamples(object = ensembleResults)
#>
#> Models: BAG, RF, GBM, C50
#> Number of resamples: 30
#>
#> Accuracy
#>      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> BAG 0.907 0.963 0.982 0.973 0.982 1 0
#> RF  0.926 0.981 0.982 0.979 0.995 1 0
#> GBM 0.929 0.963 0.981 0.973 0.995 1 0
#> C50 0.907 0.964 0.982 0.972 0.982 1 0
#>
#> Kappa
#>      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> BAG 0.804 0.920 0.959 0.942 0.96 1 0
#> RF  0.841 0.959 0.960 0.955 0.99 1 0
#> GBM 0.844 0.919 0.959 0.941 0.99 1 0
#> C50 0.795 0.918 0.959 0.939 0.96 1 0
dotplot(ensembleResults)
```



We see that Random Forest was the most accurate with a score of 97.26%. Very similar to our tuned models above. We could spend time tuning the parameters of Random Forest (e.g. increasing the number of trees) and the other ensemble methods, but I don't expect to see better accuracy scores other than random statistical fluctuations.

5.14 Finalize model

We now need to finalize the model, which really means choose which model we would like to use. For simplicity I would probably select the KNN method, at the expense of the memory required to store the training dataset. SVM would be a good choice to trade-off space and time complexity. I probably would not select the Random Forest algorithm given the complexity of the model. It seems overkill for this dataset, lots of trees with little benefit in Accuracy.

Let's go with the KNN algorithm. This is really simple, as we do not need to store a model. We do need to capture the parameters of the Box-Cox transform though. And we also need to prepare the data by removing the unused Id attribute and converting all of the inputs to numeric format.

The implementation of KNN (`knn3()`) belongs to the caret package and does not support missing values. We will have to remove the rows with missing values from the training dataset as well as the validation dataset. The code below shows the preparation of the pre-processing parameters using the training dataset.

```
# prepare parameters for data transform
set.seed(7)

datasetNoMissing <- dataset[complete.cases(dataset),]
x <- datasetNoMissing[,1:9]

# transform
preprocessParams <- preprocess(x, method=c("BoxCox"))
x <- predict(preprocessParams, x)
```

5.15 Prepare the validation set

Next we need to prepare the validation dataset for making a prediction. We must:

1. Remove the Id attribute.
2. Remove those rows with missing data.
3. Convert all input attributes to numeric.
4. Apply the Box-Cox transform to the input attributes using parameters prepared on the training dataset.

```
# prepare the validation dataset
set.seed(7)

# remove id column
validation <- validation[,-1]

# remove missing values (not allowed in this implementation of knn)
validation <- validation[complete.cases(validation),]

# convert to numeric
for(i in 1:9) {
    validation[,i] <- as.numeric(as.character(validation[,i]))
}

# transform the validation dataset
validationX <- predict(preprocessParams, validation[,1:9])

# make predictions
set.seed(7)
# knn3Train(train, test, cl, k = 1, l = 0, prob = TRUE, use.all = TRUE)
# k: number of neighbours considered.
predictions <- knn3Train(x, validationX, datasetNoMissing$Class,
                         k = 9,
                         prob = FALSE)

# convert
confusionMatrix(as.factor(predictions), validation$Class)
#> Confusion Matrix and Statistics
#>
#>           Reference
#>           Prediction benign malignant
#>   benign            83         1
#>   malignant          4        47
#>
#>           Accuracy : 0.963
#>           95% CI : (0.916, 0.988)
#>   No Information Rate : 0.644
#>   P-Value [Acc > NIR] : <2e-16
#>
#>           Kappa : 0.92
#>
#>   # McNemar's Test P-Value : 0.371
#>
#>           Sensitivity : 0.954
#>           Specificity : 0.979
#>           Pos Pred Value : 0.988
#>           Neg Pred Value : 0.922
#>           Prevalence : 0.644
#>           Detection Rate : 0.615
#>           Detection Prevalence : 0.622
```

```
#>      Balanced Accuracy : 0.967
#>
#>      'Positive' Class : benign
#>
```

We can see that the accuracy of the final model on the validation dataset is 99.26%. This is optimistic because there is only 136 rows, but it does show that we have an accurate standalone model that we could use on other unclassified data.

Chapter 6

Classification algorithms comparison. outbreaks dataset. (*RF, GLMNET,* *KNN, PDA, LDA, NSC, C5, PLS*)

6.1 Introduction

Among the many nice R packages containing data collections is the outbreaks package. It contains datasets on epidemics and among them is data from the 2013 outbreak of influenza A H7N9 in China as analysed by Kucharski et al. (2014):

A. Kucharski, H. Mills, A. Pinsent, C. Fraser, M. Van Kerkhove, C. A. Donnelly, and S. Riley. 2014. Distinguishing between reservoir exposure and human-to-human transmission for emerging pathogens using case onset data. PLOS Currents Outbreaks. Mar 7, edition 1. doi: 10.1371/currents.outbreaks.e1473d9bfc99d080ca242139a06c455f.

A. Kucharski, H. Mills, A. Pinsent, C. Fraser, M. Van Kerkhove, C. A. Donnelly, and S. Riley. 2014. Data from: Distinguishing between reservoir exposure and human-to-human transmission for emerging pathogens using case onset data. Dryad Digital Repository. <http://dx.doi.org/10.5061/dryad.2g43n>.

6.1.1 Algorithms

We compare these classification algorithms:

1. Random Forest
2. GLM net
3. k-Nearest Neighbors
4. Penalized Discriminant Analysis
5. Stabilized Linear Discriminant Analysis
6. Nearest Shrunken Centroids
7. Single C5.0 Tree
8. Partial Least Squares

6.1.2 Workflow

1. Load dataset

2. Data wrangling

- Many dates as one variable: `gather`
- Convert group to factor: `factor`
- Change dates descriptions: `dplyr::mapvalues`
- Set several provinces as Other: `mapvalues`
- Convert gender to factor
- Convert province to factor

3. Features visualization

- Area density plot of Month vs Age by Province, by date group, by outcome, by gender
- Number of flu cases by gender, by province
- Age density plot of flu cases by outcome
- Days passed from outset by province, by age, by gender, by outcome

4. Feature engineering

- Generate new features
- Gender to numeric
- Provinces to binary classifiers
- Outcome to binary classifier
- Impute missing data: `mice`

5. Split dataset in training, validation and test sets

6. Feature importance

- with Decision Trees
- with Random Forest
- Wrangling dataset
- Plot Importance vs Variable

7. Impact on Datasets

- Density plot of training, validation and test datasets
- Features vs outcome by age, days onset to hospital, day onset to outcome

8. Train models on training validation dataset

- Numeric and visual Comparison of models
 - Accuracy
 - Kappa
- Compare predictions on training validation set
 - Create dataset for results
 - New variable for outcome

11. Predicting on unknown outcomes on training data

- Numeric and visual Comparison of models
 - Accuracy
 - Kappa
- Compare predictions on training validation set
 - Create dataset for results
 - New variable for outcome
- Calculate predicted outcome
- Save to CSV file
- Calculate recovery cases

- Summarize outcome
- Plot month vs log2-ratio of recovery vs death, by gender, by age, by date group

6.1.3 Can we predict flu outcome with Machine Learning in R?

I will be using their data as an example to show how to use Machine Learning algorithms for predicting disease outcome.

To do so, I selected and extracted features from the raw data, including age, days between onset and outcome, gender, whether the patients were hospitalised, etc. Missing values were imputed and different model algorithms were used to predict outcome (death or recovery). The prediction accuracy, sensitivity and specificity. The thus prepared dataset was divided into training and testing subsets. The test subset contained all cases with an unknown outcome. Before I applied the models to the test data, I further split the training data into validation subsets.

The tested modeling algorithms were similarly successful at predicting the outcomes of the validation data. To decide on final classifications, I compared predictions from all models and defined the outcome “Death” or “Recovery” as a function of all models, whereas classifications with a low prediction probability were flagged as “uncertain”. Accounting for this uncertainty led to a 100% correct classification of the validation test set.

The training cases with unknown outcome were then classified based on the same algorithms. From 57 unknown cases, 14 were classified as “Recovery”, 10 as “Death” and 33 as uncertain.

In a Part 2, I am looking at how extreme gradient boosting performs on this dataset.

Disclaimer: I am not an expert in Machine Learning. Everything I know, I taught myself during the last months. So, if you see any mistakes or have tips and tricks for improvement, please don't hesitate to let me know! Thanks. :-)

6.2 The data

The dataset contains case ID, date of onset, date of hospitalisation, date of outcome, gender, age, province and of course the outcome: Death or Recovery. I can already see that there are a couple of missing values in the data, which I will deal with later.

```
options(width = 1000)

if (!require("outbreaks")) install.packages("outbreaks")
#> Loading required package: outbreaks
library(outbreaks)

fluH7N9_china_2013_backup <- fluH7N9_china_2013

fluH7N9_china_2013$age[which(fluH7N9_china_2013$age == "?")] <- NA
fluH7N9_china_2013$case_ID <- paste("case",
                                      fluH7N9_china_2013$case_id,
                                      sep = "_")

head(fluH7N9_china_2013)
#>   case_id date_of_onset date_of_hospitalisation date_of_outcome outcome gender age province case_ID
#> 1       1 2013-02-19                      <NA> 2013-03-04    Death     m  87 Shanghai  case_1
#> 2       2 2013-02-27                     2013-03-03 2013-03-10    Death     m  27 Shanghai  case_2
#> 3       3 2013-03-09                     2013-03-19 2013-04-09    Death     f  35  Anhui  case_3
#> 4       4 2013-03-19                      <NA>      <NA>      <NA>     f  45 Jiangsu  case_4
```

```
#> 5      5  2013-03-19          2013-03-30  2013-05-15 Recover   f 48 Jiangsu case_5
#> 6      6  2013-03-21          2013-03-28  2013-04-26 Death    f 32 Jiangsu case_6
```

Before I start preparing the data for Machine Learning, I want to get an idea of the distribution of the data points and their different variables by plotting.

Most provinces have only a handful of cases, so I am combining them into the category “other” and keep only Jiangsu, Shanghai and Zhejiang and separate provinces.

```
# make tidy data, clean up and simplify
library(tidyr)

# put different type of dates in one variable (Date)
fluH7N9_china_2013_gather <- fluH7N9_china_2013 %>%
  gather(Group, Date, date_of_onset:date_of_outcome)

# convert Group to factor
fluH7N9_china_2013_gather$Group <- factor(fluH7N9_china_2013_gather$Group,
                                              levels = c("date_of_onset",
                                                        "date_of_hospitalisation",
                                                        "date_of_outcome"))

# change the dates description with plyr::mapvalues
library(plyr)
from <- c("date_of_onset", "date_of_hospitalisation", "date_of_outcome")
to <- c("Date of onset", "Date of hospitalisation", "Date of outcome")
fluH7N9_china_2013_gather$Group <- mapvalues(fluH7N9_china_2013_gather$Group,
                                                from = from, to = to)

# change additional provinces to Other
from <- c("Anhui", "Beijing", "Fujian", "Guangdong", "Hebei", "Henan",
          "Hunan", "Jiangxi", "Shandong", "Taiwan")
to <- rep("Other", 10)
fluH7N9_china_2013_gather$province <-
  mapvalues(fluH7N9_china_2013_gather$province,
            from = from, to = to)

# convert gender to factor
levels(fluH7N9_china_2013_gather$gender) <-
  c(levels(fluH7N9_china_2013_gather$gender), "unknown")

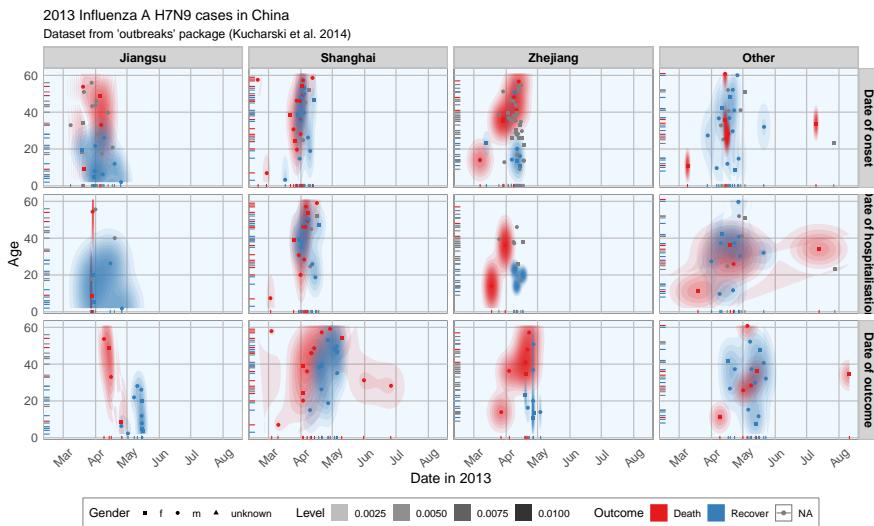
# replace NA in gender by "unknown"
is_na <- is.na(fluH7N9_china_2013_gather$gender)
fluH7N9_china_2013_gather$gender[is_na] <- "unknown"

# convert province to factor
fluH7N9_china_2013_gather$province <- factor(fluH7N9_china_2013_gather$province,
                                               levels = c("Jiangsu", "Shanghai",
                                                         "Zhejiang", "Other"))

library(ggplot2)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures   rlang
#>   c.quosures   rlang
```

```
#> print.quosures rlang
my_theme <- function(base_size = 12, base_family = "sans"){
  theme_minimal(base_size = base_size, base_family = base_family) +
  theme(
    axis.text = element_text(size = 12),
    axis.text.x = element_text(angle = 45, vjust = 0.5, hjust = 0.5),
    axis.title = element_text(size = 14),
    panel.grid.major = element_line(color = "grey"),
    panel.grid.minor = element_blank(),
    panel.background = element_rect(fill = "aliceblue"),
    strip.background = element_rect(fill = "lightgrey", color = "grey",
                                    size = 1),
    strip.text = element_text(face = "bold", size = 12, color = "black"),
    legend.position = "bottom",
    legend.justification = "top",
    legend.box = "horizontal",
    legend.box.background = element_rect(colour = "grey50"),
    legend.background = element_blank(),
    panel.border = element_rect(color = "grey", fill = NA, size = 0.5)
  )
}

ggplot(data = fluH7N9_china_2013_gather,
       aes(x = Date, y = as.numeric(age), fill = outcome)) +
  stat_density2d(aes(alpha = ..level..), geom = "polygon") +
  geom_jitter(aes(color = outcome, shape = gender), size = 1.5) +
  geom_rug(aes(color = outcome)) +
  labs(
    fill = "Outcome",
    color = "Outcome",
    alpha = "Level",
    shape = "Gender",
    x = "Date in 2013",
    y = "Age",
    title = "2013 Influenza A H7N9 cases in China",
    subtitle = "Dataset from 'outbreaks' package (Kucharski et al. 2014)",
    caption = ""
  ) +
  facet_grid(Group ~ province) +
  my_theme() +
  scale_shape_manual(values = c(15, 16, 17)) +
  scale_color_brewer(palette="Set1", na.value = "grey50") +
  scale_fill_brewer(palette="Set1")
#> Warning: Removed 149 rows containing non-finite values (stat_density2d).
#> Warning: Removed 149 rows containing missing values (geom_point).
```



This plot shows the dates of onset, hospitalisation and outcome (if known) of each data point. Outcome is marked by color and age shown on the y-axis. Gender is marked by point shape.

The density distribution of date by age for the cases seems to indicate that older people died more frequently in the Jiangsu and Zhejiang province than in Shanghai and in other provinces.

When we look at the distribution of points along the time axis, it suggests that there might be a positive correlation between the likelihood of death and an early onset or early outcome.

I also want to know how many cases there are for each gender and province and compare the genders' age distribution.

```
# more tidy data. first remove age
fluH7N9_china_2013_gather_2 <- fluH7N9_china_2013_gather[, -4] %>%
  gather(group_2, value, gender:province)
#> Warning: attributes are not identical across measure variables;
#> they will be dropped

# change descriptions
from <- c("m", "f", "unknown", "Other")
to <- c("Male", "Female", "Unknown gender", "Other province")
fluH7N9_china_2013_gather_2$value <- mapvalues(fluH7N9_china_2013_gather_2$value,
                                                 from = from, to = to)

# convert to factor
fluH7N9_china_2013_gather_2$value <- factor(fluH7N9_china_2013_gather_2$value,
                                               levels = c("Female", "Male",
                                                         "Unknown gender",
                                                         "Jiangsu", "Shanghai",
                                                         "Zhejiang", "Other
                                                         province"))

p1 <- ggplot(data = fluH7N9_china_2013_gather_2, aes(x = value,
                                                       fill = outcome,
                                                       color = outcome)) +
  geom_bar(position = "dodge", alpha = 0.7, size = 1) +
  my_theme() +
  scale_fill_brewer(palette="Set1", na.value = "grey50") +
  scale_color_brewer(palette="Set1", na.value = "grey50") +
  labs(
```

```

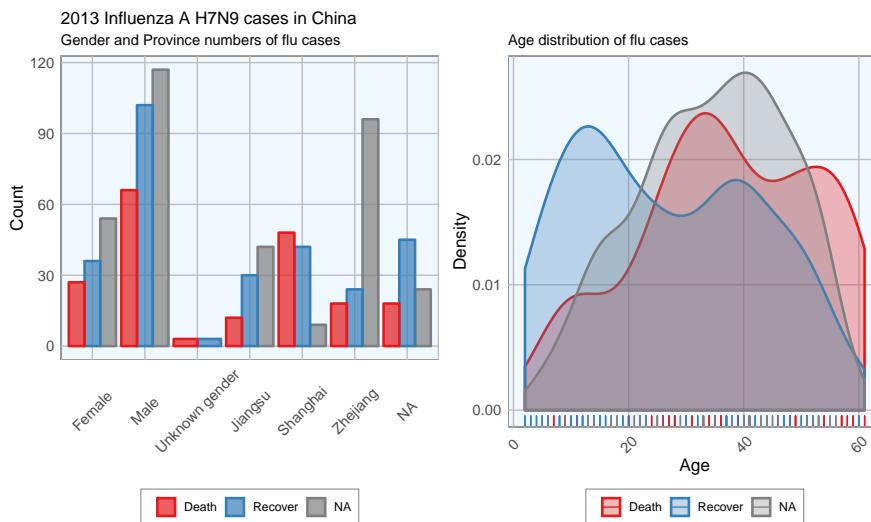
color = "",
fill = "",
x = "",
y = "Count",
title = "2013 Influenza A H7N9 cases in China",
subtitle = "Gender and Province numbers of flu cases",
caption = ""
)

p2 <- ggplot(data = fluH7N9_china_2013_gather, aes(x = as.numeric(age),
                                                    fill = outcome,
                                                    color = outcome)) +
  geom_density(alpha = 0.3, size = 1) +
  geom_rug() +
  scale_color_brewer(palette="Set1", na.value = "grey50") +
  scale_fill_brewer(palette="Set1", na.value = "grey50") +
  my_theme() +
  labs(
    color = "",
    fill = "",
    x = "Age",
    y = "Density",
    title = "",
    subtitle = "Age distribution of flu cases",
    caption = ""
  )

library(gridExtra)
library(grid)

grid.arrange(p1, p2, ncol = 2)
#> Warning: Removed 6 rows containing non-finite values (stat_density).

```



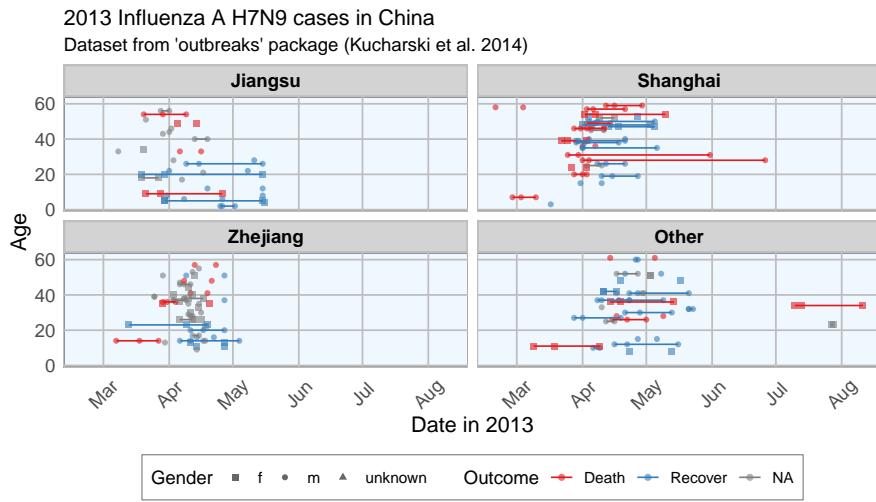
In the dataset, there are more male than female cases and correspondingly, we see more deaths, recoveries and unknown outcomes in men than in women. This is potentially a problem later on for modeling because the inherent likelihoods for outcome are not directly comparable between the sexes.

Most unknown outcomes were recorded in Zhejiang. Similarly to gender, we don't have an equal distribution of data points across provinces either.

When we look at the age distribution it is obvious that people who died tended to be slightly older than those who recovered. The density curve of unknown outcomes is more similar to that of death than of recovery, suggesting that among these people there might have been more deaths than recoveries.

And lastly, I want to plot how many days passed between onset, hospitalisation and outcome for each case.

```
ggplot(data = fluH7N9_china_2013_gather, aes(x = Date, y = as.numeric(age),
                                              color = outcome)) +
  geom_point(aes(color = outcome, shape = gender), size = 1.5, alpha = 0.6) +
  geom_path(aes(group = case_ID)) +
  facet_wrap(~ province, ncol = 2) +
  my_theme() +
  scale_shape_manual(values = c(15, 16, 17)) +
  scale_color_brewer(palette="Set1", na.value = "grey50") +
  scale_fill_brewer(palette="Set1") +
  labs(
    color = "Outcome",
    shape = "Gender",
    x = "Date in 2013",
    y = "Age",
    title = "2013 Influenza A H7N9 cases in China",
    subtitle = "Dataset from 'outbreaks' package (Kucharski et al. 2014)",
    caption = "\nTime from onset of flu to outcome."
  )
#> Warning: Removed 149 rows containing missing values (geom_point).
#> Warning: Removed 122 rows containing missing values (geom_path).
```



This plot shows that there are many missing values in the dates, so it is hard to draw a general conclusion.

6.3 Features

In Machine Learning-speak features are the variables used for model training. Using the right features dramatically influences the accuracy of the model.

Because we don't have many features, I am keeping age as it is, but I am also generating new features:

- from the date information I am calculating the days between onset and outcome and between onset and hospitalisation
- I am converting gender into numeric values with 1 for female and 0 for male
- similarly, I am converting provinces to binary classifiers (yes == 1, no == 0) for Shanghai, Zhejiang, Jiangsu and other provinces
- the same binary classification is given for whether a case was hospitalised, and whether they had an early onset or early outcome (earlier than the median date)

```
# convert gender, provinces to discrete and numeric values
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following object is masked from 'package:gridExtra':
#>
#>     combine
#> The following objects are masked from 'package:plyr':
#>
#>     arrange, count, desc, failwith, id, mutate, rename, summarise, summarize
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union

dataset <- fluH7N9_china_2013 %>%
  mutate(hospital = as.factor(ifelse(is.na(date_of_hospitalisation), 0, 1)),
         gender_f = as.factor(ifelse(gender == "f", 1, 0)),
         province_Jiangsu = as.factor(ifelse(province == "Jiangsu", 1, 0)),
         province_Shanghai = as.factor(ifelse(province == "Shanghai", 1, 0)),
         province_Zhejiang = as.factor(ifelse(province == "Zhejiang", 1, 0)),
         province_other = as.factor(ifelse(province == "Zhejiang" |
                                              province == "Jiangsu" |
                                              province == "Shanghai", 0, 1)),
         days_onset_to_outcome = as.numeric(as.character(gsub(" days", "",
                                               as.Date(as.character(date_of_outcome), format = "%Y-%m-%d") -
                                               as.Date(as.character(date_of_onset), format = "%Y-%m-%d")))),
         days_onset_to_hospital = as.numeric(as.character(gsub(" days", "",
                                               as.Date(as.character(date_of_hospitalisation), format = "%Y-%m-%d") -
                                               as.Date(as.character(date_of_onset), format = "%Y-%m-%d")))),
         age = as.numeric(as.character(age)),
         early_onset = as.factor(ifelse(date_of_onset <
                                              summary(date_of_onset)[[3]], 1, 0)),
         early_outcome = as.factor(ifelse(date_of_outcome <
                                              summary(date_of_outcome)[[3]], 1, 0))) %>%
  subset(select = -c(2:4, 6, 8))
  # print

rownames(dataset) <- dataset$case_ID
dataset <- dataset[, -c(1,4)]
```

```

head(dataset)
#>      outcome age hospital gender_f province_Jiangsu province_Shanghai province_Zhejiang province_other
#> case_1   Death  87       0       0           0           1           0
#> case_2   Death  27       1       0           0           1           0
#> case_3   Death  35       1       1           0           0           0
#> case_4   <NA>   45       1       1           1           0           0
#> case_5 Recover 48       1       1           1           0           0
#> case_6   Death  32       1       1           1           0           0

summary(dataset$outcome)
#>   Death Recover    NA's
#>     32      47      57

```

6.3.1 Imputing missing values

<https://www.r-bloggers.com/imputing-missing-data-with-r-mice-package/>

When looking at the dataset I created for modeling, it is obvious that we have quite a few missing values.

The missing values from the outcome column are what I want to predict but for the rest I would either have to remove the entire row from the data or impute the missing information. I decided to try the latter with the `mice` package.

```

library(mice)
#> Loading required package: lattice
#>
#> Attaching package: 'mice'
#> The following object is masked from 'package:tidyverse':
#>
#>     complete
#> The following objects are masked from 'package:base':
#>
#>     cbind, rbind

dataset_impute <- mice(dataset[, -1], print = FALSE) # remove outcome
#> Warning: Number of logged events: 150
impute_complete <- mice:::complete(dataset_impute, 1) # return 1st imputed dataset
rownames(impute_complete) <- row.names(dataset)
impute_complete
#>      age hospital gender_f province_Jiangsu province_Shanghai province_Zhejiang province_other d
#> case_1  87       0       0           0           1           0           0
#> case_2  27       1       0           0           1           0           0
#> case_3  35       1       1           0           0           0           1
#> case_4  45       1       1           1           0           0           0
#> case_5  48       1       1           1           0           0           0
#> case_6  32       1       1           1           0           0           0
#> case_7  83       1       0           1           0           0           0
#> case_8  38       1       0           0           0           1           0
#> case_9  67       1       0           0           0           1           0
#> case_10 48       1       0           0           1           0           0
#> case_11 64       1       0           0           0           1           0
#> case_12 52       0       1           0           1           0           0
#> case_13 67       1       1           0           1           0           0
#> case_14  4       0       0           0           1           0           0

```

```
#> case_15 61 0 1 1 0 0 0
#> case_16 79 0 0 1 0 0 0
#> case_17 74 1 0 0 1 0 0
#> case_18 66 1 0 0 1 0 0
#> case_19 59 1 0 0 1 0 0
#> case_20 55 1 0 0 0 0 1
#> case_21 67 1 0 0 1 0 0
#> case_22 85 1 0 1 0 0 0
#> case_23 25 1 1 1 0 0 0
#> case_24 64 0 0 0 1 0 0
#> case_25 62 1 0 0 1 0 0
#> case_26 77 1 0 0 1 0 0
#> case_27 51 1 1 0 0 1 0
#> case_28 79 0 0 0 0 1 0
#> case_29 76 1 1 0 1 0 0
#> case_30 81 0 1 0 1 0 0
#> case_31 70 0 0 1 0 0 0
#> case_32 74 0 0 1 0 0 0
#> case_33 65 0 0 0 0 1 0
#> case_34 74 1 0 0 1 0 0
#> case_35 83 1 1 0 1 0 0
#> case_36 68 0 0 0 1 0 0
#> case_37 31 0 0 1 0 0 0
#> case_38 56 0 0 1 0 0 0
#> case_39 66 1 0 0 0 1 0
#> case_40 74 1 0 0 0 1 0
#> case_41 54 1 1 0 0 1 0
#> case_42 53 1 0 0 1 0 0
#> case_43 86 1 0 0 1 0 0
#> case_44 7 1 1 0 0 0 1
#> case_45 56 1 0 0 1 0 0
#> case_46 77 0 1 1 0 0 0
#> case_47 72 0 0 1 0 0 0
#> case_48 65 1 0 0 0 1 0
#> case_49 38 1 0 0 0 1 0
#> case_50 34 1 0 0 0 0 1
#> case_51 65 1 0 0 0 0 1
#> case_52 64 0 1 0 0 1 0
#> case_53 62 0 1 0 0 1 0
#> case_54 75 0 0 0 0 1 0
#> case_55 79 0 0 0 0 1 0
#> case_56 73 1 0 0 1 0 0
#> case_57 54 1 0 0 1 0 0
#> case_58 78 1 0 0 1 0 0
#> case_59 50 0 0 1 0 0 0
#> case_60 26 0 0 1 0 0 0
#> case_61 60 0 0 1 0 0 0
#> case_62 68 0 1 0 0 1 0
#> case_63 60 0 0 0 0 0 1
#> case_64 56 0 0 1 0 0 0
#> case_65 21 0 1 1 0 0 0
#> case_66 72 0 0 1 0 0 0
#> case_67 56 0 0 0 0 1 0
```

#> case_68	57	0	0	0	0	1	0
#> case_69	62	0	0	0	0	1	0
#> case_70	58	0	1	0	0	1	0
#> case_71	72	0	1	0	0	1	0
#> case_72	47	1	0	0	1	0	0
#> case_73	69	0	0	0	1	0	0
#> case_74	54	0	0	0	1	0	0
#> case_75	83	0	0	0	1	0	0
#> case_76	55	0	0	0	1	0	0
#> case_77	2	0	0	0	1	0	0
#> case_78	89	1	0	0	1	0	0
#> case_79	37	0	1	0	0	1	0
#> case_80	74	0	0	0	0	1	0
#> case_81	86	0	0	0	0	1	0
#> case_82	41	0	0	0	0	1	0
#> case_83	38	0	0	0	0	0	1
#> case_84	26	0	1	1	0	0	0
#> case_85	80	1	1	0	1	0	0
#> case_86	54	0	1	0	0	1	0
#> case_87	69	0	0	0	0	1	0
#> case_88	4	0	0	0	0	0	1
#> case_89	54	1	0	1	0	0	0
#> case_90	43	1	0	0	0	1	0
#> case_91	48	1	0	0	0	1	0
#> case_92	66	1	1	0	0	1	0
#> case_93	56	0	0	0	0	1	0
#> case_94	35	0	1	0	0	1	0
#> case_95	37	0	0	0	0	1	0
#> case_96	43	0	0	1	0	0	0
#> case_97	75	1	1	0	1	0	0
#> case_98	76	0	0	0	0	1	0
#> case_99	68	0	1	0	0	1	0
#> case_100	58	0	0	0	0	1	0
#> case_101	79	0	1	0	0	1	0
#> case_102	81	0	0	0	0	1	0
#> case_103	68	1	0	1	0	0	0
#> case_104	54	0	1	0	0	1	0
#> case_105	32	0	0	0	0	1	0
#> case_106	36	1	0	0	0	0	1
#> case_107	91	0	0	0	0	0	1
#> case_108	84	0	0	0	0	1	0
#> case_109	62	0	0	0	0	1	0
#> case_110	53	1	0	0	0	0	1
#> case_111	56	0	0	0	0	0	1
#> case_112	69	0	0	0	0	0	1
#> case_113	60	0	1	0	0	1	0
#> case_114	50	0	1	0	0	1	0
#> case_115	38	0	0	0	0	1	0
#> case_116	65	1	0	0	0	0	1
#> case_117	76	0	1	0	0	0	1
#> case_118	49	0	0	1	0	0	0
#> case_119	36	0	0	1	0	0	0
#> case_120	60	0	0	1	0	0	0

```

#> case_121 64      1      1      0      0      0      0      0      1
#> case_122 38      0      0      0      0      0      1      0      0
#> case_123 54      1      0      0      0      0      0      0      1
#> case_124 80      0      0      0      0      0      0      0      1
#> case_125 31      0      1      0      0      0      0      0      1
#> case_126 80      1      0      0      0      0      0      0      1
#> case_127 4       0      0      0      0      0      0      0      1
#> case_128 58      1      0      0      0      0      0      0      1
#> case_129 69      1      0      0      0      0      0      0      1
#> case_130 69      1      0      0      0      0      0      0      1
#> case_131 9       1      0      0      0      0      0      0      1
#> case_132 79      1      1      0      0      0      0      0      1
#> case_133 6       1      0      0      0      0      0      0      1
#> case_134 15      1      0      0      1      0      0      0      0
#> case_135 61      1      1      0      0      0      0      0      1
#> case_136 51      1      1      0      0      0      0      0      1

dataset_complete <- merge(dataset[, 1, drop = FALSE],
                           # mice::complete(dataset_impute, 1),
                           impute_complete,
                           by = "row.names", all = TRUE)
rownames(dataset_complete) <- dataset_complete$Row.names
dataset_complete <- dataset_complete[, -1]
dataset_complete

#>           outcome age hospital gender_f province_Jiangsu province_Shanghai province_Zhejiang province_
#> case_1     Death  87      0      0      0      1      0
#> case_10    Death  48      1      0      0      1      0
#> case_100   <NA>  58      0      0      0      0      1
#> case_101   <NA>  79      0      1      0      0      1
#> case_102   <NA>  81      0      0      0      0      1
#> case_103   <NA>  68      1      0      1      0      0
#> case_104   <NA>  54      0      1      0      0      1
#> case_105   <NA>  32      0      0      0      0      1
#> case_106   Recover 36      1      0      0      0      0
#> case_107   Death  91      0      0      0      0      0
#> case_108   <NA>  84      0      0      0      0      1
#> case_109   <NA>  62      0      0      0      0      1
#> case_11    Death  64      1      0      0      0      1
#> case_110   <NA>  53      1      0      0      0      0
#> case_111   Death  56      0      0      0      0      0
#> case_112   <NA>  69      0      0      0      0      0
#> case_113   <NA>  60      0      1      0      0      1
#> case_114   <NA>  50      0      1      0      0      1
#> case_115   <NA>  38      0      0      0      0      1
#> case_116   Recover 65      1      0      0      0      0
#> case_117   Recover 76      0      1      0      0      0
#> case_118   <NA>  49      0      0      1      0      0
#> case_119   Recover 36      0      0      1      0      0
#> case_12    Death  52      0      1      0      1      0
#> case_120   <NA>  60      0      0      1      0      0
#> case_121   Death  64      1      1      0      0      0
#> case_122   <NA>  38      0      0      0      0      1
#> case_123   Death  54      1      0      0      0      0
#> case_124   Recover 80      0      0      0      0      0

```

```

#> case_125 Recover 31 0 1 0 0 0 0
#> case_126 <NA> 80 1 0 0 0 0 0
#> case_127 Recover 4 0 0 0 0 0 0
#> case_128 Recover 58 1 0 0 0 0 0
#> case_129 Recover 69 1 0 0 0 0 0
#> case_13 Death 67 1 1 0 0 1 0
#> case_130 <NA> 69 1 0 0 0 0 0
#> case_131 Recover 9 1 0 0 0 0 0
#> case_132 <NA> 79 1 1 0 0 0 0
#> case_133 Recover 6 1 0 0 0 0 0
#> case_134 Recover 15 1 0 1 0 0 0
#> case_135 Death 61 1 1 0 0 0 0
#> case_136 <NA> 51 1 1 0 0 0 0
#> case_14 Recover 4 0 0 0 0 1 0
#> case_15 <NA> 61 0 1 1 0 0 0
#> case_16 <NA> 79 0 0 1 0 0 0
#> case_17 Death 74 1 0 0 1 0 0
#> case_18 Recover 66 1 0 0 0 1 0
#> case_19 Death 59 1 0 0 1 0 0
#> case_2 Death 27 1 0 0 0 1 0
#> case_20 Recover 55 1 0 0 0 0 0
#> case_21 Recover 67 1 0 0 0 1 0
#> case_22 <NA> 85 1 0 1 0 0 0
#> case_23 Recover 25 1 1 1 0 0 0
#> case_24 Death 64 0 0 0 1 0 0
#> case_25 Recover 62 1 0 0 0 1 0
#> case_26 Death 77 1 0 0 1 0 0
#> case_27 Recover 51 1 1 0 0 0 1
#> case_28 <NA> 79 0 0 0 0 0 1
#> case_29 Recover 76 1 1 0 1 0 0
#> case_3 Death 35 1 1 0 0 0 0
#> case_30 Recover 81 0 1 0 1 0 0
#> case_31 <NA> 70 0 0 1 0 0 0
#> case_32 <NA> 74 0 0 1 0 0 0
#> case_33 Recover 65 0 0 0 0 0 1
#> case_34 Death 74 1 0 0 1 0 0
#> case_35 Death 83 1 1 0 0 1 0
#> case_36 Recover 68 0 0 0 0 1 0
#> case_37 Recover 31 0 0 1 0 0 0
#> case_38 <NA> 56 0 0 1 0 0 0
#> case_39 <NA> 66 1 0 0 0 0 1
#> case_4 <NA> 45 1 1 1 0 0 0
#> case_40 <NA> 74 1 0 0 0 0 1
#> case_41 <NA> 54 1 1 0 0 0 1
#> case_42 <NA> 53 1 0 0 0 1 0
#> case_43 Death 86 1 0 0 1 0 0
#> case_44 Recover 7 1 1 0 0 0 0
#> case_45 Death 56 1 0 0 1 0 0
#> case_46 Death 77 0 1 1 0 0 0
#> case_47 <NA> 72 0 0 1 0 0 0
#> case_48 <NA> 65 1 0 0 0 0 1
#> case_49 Recover 38 1 0 0 0 0 1
#> case_5 Recover 48 1 1 1 0 0 0

```

#> case_50	Recover	34	1	0	0	0	0
#> case_51	Recover	65	1	0	0	0	0
#> case_52	<NA>	64	0	1	0	0	1
#> case_53	Death	62	0	1	0	0	1
#> case_54	<NA>	75	0	0	0	0	1
#> case_55	Recover	79	0	0	0	0	1
#> case_56	<NA>	73	1	0	0	1	0
#> case_57	Recover	54	1	0	0	1	0
#> case_58	Recover	78	1	0	0	1	0
#> case_59	Recover	50	0	0	1	0	0
#> case_6	Death	32	1	1	1	0	0
#> case_60	Recover	26	0	0	1	0	0
#> case_61	Death	60	0	0	1	0	0
#> case_62	<NA>	68	0	1	0	0	1
#> case_63	<NA>	60	0	0	0	0	0
#> case_64	Recover	56	0	0	1	0	0
#> case_65	Recover	21	0	1	1	0	0
#> case_66	<NA>	72	0	0	1	0	0
#> case_67	<NA>	56	0	0	0	0	1
#> case_68	<NA>	57	0	0	0	0	1
#> case_69	<NA>	62	0	0	0	0	1
#> case_7	Death	83	1	0	1	0	0
#> case_70	<NA>	58	0	1	0	0	1
#> case_71	<NA>	72	0	1	0	0	1
#> case_72	Recover	47	1	0	0	1	0
#> case_73	Recover	69	0	0	0	1	0
#> case_74	Recover	54	0	0	0	1	0
#> case_75	Death	83	0	0	0	1	0
#> case_76	Death	55	0	0	0	1	0
#> case_77	Recover	2	0	0	0	1	0
#> case_78	Death	89	1	0	0	1	0
#> case_79	Recover	37	0	1	0	0	1
#> case_8	Death	38	1	0	0	0	1
#> case_80	<NA>	74	0	0	0	0	1
#> case_81	Death	86	0	0	0	0	1
#> case_82	Recover	41	0	0	0	0	1
#> case_83	Recover	38	0	0	0	0	0
#> case_84	<NA>	26	0	1	1	0	0
#> case_85	<NA>	80	1	1	0	1	0
#> case_86	<NA>	54	0	1	0	0	1
#> case_87	Death	69	0	0	0	0	1
#> case_88	<NA>	4	0	0	0	0	0
#> case_89	Recover	54	1	0	1	0	0
#> case_9	<NA>	67	1	0	0	0	1
#> case_90	<NA>	43	1	0	0	0	1
#> case_91	Recover	48	1	0	0	0	1
#> case_92	<NA>	66	1	1	0	0	1
#> case_93	<NA>	56	0	0	0	0	1
#> case_94	Recover	35	0	1	0	0	1
#> case_95	<NA>	37	0	0	0	0	1
#> case_96	<NA>	43	0	0	1	0	0
#> case_97	Recover	75	1	1	0	1	0
#> case_98	Death	76	0	0	0	0	1

```
#> case_99      <NA> 68      0      1      0      0      1
#>
cat("NAs before imput: ", sum(is.na(dataset)), "\n")
#> NAs before imput: 277
summary(dataset$outcome)
#>   Death Recover   NA's
#>     32       47      57
#>
cat("NAs after imput: ", sum(is.na(dataset_complete)), "\n")
#> NAs after imput: 57
summary(dataset_complete$outcome)
#>   Death Recover   NA's
#>     32       47      57
```

6.4 Test, train and validation data sets

For building the model, I am separating the imputed data frame into training and test data. Test data are the 57 cases with unknown outcome.

The training data will be further devided for validation of the models: 70% of the training data will be kept for model building and the remaining 30% will be used for model testing.

I am using the caret package for modeling.

```
train_index <- which(is.na(dataset_complete$outcome))
train_data <- dataset_complete[-train_index, ]      # 79x12
test_data  <- dataset_complete[train_index, -1]      # 57x11

library(caret)
set.seed(27)
val_index <- createDataPartition(train_data$outcome, p = 0.7, list=FALSE)
val_train_data <- train_data[val_index, ]
val_test_data  <- train_data[-val_index, ]
val_train_X <- val_train_data[,-1]
val_test_X <- val_test_data[,-1]
```

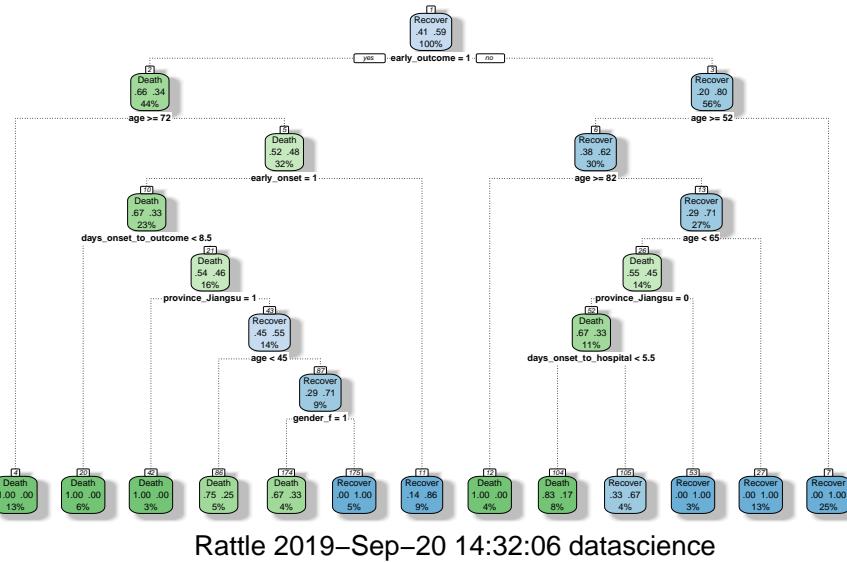
6.4.1 Decision trees

To get an idea about how each feature contributes to the prediction of the outcome, I first built a decision tree based on the training data using `rpart` and `rattle`.

```
library(rpart)
library(rattle)
#> Rattle: A free graphical interface for data science with R.
#> Version 5.2.0 Copyright (c) 2006-2018 Togaware Pty Ltd.
#> Type 'rattle()' to shake, rattle, and roll your data.
library(rpart.plot)
library(RColorBrewer)

set.seed(27)
fit <- rpart(outcome ~ ., data = train_data, method = "class",
             control = rpart.control(xval = 10, minbucket = 2, cp = 0),
             parms = list(split = "information"))
```

```
fancyRpartPlot(fit)
```



This randomly generated decision tree shows that cases with an early outcome were most likely to die when they were 68 or older, when they also had an early onset and when they were sick for fewer than 13 days. If a person was not among the first cases and was younger than 52, they had a good chance of recovering, but if they were 82 or older, they were more likely to die from the flu.

6.4.2 Feature Importance

Not all of the features I created will be equally important to the model. The decision tree already gave me an idea of which features might be most important but I also want to estimate feature importance using a Random Forest approach with repeated cross validation.

```
# prepare training scheme
control <- trainControl(method = "repeatedcv", number = 10, repeats = 10)

# train the model
set.seed(27)
model <- train(outcome ~ ., data = train_data, method = "rf",
                preProcess = NULL, trControl = control)

# estimate variable importance
importance <- varImp(model, scale=TRUE)

from <- c("age", "hospital1", "gender_f1", "province_Jiangsu1",
        "province_Shanghai1", "province_Zhejiang1", "province_other1",
        "days_onset_to_outcome", "days_onset_to_hospital", "early_onset1",
        "early_outcome1")

to <- c("Age", "Hospital", "Female", "Jiangsu", "Shanghai", "Zhejiang",
       "Other province", "Days onset to outcome", "Days onset to hospital",
       "Early onset", "Early outcome" )

importance_df_1 <- importance$importance
importance_df_1$group <- rownames(importance_df_1)
```

```

importance_df_1$group <- mapvalues(importance_df_1$group,
                                    from = from,
                                    to = to)
f = importance_df_1[order(importance_df_1$Overall, decreasing = FALSE), "group"]

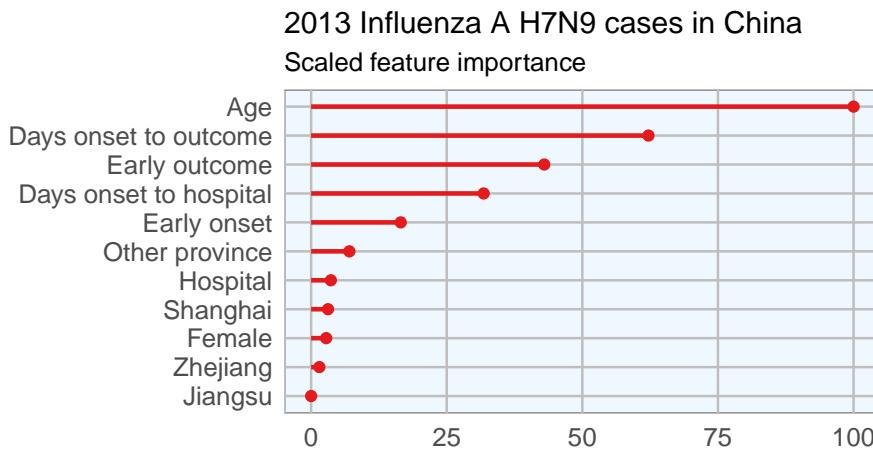
importance_df_2 <- importance_df_1
importance_df_2$Overall <- 0

importance_df <- rbind(importance_df_1, importance_df_2)

# setting factor levels
importance_df <- within(importance_df, group <- factor(group, levels = f))
importance_df_1 <- within(importance_df_1, group <- factor(group, levels = f))

ggplot() +
  geom_point(data = importance_df_1, aes(x = Overall, y = group,
                                           color = group), size = 2) +
  geom_path(data = importance_df, aes(x = Overall, y = group, color = group,
                                       group = group), size = 1) +
  scale_color_manual(values = rep(brewer.pal(1, "Set1")[1], 11)) +
  my_theme() +
  theme(legend.position = "none",
        axis.text.x = element_text(angle = 0, vjust = 0.5, hjust = 0.5)) +
  labs(
    x = "Importance",
    y = "",
    title = "2013 Influenza A H7N9 cases in China",
    subtitle = "Scaled feature importance",
    caption = "\nDetermined with Random Forest and
repeated cross validation (10 repeats, 10 times)"
  )
#> Warning in brewer.pal(1, "Set1"): minimal value for n is 3, returning requested palette with 3 different colors

```



Determined with Random Forest and
repeated cross validation (10 repeats, 10 times)

This tells me that age is the most important determining factor for predicting disease outcome, followed by days between onset and outcome, early outcome and days between onset and hospitalisation.

6.4.3 Feature Plot

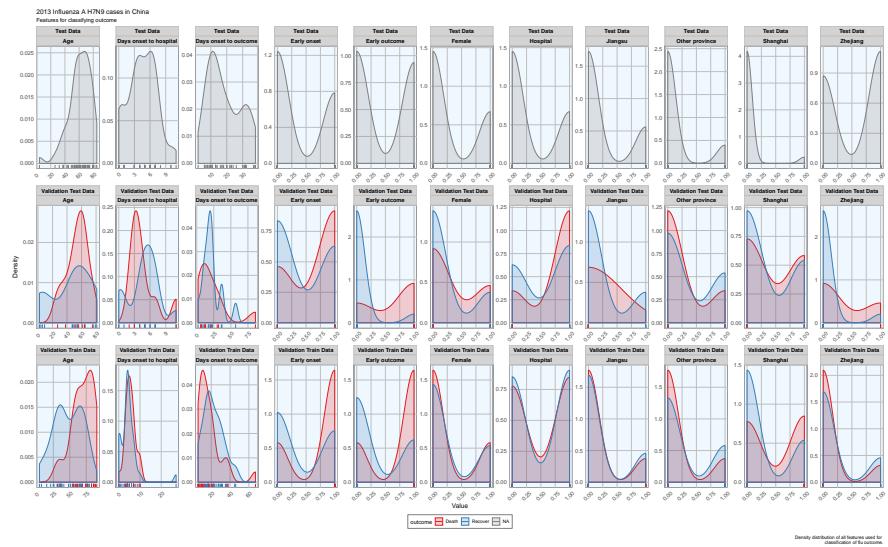
Before I start actually building models, I want to check whether the distribution of feature values is comparable between training, validation and test datasets.

```
# tidy dataframe of 11 variables for plotting features for all datasets
# dataset_complete: 136x12. test + val_train + val_test
dataset_complete_gather <- dataset_complete %>%
  mutate(set = ifelse(rownames(dataset_complete) %in% rownames(test_data),
                     "Test Data",
                     ifelse(rownames(dataset_complete) %in% rownames(val_train_data),
                            "Validation Train Data",
                            ifelse(rownames(dataset_complete) %in% rownames(val_test_data),
                                   "Validation Test Data", "NA"))),
         case_ID = rownames(.)) %>%
  gather(group, value, age:early_outcome)
#> Warning: attributes are not identical across measure variables;
#> they will be dropped

# map values in group to more readable
from <- c("age", "hospital", "gender_f", "province_Jiangsu", "province_Shanghai",
         "province_Zhejiang", "province_other", "days_onset_to_outcome",
         "days_onset_to_hospital", "early_onset", "early_outcome" )
to <- c("Age", "Hospital", "Female", "Jiangsu", "Shanghai", "Zhejiang",
       "Other province", "Days onset to outcome", "Days onset to hospital",
       "Early onset", "Early outcome" )
dataset_complete_gather$group <- mapvalues(dataset_complete_gather$group,
                                             from = from,
                                             to = to)
```

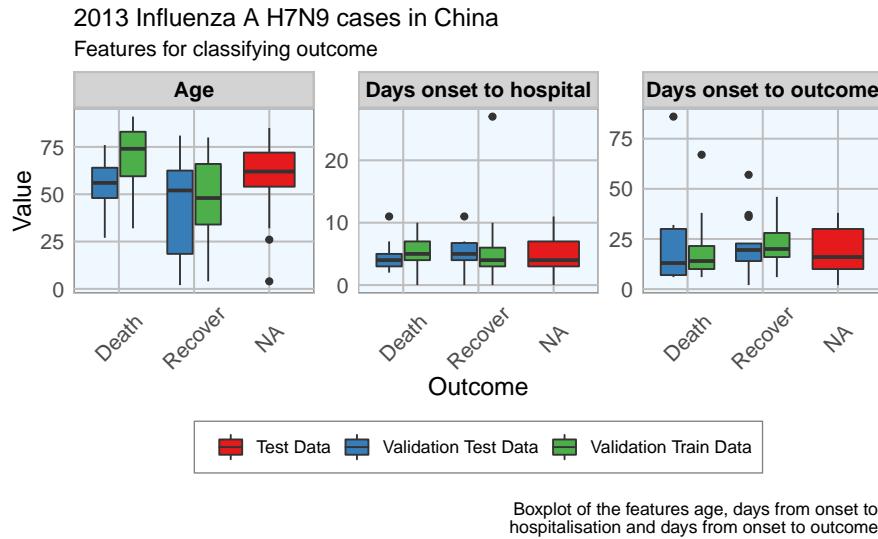
6.4.4 Plot distribution of features in each dataset

```
# plot features all datasets
ggplot(data = dataset_complete_gather,
       aes(x = as.numeric(value), fill = outcome, color = outcome)) +
  geom_density(alpha = 0.2) +
  geom_rug() +
  scale_color_brewer(palette="Set1", na.value = "grey50") +
  scale_fill_brewer(palette="Set1", na.value = "grey50") +
  my_theme() +
  facet_wrap(~ group, ncol = 11, scales = "free") +
  labs(
    x = "Value",
    y = "Density",
    title = "2013 Influenza A H7N9 cases in China",
    subtitle = "Features for classifying outcome",
    caption = "\nDensity distribution of all features used for
classification of flu outcome."
)
```



6.4.5 Plot 3 features vs outcome, all datasets

```
# plot three groups vs outcome
ggplot(subset(dataset_complete_gather,
  group == "Age" |
  group == "Days onset to hospital" |
  group == "Days onset to outcome"),
  aes(x=outcome, y=as.numeric(value), fill=set)) +
  geom_boxplot() +
  my_theme() +
  scale_fill_brewer(palette="Set1", type = "div ") +
  facet_wrap(~ group, ncol = 3, scales = "free") +
  labs(
    fill = "",
    x = "Outcome",
    y = "Value",
    title = "2013 Influenza A H7N9 cases in China",
    subtitle = "Features for classifying outcome",
    caption = "\nBoxplot of the features age, days from onset to hospitalisation and days from onset to outcome."
)
```



Luckily, the distributions looks reasonably similar between the validation and test data, except for a few outliers.

6.5 Comparing Machine Learning algorithms

Before I try to predict the outcome of the unknown cases, I am testing the models' accuracy with the validation datasets on a couple of algorithms. I have chosen only a few more well known algorithms, but `caret` implements many more.

I have chosen to not do any preprocessing because I was worried that the different data distributions with continuous variables (e.g. age) and binary variables (i.e. 0, 1 classification of e.g. hospitalisation) would lead to problems.

- Random Forest
- GLM net
- k-Nearest Neighbors
- Penalized Discriminant Analysis
- Stabilized Linear Discriminant Analysis
- Nearest Shrunken Centroids
- Single C5.0 Tree
- Partial Least Squares

```
train_control <- trainControl(method = "repeatedcv",
                               number = 10,
                               repeats = 10,
                               verboseIter = FALSE)
```

6.5.1 Random Forest

Random Forests predictions are based on the generation of multiple classification trees.

This model classified 14 out of 23 cases correctly.

```
set.seed(27)
model_rf <- caret::train(outcome ~ .,
                         data = val_train_data,
```

```

    method = "rf",
    preProcess = NULL,
    trControl = train_control)

model_rf
#> Random Forest
#>
#> 56 samples
#> 11 predictors
#> 2 classes: 'Death', 'Recover'
#>
#> No pre-processing
#> Resampling: Cross-Validated (10 fold, repeated 10 times)
#> Summary of sample sizes: 51, 49, 50, 51, 49, 51, ...
#> Resampling results across tuning parameters:
#>
#>   mtry  Accuracy  Kappa
#>     2    0.687    0.340
#>     6    0.732    0.432
#>    11    0.726    0.423
#>
#> Accuracy was used to select the optimal model using the largest value.
#> The final value used for the model was mtry = 6.

confusionMatrix(predict(model_rf, val_test_data[, -1]), val_test_data$outcome)
#> Confusion Matrix and Statistics
#>
#>           Reference
#> Prediction Death Recover
#>       Death      3       0
#>       Recover     6      14
#>
#>           Accuracy : 0.739
#>             95% CI : (0.516, 0.898)
#>   No Information Rate : 0.609
#>   P-Value [Acc > NIR] : 0.1421
#>
#>           Kappa : 0.378
#>
#>   Mcnemar's Test P-Value : 0.0412
#>
#>           Sensitivity : 0.333
#>           Specificity : 1.000
#>   Pos Pred Value : 1.000
#>   Neg Pred Value : 0.700
#>           Prevalence : 0.391
#>   Detection Rate : 0.130
#> Detection Prevalence : 0.130
#>   Balanced Accuracy : 0.667
#>
#>   'Positive' Class : Death
#>
```

6.5.2 GLM net

Lasso or elastic net regularization for generalized linear model regression are based on linear regression models and is useful when we have feature correlation in our model.

This model classified 13 out of 23 cases correctly.

```
set.seed(27)
model_glmnet <- caret::train(outcome ~.,
                               data = val_train_data,
                               method = "glmnet",
                               preProcess = NULL,
                               trControl = train_control)

model_glmnet
#> glmnet
#>
#> #> 56 samples
#> #> 11 predictors
#> #> 2 classes: 'Death', 'Recover'
#>
#> #> No pre-processing
#> #> Resampling: Cross-Validated (10 fold, repeated 10 times)
#> #> Summary of sample sizes: 51, 49, 50, 51, 49, 51, ...
#> #> Resampling results across tuning parameters:
#>
#> #>   alpha  lambda  Accuracy  Kappa
#> #>   0.10  0.000491  0.671    0.324
#> #>   0.10  0.004909  0.669    0.318
#> #>   0.10  0.049093  0.680    0.339
#> #>   0.55  0.000491  0.671    0.324
#> #>   0.55  0.004909  0.671    0.322
#> #>   0.55  0.049093  0.695    0.365
#> #>   1.00  0.000491  0.671    0.324
#> #>   1.00  0.004909  0.672    0.326
#> #>   1.00  0.049093  0.714    0.414
#>
#> #> Accuracy was used to select the optimal model using the largest value.
#> #> The final values used for the model were alpha = 1 and lambda = 0.0491.

confusionMatrix(predict(model_glmnet, val_test_data[, -1]), val_test_data$outcome)
#> Confusion Matrix and Statistics
#>
#>           Reference
#> Prediction Death Recover
#>       Death      3      2
#>       Recover     6     12
#>
#>           Accuracy : 0.652
#>                   95% CI : (0.427, 0.836)
#>       No Information Rate : 0.609
#>       P-Value [Acc > NIR] : 0.422
#>
#>           Kappa : 0.207
#>
#>       Mcnemar's Test P-Value : 0.289
```

```
#>           Sensitivity : 0.333
#>           Specificity : 0.857
#>           Pos Pred Value : 0.600
#>           Neg Pred Value : 0.667
#>           Prevalence : 0.391
#>           Detection Rate : 0.130
#>           Detection Prevalence : 0.217
#>           Balanced Accuracy : 0.595
#>
#>           'Positive' Class : Death
#>
```

6.5.3 k-Nearest Neighbors

k-nearest neighbors predicts based on point distances with predefined constants.

This model classified 14 out of 23 cases correctly.

```
set.seed(27)
model_kknn <- caret::train(outcome ~ .,
                           data = val_train_data,
                           method = "kknn",
                           preProcess = NULL,
                           trControl = train_control)

model_kknn
#> k-Nearest Neighbors
#>
#> 56 samples
#> 11 predictors
#> 2 classes: 'Death', 'Recover'
#>
#> No pre-processing
#> Resampling: Cross-Validated (10 fold, repeated 10 times)
#> Summary of sample sizes: 51, 49, 50, 51, 49, 51, ...
#> Resampling results across tuning parameters:
#>
#>   kmax  Accuracy  Kappa
#>   5     0.666    0.313
#>   7     0.653    0.274
#>   9     0.648    0.263
#>
#> Tuning parameter 'distance' was held constant at a value of 2
#> Tuning parameter 'kernel' was held constant at a value of optimal
#> Accuracy was used to select the optimal model using the largest value.
#> The final values used for the model were kmax = 5, distance = 2 and kernel = optimal.

confusionMatrix(predict(model_kknn, val_test_data[, -1]), val_test_data$outcome)
#> Confusion Matrix and Statistics
#>
#>           Reference
#>           Prediction Death Recover
#>           Death      5       3
#>           Recover     4      11
```

```
#>                               Accuracy : 0.696
#>                               95% CI : (0.471, 0.868)
#> No Information Rate : 0.609
#> P-Value [Acc > NIR] : 0.264
#>
#>                               Kappa : 0.348
#>
#> Mcnemar's Test P-Value : 1.000
#>
#>                               Sensitivity : 0.556
#>                               Specificity : 0.786
#> Pos Pred Value : 0.625
#> Neg Pred Value : 0.733
#> Prevalence : 0.391
#> Detection Rate : 0.217
#> Detection Prevalence : 0.348
#> Balanced Accuracy : 0.671
#>
#> 'Positive' Class : Death
#>
```

6.5.4 Penalized Discriminant Analysis

Penalized Discriminant Analysis is the penalized linear discriminant analysis and is also useful for when we have highly correlated features.

This model classified 14 out of 23 cases correctly.

```
set.seed(27)
model_pda <- caret::train(outcome ~ .,
                           data = val_train_data,
                           method = "pda",
                           preProcess = NULL,
                           trControl = train_control)
#> Warning: predictions failed for Fold01.Rep01: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold02.Rep01: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold03.Rep01: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold04.Rep01: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold05.Rep01: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold06.Rep01: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold07.Rep01: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold08.Rep01: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold09.Rep01: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
```



```

#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold10.Rep09: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold01.Rep10: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold02.Rep10: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold03.Rep10: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold04.Rep10: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold05.Rep10: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold06.Rep10: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold07.Rep10: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold08.Rep10: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold09.Rep10: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold10.Rep10: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning in nominalTrainWorkflow(x = x, y = y, wts = weights, info = trainInfo, : There were missing ...
#> Warning in train.default(x, y, weights = w, ...): missing values found in aggregated results
model_pda
#> Penalized Discriminant Analysis
#>
#> 56 samples
#> 11 predictors
#> 2 classes: 'Death', 'Recover'
#>
#> No pre-processing
#> Resampling: Cross-Validated (10 fold, repeated 10 times)
#> Summary of sample sizes: 51, 49, 50, 51, 49, 51, ...
#> Resampling results across tuning parameters:
#>
#>   lambda  Accuracy  Kappa
#>   0e+00    NaN       NaN
#>   1e-04    0.681    0.343
#>   1e-01    0.681    0.343
#>
#> Accuracy was used to select the optimal model using the largest value.
#> The final value used for the model was lambda = 1e-04.

confusionMatrix(predict(model_pda, val_test_data[, -1]), val_test_data$outcome)
#> Confusion Matrix and Statistics
#>
#>             Reference
#> Prediction Death Recover
#>   Death      3      2
#>   Recover     6     12
#>
#>           Accuracy : 0.652

```

```

#>          95% CI : (0.427, 0.836)
#>      No Information Rate : 0.609
#>      P-Value [Acc > NIR] : 0.422
#>
#>          Kappa : 0.207
#>
#> McNemar's Test P-Value : 0.289
#>
#>          Sensitivity : 0.333
#>          Specificity : 0.857
#>          Pos Pred Value : 0.600
#>          Neg Pred Value : 0.667
#>          Prevalence : 0.391
#>          Detection Rate : 0.130
#>          Detection Prevalence : 0.217
#>          Balanced Accuracy : 0.595
#>
#> 'Positive' Class : Death
#>

```

6.5.5 Stabilized Linear Discriminant Analysis

Stabilized Linear Discriminant Analysis is designed for high-dimensional data and correlated co-variables.

This model classified 15 out of 23 cases correctly.

```

set.seed(27)
model_slda <- caret::train(outcome ~ .,
                           data = val_train_data,
                           method = "slda",
                           preProcess = NULL,
                           trControl = train_control)

model_slda
#> Stabilized Linear Discriminant Analysis
#>
#> 56 samples
#> 11 predictors
#> 2 classes: 'Death', 'Recover'
#>
#> No pre-processing
#> Resampling: Cross-Validated (10 fold, repeated 10 times)
#> Summary of sample sizes: 51, 49, 50, 51, 49, 51, ...
#> Resampling results:
#>
#>   Accuracy  Kappa
#>   0.682     0.358

confusionMatrix(predict(model_slda, val_test_data[, -1]), val_test_data$outcome)
#> Confusion Matrix and Statistics
#>
#>           Reference
#>           Prediction Death Recover
#>           Death       3       3
#>           Recover      6      11

```

```
#> Accuracy : 0.609
#> 95% CI : (0.385, 0.803)
#> No Information Rate : 0.609
#> P-Value [Acc > NIR] : 0.590
#>
#> Kappa : 0.127
#>
#> Mcnemar's Test P-Value : 0.505
#>
#> Sensitivity : 0.333
#> Specificity : 0.786
#> Pos Pred Value : 0.500
#> Neg Pred Value : 0.647
#> Prevalence : 0.391
#> Detection Rate : 0.130
#> Detection Prevalence : 0.261
#> Balanced Accuracy : 0.560
#>
#> 'Positive' Class : Death
#>
```

6.5.6 Nearest Shrunken Centroids

Nearest Shrunken Centroids computes a standardized centroid for each class and shrinks each centroid toward the overall centroid for all classes.

This model classified 15 out of 23 cases correctly.

```
#> Accuracy was used to select the optimal model using the largest value.
#> The final value used for the model was threshold = 2.06.

confusionMatrix(predict(model_pam, val_test_data[, -1]), val_test_data$outcome)
#> Confusion Matrix and Statistics
#>
#>           Reference
#> Prediction Death Recover
#>     Death       1       3
#>     Recover      8      11
#>
#>           Accuracy : 0.522
#>           95% CI : (0.306, 0.732)
#>     No Information Rate : 0.609
#>     P-Value [Acc > NIR] : 0.857
#>
#>           Kappa : -0.115
#>
#> McNemar's Test P-Value : 0.228
#>
#>           Sensitivity : 0.1111
#>           Specificity : 0.7857
#>           Pos Pred Value : 0.2500
#>           Neg Pred Value : 0.5789
#>           Prevalence : 0.3913
#>           Detection Rate : 0.0435
#>     Detection Prevalence : 0.1739
#>           Balanced Accuracy : 0.4484
#>
#>     'Positive' Class : Death
#>
```

6.5.7 Single C5.0 Tree

C5.0 is another tree-based modeling algorithm.

This model classified 15 out of 23 cases correctly.

```
set.seed(27)
model_C5.0Tree <- caret::train(outcome ~ .,
                                 data = val_train_data,
                                 method = "C5.0Tree",
                                 preProcess = NULL,
                                 trControl = train_control)

model_C5.0Tree
#> Single C5.0 Tree
#>
#> 56 samples
#> 11 predictors
#> 2 classes: 'Death', 'Recover'
#>
#> No pre-processing
#> Resampling: Cross-Validated (10 fold, repeated 10 times)
#> Summary of sample sizes: 51, 49, 50, 51, 49, 51, ...
```

```
#> Resampling results:
#>
#>   Accuracy  Kappa
#>   0.696     0.359

confusionMatrix(predict(model_C5.0Tree, val_test_data[, -1]), val_test_data$outcome)
#> Confusion Matrix and Statistics
#>
#>           Reference
#>           Prediction Death Recover
#>           Death      4       1
#>           Recover     5      13
#>
#>           Accuracy : 0.739
#>           95% CI : (0.516, 0.898)
#>           No Information Rate : 0.609
#>           P-Value [Acc > NIR] : 0.142
#>
#>           Kappa : 0.405
#>
#>   Mcnemar's Test P-Value : 0.221
#>
#>           Sensitivity : 0.444
#>           Specificity  : 0.929
#>           Pos Pred Value : 0.800
#>           Neg Pred Value : 0.722
#>           Prevalence    : 0.391
#>           Detection Rate : 0.174
#>           Detection Prevalence : 0.217
#>           Balanced Accuracy : 0.687
#>
#>           'Positive' Class : Death
#>
```

6.5.8 Partial Least Squares

modeling with correlated features.

This model classified 15 out of 23 cases correctly.

```
set.seed(27)
model_pls <- caret::train(outcome ~ .,
                           data = val_train_data,
                           method = "pls",
                           preProcess = NULL,
                           trControl = train_control)

model_pls
#> Partial Least Squares
#>
#> 56 samples
#> 11 predictors
#> 2 classes: 'Death', 'Recover'
#>
#> No pre-processing
```

```

#> Resampling: Cross-Validated (10 fold, repeated 10 times)
#> Summary of sample sizes: 51, 49, 50, 51, 49, 51, ...
#> Resampling results across tuning parameters:
#>
#>   ncomp  Accuracy  Kappa
#>   1      0.663     0.315
#>   2      0.676     0.341
#>   3      0.691     0.376
#>
#> #> Accuracy was used to select the optimal model using the largest value.
#> #> The final value used for the model was ncomp = 3.

confusionMatrix(predict(model_pls, val_test_data[, -1]), val_test_data$outcome)
#> Confusion Matrix and Statistics
#>
#>           Reference
#>           Prediction Death Recover
#>           Death       2       3
#>           Recover      7      11
#>
#>           Accuracy : 0.565
#>           95% CI : (0.345, 0.768)
#>           No Information Rate : 0.609
#>           P-Value [Acc > NIR] : 0.742
#>
#>           Kappa : 0.009
#>
#> #> McNemar's Test P-Value : 0.343
#>
#>           Sensitivity : 0.222
#>           Specificity : 0.786
#>           Pos Pred Value : 0.400
#>           Neg Pred Value : 0.611
#>           Prevalence : 0.391
#>           Detection Rate : 0.087
#>           Detection Prevalence : 0.217
#>           Balanced Accuracy : 0.504
#>
#>           'Positive' Class : Death
#>

```

6.6 Comparing accuracy of models

All models were similarly accurate.

6.6.1 Summary Accuracy and Kappa

```

# Create a list of models
models <- list(rf      = model_rf,
               glmnet   = model_glmnet,
               kknn    = model_kknn,

```

```

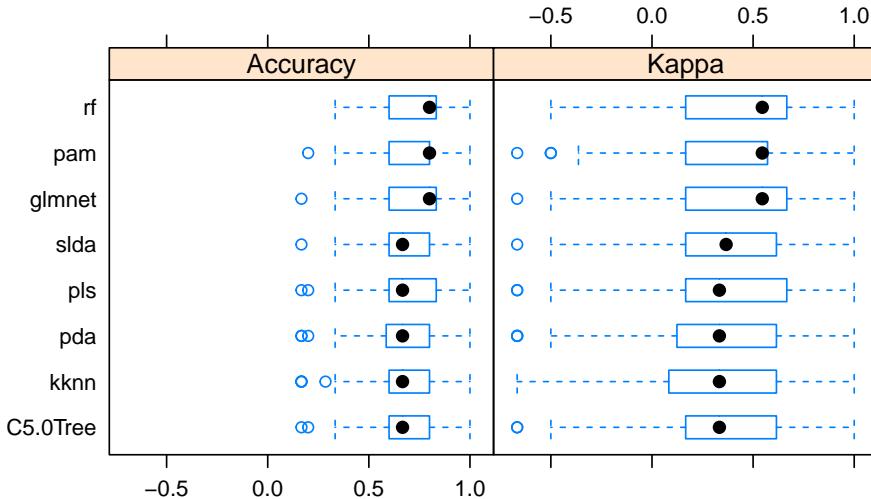
pda      = model_pda,
slda     = model_slda,
pam      = model_pam,
C5.OTree = model_C5.OTree,
pls      = model_pls)

# Resample the models
resample_results <- resamples(models)

# Generate a summary
summary(resample_results, metric = c("Kappa", "Accuracy"))
#>
#> Call:
#> summary.resamples(object = resample_results, metric = c("Kappa", "Accuracy"))
#>
#> Models: rf, glmnet, kknn, pda, slda, pam, C5.OTree, pls
#> Number of resamples: 100
#>
#> Kappa
#>      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> rf    -0.500 0.167 0.545 0.432 0.667 1 0
#> glmnet -0.667 0.167 0.545 0.414 0.667 1 0
#> kknn   -0.667 0.125 0.333 0.313 0.615 1 0
#> pda    -0.667 0.142 0.333 0.343 0.615 1 0
#> slda   -0.667 0.167 0.367 0.358 0.615 1 0
#> pam    -0.667 0.167 0.545 0.382 0.571 1 0
#> C5.OTree -0.667 0.167 0.333 0.359 0.615 1 0
#> pls    -0.667 0.167 0.333 0.376 0.667 1 0
#>
#> Accuracy
#>      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> rf    0.333 0.600 0.800 0.732 0.833 1 0
#> glmnet 0.167 0.600 0.800 0.714 0.833 1 0
#> kknn   0.167 0.600 0.667 0.666 0.800 1 0
#> pda    0.167 0.593 0.667 0.681 0.800 1 0
#> slda   0.167 0.600 0.667 0.682 0.800 1 0
#> pam    0.200 0.600 0.800 0.714 0.800 1 0
#> C5.OTree 0.167 0.600 0.667 0.696 0.800 1 0
#> pls    0.167 0.600 0.667 0.691 0.833 1 0

bwplot(resample_results , metric = c("Kappa","Accuracy"))

```



6.6.2 Combined results of predicting validation test samples

To compare the predictions from all models, I summed up the prediction probabilities for Death and Recovery from all models and calculated the log₂ of the ratio between the summed probabilities for Recovery by the summed probabilities for Death. All cases with a log₂ ratio bigger than 1.5 were defined as Recover, cases with a log₂ ratio below -1.5 were defined as Death, and the remaining cases were defined as uncertain.

```
results <- data.frame(
  randomForest = predict(model_rf, newdata = val_test_data[, -1], type="prob"),
  glmnet = predict(model_glmnet, newdata = val_test_data[, -1], type="prob"),
  kknn = predict(model_kknn, newdata = val_test_data[, -1], type="prob"),
  pda = predict(model_pda, newdata = val_test_data[, -1], type="prob"),
  slda = predict(model_slda, newdata = val_test_data[, -1], type="prob"),
  pam = predict(model_pam, newdata = val_test_data[, -1], type="prob"),
  C5.0Tree = predict(model_C5.0Tree, newdata = val_test_data[, -1], type="prob"),
  pls = predict(model_pls, newdata = val_test_data[, -1], type="prob"))

results$sum_Death <- rowSums(results[, grep("Death", colnames(results))])
results$sum_Recover <- rowSums(results[, grep("Recover", colnames(results))])
results$log2_ratio <- log2(results$sum_Recover/results$sum_Death)
results$true_outcome <- val_test_data$outcome
results$pred_outcome <- ifelse(results$log2_ratio > 1.5, "Recover",
                                ifelse(results$log2_ratio < -1.5, "Death", "uncertain"))
results$prediction <- ifelse(results$pred_outcome == results$true_outcome, "CORRECT",
                             ifelse(results$pred_outcome == "uncertain", "uncertain", "wrong"))
results[, -c(1:16)]
#>   sum_Death sum_Recover log2_ratio true_outcome pred_outcome prediction
#> case_10    4.237      3.76   -0.1709     Death   uncertain   uncertain
#> case_11    5.181      2.82   -0.8778     Death   uncertain   uncertain
#> case_116   2.412      5.59    1.2123    Recover  uncertain   uncertain
#> case_12    5.219      2.78   -0.9085     Death   uncertain   uncertain
#> case_121   2.356      5.64    1.2606     Death   uncertain   uncertain
#> case_127   0.694      7.31    3.3972    Recover  Recover    CORRECT
#> case_131   0.685      7.31    3.4164    Recover  Recover    CORRECT
#> case_133   0.649      7.35    3.5024    Recover  Recover    CORRECT
#> case_135   2.027      5.97    1.5589     Death   Recover    wrong
#> case_2     2.161      5.84    1.4337     Death   uncertain   uncertain
```

#> case_20	3.144	4.86	0.6272	Recover	uncertain	uncertain
#> case_30	4.493	3.51	-0.3576	Recover	uncertain	uncertain
#> case_45	2.594	5.41	1.0590	Death	uncertain	uncertain
#> case_5	3.019	4.98	0.7227	Recover	uncertain	uncertain
#> case_55	3.925	4.08	0.0543	Recover	uncertain	uncertain
#> case_59	1.894	6.11	1.6886	Recover	Recover	CORRECT
#> case_72	2.545	5.46	1.1002	Recover	uncertain	uncertain
#> case_74	2.339	5.66	1.2748	Recover	uncertain	uncertain
#> case_77	0.845	7.15	3.0819	Recover	Recover	CORRECT
#> case_8	2.237	5.76	1.3650	Death	uncertain	uncertain
#> case_89	1.712	6.29	1.8772	Recover	Recover	CORRECT
#> case_97	2.959	5.04	0.7687	Recover	uncertain	uncertain
#> case_98	4.798	3.20	-0.5833	Death	uncertain	uncertain

All predictions based on all models were either correct or uncertain.

6.7 Predicting unknown outcomes

The above models will now be used to predict the outcome of cases with unknown fate.

```
train_control <- trainControl(method = "repeatedcv",
                               number = 10,
                               repeats = 10,
                               verboseIter = FALSE)

set.seed(27)
model_rf <- caret::train(outcome ~ .,
                           data = train_data,
                           method = "rf",
                           preProcess = NULL,
                           trControl = train_control)
model_glmnet <- caret::train(outcome ~ .,
                             data = train_data,
                             method = "glmnet",
                             preProcess = NULL,
                             trControl = train_control)
model_kknn <- caret::train(outcome ~ .,
                            data = train_data,
                            method = "kknn",
                            preProcess = NULL,
                            trControl = train_control)
model_pda <- caret::train(outcome ~ .,
                           data = train_data,
                           method = "pda",
                           preProcess = NULL,
                           trControl = train_control)
#> Warning: predictions failed for Fold01.Rep01: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold02.Rep01: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
#> Warning: predictions failed for Fold03.Rep01: lambda=0e+00 Error in mindist[l] <- ndist[l] :
#>   NAs are not allowed in subscripted assignments
```



```

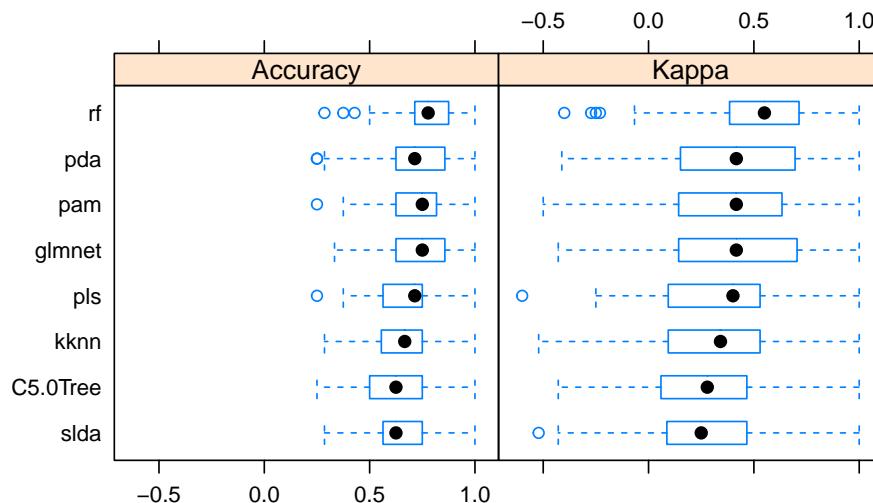
model_pls <- caret::train(outcome ~ .,
                           data = train_data,
                           method = "pls",
                           preProcess = NULL,
                           trControl = train_control)

models <- list(rf = model_rf,
                glmnet = model_glmnet,
                kknn = model_kknn,
                pda = model_pda,
                slda = model_slda,
                pam = model_pam,
                C5.0Tree = model_C5.0Tree,
                pls = model_pls)

# Resample the models
resample_results <- resamples(models)

bwplot(resample_results , metric = c("Kappa", "Accuracy"))

```



Here again, the accuracy is similar in all models.

6.7.1 Final results

The final results are calculated as described above.

```

results <- data.frame(
  randomForest = predict(model_rf, newdata = test_data, type="prob"),
  glmnet = predict(model_glmnet, newdata = test_data, type="prob"),
  kknn = predict(model_kknn, newdata = test_data, type="prob"),
  pda = predict(model_pda, newdata = test_data, type="prob"),
  slda = predict(model_slda, newdata = test_data, type="prob"),
  pam = predict(model_pam, newdata = test_data, type="prob"),
  C5.0Tree = predict(model_C5.0Tree, newdata = test_data, type="prob"),
  pls = predict(model_pls, newdata = test_data, type="prob"))

results$sum_Death <- rowSums(results[, grep("Death", colnames(results))])

```

```

results$sum_Recover <- rowSums(results[, grep("Recover", colnames(results))])
results$log2_ratio <- log2(results$sum_Recover/results$sum_Death)
results$predicted_outcome <- ifelse(results$log2_ratio > 1.5, "Recover",
                                      ifelse(results$log2_ratio < -1.5, "Death",
                                             "uncertain"))

results[, -c(1:16)]
#>   sum_Death sum_Recover log2_ratio predicted_outcome
#> case_100    1.854      6.15    1.7287       Recover
#> case_101    5.432      2.57   -1.0807     uncertain
#> case_102    2.806      5.19    0.8887     uncertain
#> case_103    2.342      5.66    1.2729     uncertain
#> case_104    1.744      6.26    1.8432       Recover
#> case_105    0.955      7.04    2.8828       Recover
#> case_108    4.489      3.51   -0.3543     uncertain
#> case_109    4.515      3.48   -0.3736     uncertain
#> case_110    2.411      5.59    1.2132     uncertain
#> case_112    2.632      5.37    1.0281     uncertain
#> case_113    2.198      5.80    1.4004     uncertain
#> case_114    3.339      4.66    0.4810     uncertain
#> case_115    1.112      6.89    2.6307       Recover
#> case_118    2.778      5.22    0.9109     uncertain
#> case_120    2.213      5.79    1.3868     uncertain
#> case_122    3.235      4.77    0.5590     uncertain
#> case_126    3.186      4.81    0.5952     uncertain
#> case_130    2.300      5.70    1.3091     uncertain
#> case_132    4.473      3.53   -0.3427     uncertain
#> case_136    3.281      4.72    0.5243     uncertain
#> case_15     2.270      5.73    1.3355     uncertain
#> case_16     2.820      5.18    0.8772     uncertain
#> case_22     4.779      3.22   -0.5689     uncertain
#> case_28     2.862      5.14    0.8445     uncertain
#> case_31     2.412      5.59    1.2117     uncertain
#> case_32     2.591      5.41    1.0616     uncertain
#> case_38     2.060      5.94    1.5280       Recover
#> case_39     4.749      3.25   -0.5466     uncertain
#> case_4      5.342      2.66   -1.0074     uncertain
#> case_40     6.550      1.45   -2.1750      Death
#> case_41     4.611      3.39   -0.4441     uncertain
#> case_42     5.570      2.43   -1.1966     uncertain
#> case_47     2.563      5.44    1.0850     uncertain
#> case_48     4.850      3.15   -0.6224     uncertain
#> case_52     4.709      3.29   -0.5173     uncertain
#> case_54     2.718      5.28    0.9586     uncertain
#> case_56     6.394      1.61   -1.9933      Death
#> case_62     6.048      1.95   -1.6319      Death
#> case_63     2.337      5.66    1.2766     uncertain
#> case_66     2.176      5.82    1.4205     uncertain
#> case_67     1.893      6.11    1.6895       Recover
#> case_68     3.907      4.09    0.0672     uncertain
#> case_69     4.465      3.53   -0.3370     uncertain
#> case_70     3.885      4.12    0.0833     uncertain
#> case_71     2.524      5.48    1.1172     uncertain
#> case_80     2.759      5.24    0.9261     uncertain

```

```

#> case_84      3.661      4.34      0.2448      uncertain
#> case_85      4.921      3.08     -0.6762      uncertain
#> case_86      3.563      4.44      0.3164      uncertain
#> case_88      0.566      7.43      3.7160      Recover
#> case_9       5.981      2.02     -1.5665      Death
#> case_90      3.570      4.43      0.3116      uncertain
#> case_92      4.664      3.34     -0.4835      uncertain
#> case_93      2.056      5.94      1.5319      Recover
#> case_95      4.495      3.50     -0.3589      uncertain
#> case_96      1.313      6.69      2.3482      Recover
#> case_99      4.799      3.20     -0.5839      uncertain
write.table(results, "results_prediction_unknown_outcome_ML_part1.txt",
            col.names = T, sep = "\t")

results %>%
  filter(predicted_outcome == "Recover") %>%
  select(-c(1:16))
#>   sum_Death sum_Recover log2_ratio predicted_outcome
#> 1   1.854      6.15      1.73      Recover
#> 2   1.744      6.26      1.84      Recover
#> 3   0.955      7.04      2.88      Recover
#> 4   1.112      6.89      2.63      Recover
#> 5   2.060      5.94      1.53      Recover
#> 6   1.893      6.11      1.69      Recover
#> 7   0.566      7.43      3.72      Recover
#> 8   2.056      5.94      1.53      Recover
#> 9   1.313      6.69      2.35      Recover

```

6.7.2 Predicted outcome

```

results %>%
  group_by(predicted_outcome) %>%
  summarize(n = n())
#> # A tibble: 3 x 2
#>   predicted_outcome     n
#>   <chr>              <int>
#> 1 Death                  4
#> 2 Recover                 9
#> 3 uncertain                44

```

From 57 cases, 14 were defined as Recover, 3 as Death and 40 as uncertain.

```

results_combined <- merge(results[, -c(1:16)],
                           fluH7N9_china_2013[which(fluH7N9_china_2013$case_ID
                                         %in% rownames(results)), ],
                           by.x = "row.names", by.y = "case_ID")
# results_combined <- results_combined[, -c(2, 3, 8, 9)]
results_combined <- results_combined[, -c(1, 2, 3, 9, 10)]
results_combined
#>   log2_ratio predicted_outcome case_id date_of_onset date_of_hospitalisation gender age province
#> 1    1.7287      Recover      100  2013-04-16             <NA>     m  58 Zhejiang
#> 2   -1.0807      uncertain     101  2013-04-13             <NA>     f  79 Zhejiang
#> 3    0.8887      uncertain     102  2013-04-12             <NA>     m  81 Zhejiang

```

#> 4	1.2729	uncertain	103	2013-04-13		2013-04-19	m	68	Jiangsu
#> 5	1.8432	Recover	104	2013-04-16		<NA>	f	54	Zhejiang
#> 6	2.8828	Recover	105	2013-04-14		<NA>	m	32	Zhejiang
#> 7	-0.3543	uncertain	108	2013-04-15		<NA>	m	84	Zhejiang
#> 8	-0.3736	uncertain	109	2013-04-15		<NA>	m	62	Zhejiang
#> 9	1.2132	uncertain	110	2013-04-12	2013-04-16	m	53	Taiwan	
#> 10	1.0281	uncertain	112	2013-04-17		<NA>	m	69	Jiangxi
#> 11	1.4004	uncertain	113	2013-04-15		<NA>	f	60	Zhejiang
#> 12	0.4810	uncertain	114	2013-04-18		<NA>	f	50	Zhejiang
#> 13	2.6307	Recover	115	2013-04-17		<NA>	m	38	Zhejiang
#> 14	0.9109	uncertain	118	2013-04-17		<NA>	m	49	Jiangsu
#> 15	1.3868	uncertain	120	2013-03-08		<NA>	m	60	Jiangsu
#> 16	0.5590	uncertain	122	2013-04-18		<NA>	m	38	Zhejiang
#> 17	0.5952	uncertain	126	2013-04-17	2013-04-27	m	80	Fujian	
#> 18	1.3091	uncertain	130	2013-04-29	2013-04-30	m	69	Fujian	
#> 19	-0.3427	uncertain	132	2013-05-03	2013-05-03	f	79	Jiangxi	
#> 20	0.5243	uncertain	136	2013-07-27	2013-07-28	f	51	Guangdong	
#> 21	1.3355	uncertain	15	2013-03-20		<NA>	f	61	Jiangsu
#> 22	0.8772	uncertain	16	2013-03-21		<NA>	m	79	Jiangsu
#> 23	-0.5689	uncertain	22	2013-03-28	2013-04-01	m	85	Jiangsu	
#> 24	0.8445	uncertain	28	2013-03-29		<NA>	m	79	Zhejiang
#> 25	1.2117	uncertain	31	2013-03-29		<NA>	m	70	Jiangsu
#> 26	1.0616	uncertain	32	2013-04-02		<NA>	m	74	Jiangsu
#> 27	1.5280	Recover	38	2013-04-03		<NA>	m	56	Jiangsu
#> 28	-0.5466	uncertain	39	2013-04-08	2013-04-08	m	66	Zhejiang	
#> 29	-1.0074	uncertain	4	2013-03-19	2013-03-27	f	45	Jiangsu	
#> 30	-2.1750	Death	40	2013-04-06	2013-04-11	m	74	Zhejiang	
#> 31	-0.4441	uncertain	41	2013-04-06	2013-04-12	f	54	Zhejiang	
#> 32	-1.1966	uncertain	42	2013-04-03	2013-04-10	m	53	Shanghai	
#> 33	1.0850	uncertain	47	2013-04-01		<NA>	m	72	Jiangsu
#> 34	-0.6224	uncertain	48	2013-04-03	2013-04-09	m	65	Zhejiang	
#> 35	-0.5173	uncertain	52	2013-04-06		<NA>	f	64	Zhejiang
#> 36	0.9586	uncertain	54	2013-04-06		<NA>	m	75	Zhejiang
#> 37	-1.9933	Death	56	2013-04-05	2013-04-11	m	73	Shanghai	
#> 38	-1.6319	Death	62	2013-04-03		<NA>	f	68	Zhejiang
#> 39	1.2766	uncertain	63	2013-04-10		<NA>	m	60	Anhui
#> 40	1.4205	uncertain	66	<NA>		<NA>	m	72	Jiangsu
#> 41	1.6895	Recover	67	2013-04-12		<NA>	m	56	Zhejiang
#> 42	0.0672	uncertain	68	2013-04-10		<NA>	m	57	Zhejiang
#> 43	-0.3370	uncertain	69	2013-04-10		<NA>	m	62	Zhejiang
#> 44	0.0833	uncertain	70	2013-04-11		<NA>	f	58	Zhejiang
#> 45	1.1172	uncertain	71	2013-04-10		<NA>	f	72	Zhejiang
#> 46	0.9261	uncertain	80	2013-04-08		<NA>	m	74	Zhejiang
#> 47	0.2448	uncertain	84	<NA>		<NA>	f	26	Jiangsu
#> 48	-0.6762	uncertain	85	2013-04-09	2013-04-16	f	80	Shanghai	
#> 49	0.3164	uncertain	86	2013-04-13		<NA>	f	54	Zhejiang
#> 50	3.7160	Recover	88	<NA>		<NA>	m	4	Beijing
#> 51	-1.5665	Death	9	2013-03-25	2013-03-25	m	67	Zhejiang	
#> 52	0.3116	uncertain	90	2013-04-12	2013-04-15	m	43	Zhejiang	
#> 53	-0.4835	uncertain	92	2013-04-10	2013-04-17	f	66	Zhejiang	
#> 54	1.5319	Recover	93	2013-04-11		<NA>	m	56	Zhejiang
#> 55	-0.3589	uncertain	95	2013-03-30		<NA>	m	37	Zhejiang
#> 56	2.3482	Recover	96	2013-04-07		<NA>	m	43	Jiangsu

```
#> 57      -0.5839      uncertain      99      2013-04-12      <NA>      f  68  Zhejiang

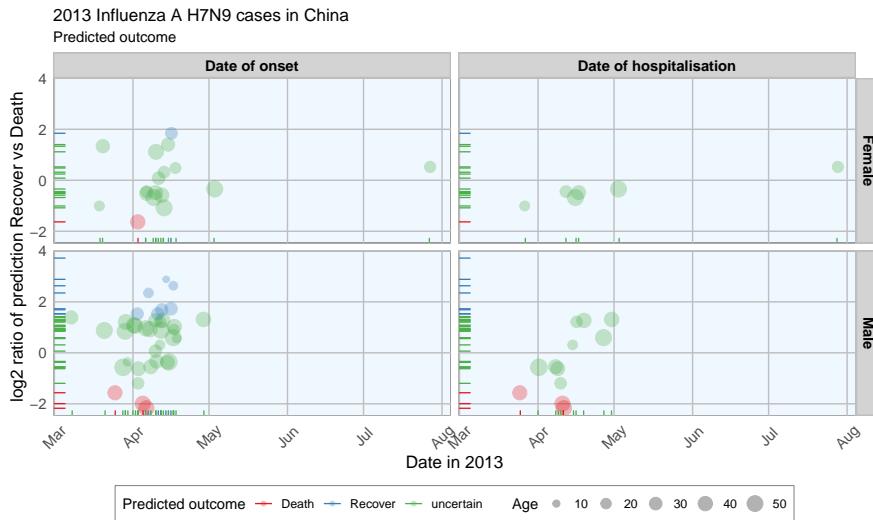
# tidy dataframe for plotting
results_combined_gather <- results_combined %>%
  gather(group_dates, date, date_of_onset:date_of_hospitalisation)

results_combined_gather$group_dates <- factor(results_combined_gather$group_dates,
                                               levels = c("date_of_onset", "date_of_hospitalisation"))

results_combined_gather$group_dates <- mapvalues(results_combined_gather$group_dates,
                                                 from = c("date_of_onset", "date_of_hospitalisation"),
                                                 to = c("Date of onset", "Date of hospitalisation"))

results_combined_gather$gender <- mapvalues(results_combined_gather$gender,
                                             from = c("f", "m"),
                                             to = c("Female", "Male"))
levels(results_combined_gather$gender) <- c(levels(results_combined_gather$gender),
                                             "unknown")
results_combined_gather$gender[is.na(results_combined_gather$gender)] <- "unknown"

ggplot(data = results_combined_gather, aes(x = date, y = log2_ratio,
                                              color = predicted_outcome)) +
  geom_jitter(aes(size = as.numeric(age)), alpha = 0.3) +
  geom_rug() +
  facet_grid(gender ~ group_dates) +
  labs(
    color = "Predicted outcome",
    size = "Age",
    x = "Date in 2013",
    y = "log2 ratio of prediction Recover vs Death",
    title = "2013 Influenza A H7N9 cases in China",
    subtitle = "Predicted outcome",
    caption = ""
  ) +
  my_theme() +
  scale_shape_manual(values = c(15, 16, 17)) +
  scale_color_brewer(palette="Set1") +
  scale_fill_brewer(palette="Set1")
#> Warning: Removed 42 rows containing missing values (geom_point).
```



The comparison of date of onset, date of hospitalisation, gender and age with predicted outcome shows that predicted deaths were associated with older age than predicted recoveries. Date of onset does not show an obvious bias in either direction.

6.8 Conclusions

This dataset posed a couple of difficulties to begin with, like unequal distribution of data points across variables and missing data. This makes the modeling inherently prone to flaws. However, real life data isn't perfect either, so I went ahead and tested the modeling success anyway.

By accounting for uncertain classification with low predictions probability, the validation data could be classified accurately. However, for a more accurate model, these few cases don't give enough information to reliably predict the outcome. More cases, more information (i.e. more features) and fewer missing data would improve the modeling outcome.

Also, this example is only applicable for this specific case of flu. In order to be able to draw more general conclusions about flu outcome, other cases and additional information, for example on medical parameters like preexisting medical conditions, disease parameters, demographic information, etc. would be necessary.

All in all, this dataset served as a nice example of the possibilities (and pitfalls) of machine learning applications and showcases a basic workflow for building prediction models with R.

For a comparison of feature selection methods see here.

If you see any mistakes or have tips and tricks for improvement, please don't hesitate to let me know! Thanks. :-)

```
sessionInfo()
#> R version 3.6.0 (2019-04-26)
#> Platform: x86_64-pc-linux-gnu (64-bit)
#> Running under: Ubuntu 18.04.3 LTS
#>
#> Matrix products: default
#> BLAS/LAPACK: /usr/lib/x86_64-linux-gnu/libopenblas-r0.2.20.so
#>
#> locale:
#> [1] LC_CTYPE=en_US.UTF-8          LC_NUMERIC=C                  LC_TIME=en_US.UTF-8           LC_COLLATE=en_US.UTF-8
```

```
#>
#> attached base packages:
#> [1] grid      stats     graphics grDevices utils      datasets  methods   base
#>
#> other attached packages:
#> [1] RColorBrewer_1.1-2 rpart.plot_3.0.7   rattle_5.2.0       rpart_4.1-15    caret_6.0-84
#>
#> loaded via a namespace (and not attached):
#> [1] nlme_3.1-139        lubridate_1.7.4    rprojroot_1.3-2   C50_0.1.2      tools_3.6.0
#> [50] pls_2.7-1          mitml_0.3-7       crayon_1.3.4     splines_3.6.0   zeallot_0.1.0
```

Part II

Classification

Chapter 7

A gentle introduction to support vector machines using R

7.1 Introduction

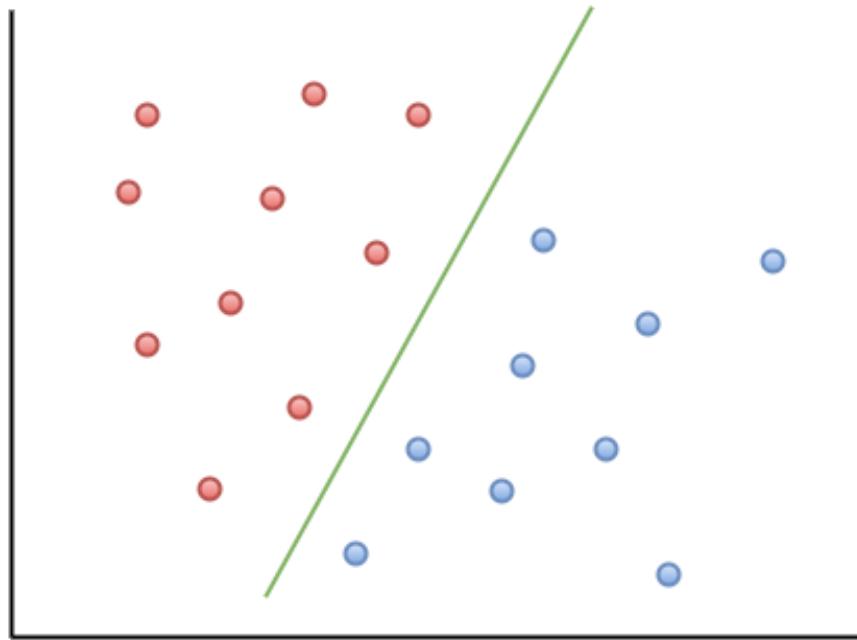
Source: <https://eight2late.wordpress.com/2017/02/07/a-gentle-introduction-to-support-vector-machines-using-r/>

Most machine learning algorithms involve minimising an error measure of some kind (this measure is often called an objective function or loss function). For example, the error measure in linear regression problems is the famous mean squared error – i.e. the averaged sum of the squared differences between the predicted and actual values. Like the mean squared error, most objective functions depend on all points in the training dataset. In this post, I describe the support vector machine (SVM) approach which focuses instead on finding the optimal separation boundary between datapoints that have different classifications. I'll elaborate on what this means in the next section.

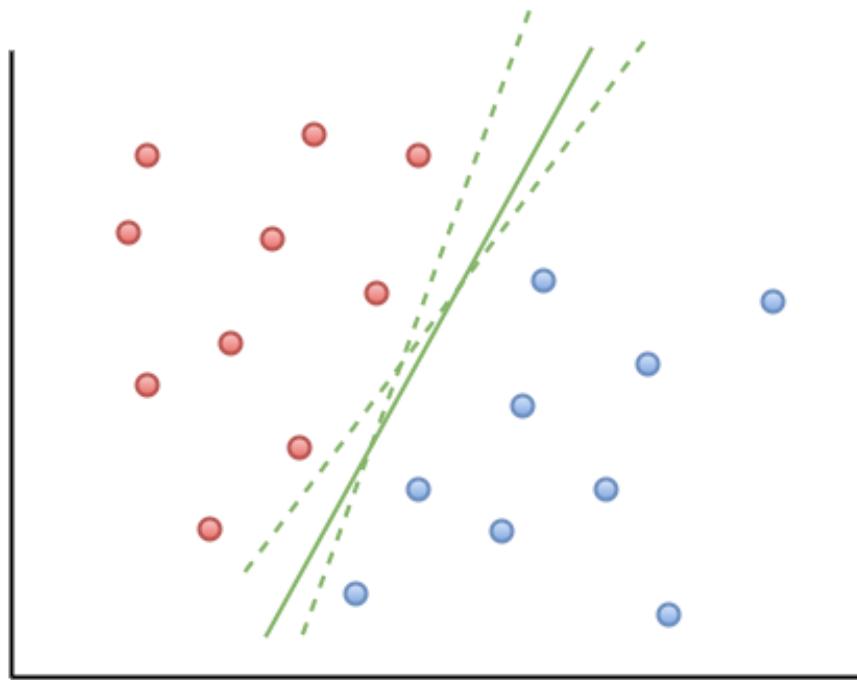
Here's the plan in brief. I'll begin with the rationale behind SVMs using a simple case of a binary (two class) dataset with a simple separation boundary (I'll clarify what "simple" means in a minute). Following that, I'll describe how this can be generalised to datasets with more complex boundaries. Finally, I'll work through a couple of examples in R, illustrating the principles behind SVMs. In line with the general philosophy of my "Gentle Introduction to Data Science Using R" series, the focus is on developing an intuitive understanding of the algorithm along with a practical demonstration of its use through a toy example.

7.2 The rationale

The basic idea behind SVMs is best illustrated by considering a simple case: a set of data points that belong to one of two classes, red and blue, as illustrated in figure 1 below. To make things simpler still, I have assumed that the boundary separating the two classes is a straight line, represented by the solid green line in the diagram. In the technical literature, such datasets are called linearly separable.



In the linearly separable case, there is usually a fair amount of freedom in the way a separating line can be drawn. Figure 2 illustrates this point: the two broken green lines are also valid separation boundaries. Indeed, because there is a non-zero distance between the two closest points between categories, there are an infinite number of possible separation lines. This, quite naturally, raises the question as to whether it is possible to choose a separation boundary that is optimal.



The short answer is, yes there is. One way to do this is to select a boundary line that maximises the margin, i.e. the distance between the separation boundary and the points that are closest to it. Such an optimal boundary is illustrated by the black brace in Figure 3. The really cool thing about this criterion is that the location of the separation boundary depends only on the points that are closest to it. This means, unlike other classification methods, the classifier does not depend on any other points in dataset. The directed

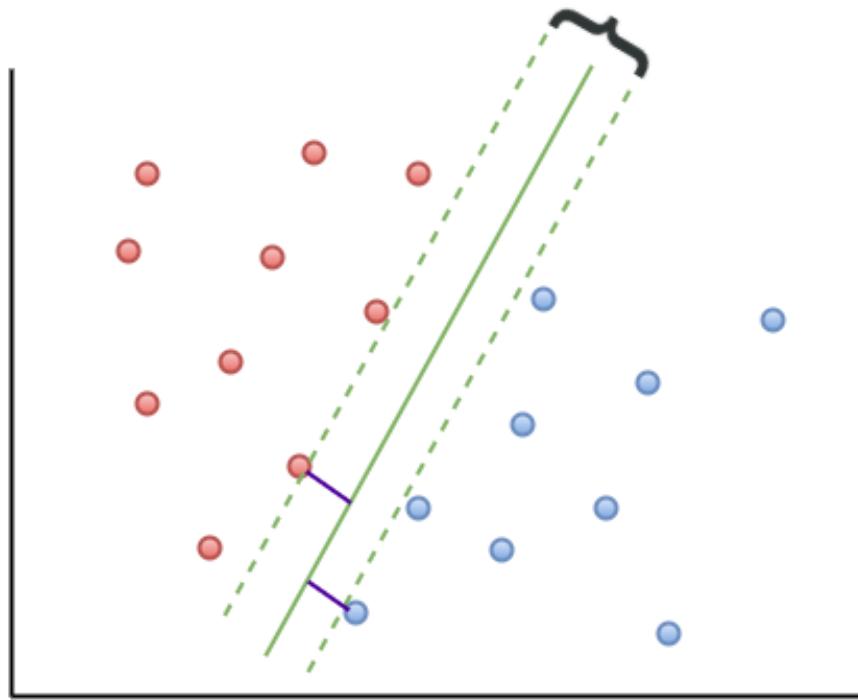


Figure 7.1: Figure 3

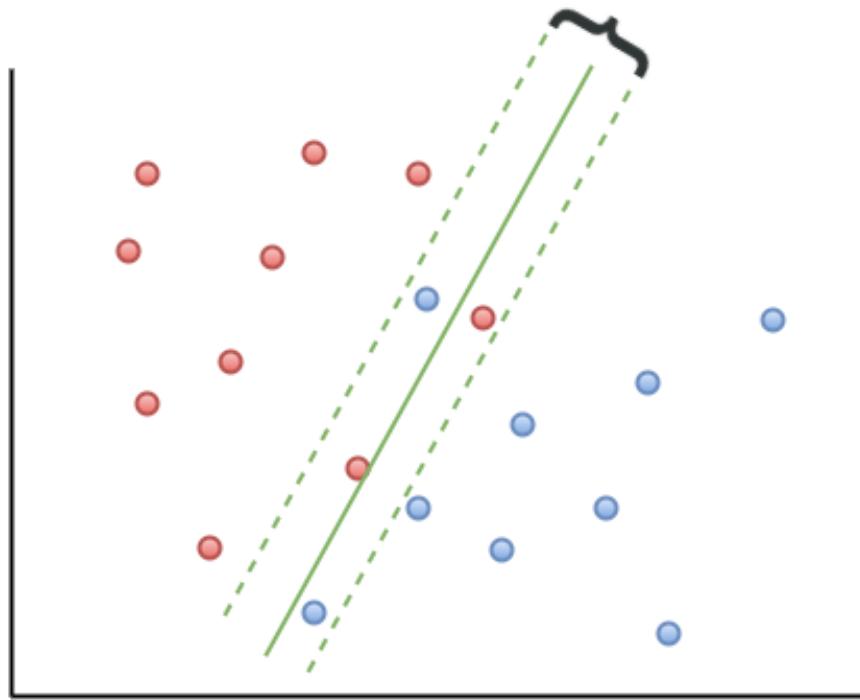
lines between the boundary and the closest points on either side are called support vectors (these are the solid black lines in figure 3). A direct implication of this is that the fewer the support vectors, the better the generalizability of the boundary.

Although the above sounds great, it is of limited practical value because real data sets are seldom (if ever) linearly separable.

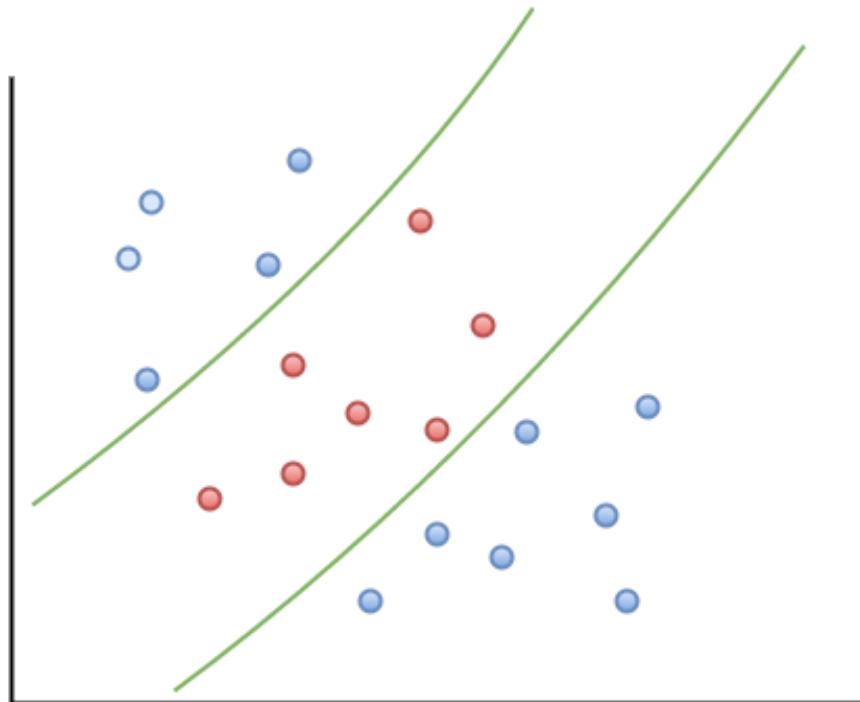
So, what can we do when dealing with real (i.e. non linearly separable) data sets?

A simple approach to tackle small deviations from linear separability is to allow a small number of points (those that are close to the boundary) to be misclassified. The number of possible misclassifications is governed by a free parameter C , which is called the cost. The cost is essentially the penalty associated with making an error: the higher the value of C , the less likely it is that the algorithm will misclassify a point.

This approach – which is called soft margin classification – is illustrated in Figure 4. Note the points on the wrong side of the separation boundary. We will demonstrate soft margin SVMs in the next section. (Note: At the risk of belabouring the obvious, the purely linearly separable case discussed in the previous para is simply a special case of the soft margin classifier.)



Real life situations are much more complex and cannot be dealt with using soft margin classifiers. For example, as shown in Figure 5, one could have widely separated clusters of points that belong to the same classes. Such situations, which require the use of multiple (and nonlinear) boundaries, can sometimes be dealt with using a clever approach called the kernel trick.



7.3 The kernel trick

Recall that in the linearly separable (or soft margin) case, the SVM algorithm works by finding a separation boundary that maximises the margin, which is the distance between the boundary and the points closest to it. The distance here is the usual straight line distance between the boundary and the closest point(s). This is called the Euclidean distance in honour of the great geometer of antiquity. The point to note is that this process results in a separation boundary that is a straight line, which as Figure 5 illustrates, does not always work. In fact in most cases it won't.

So what can we do? To answer this question, we have to take a bit of a detour...

What if we were able to generalize the notion of distance in a way that generates nonlinear separation boundaries? It turns out that this is possible. To see how, one has to first understand how the notion of distance can be generalized.

The key properties that any measure of distance must satisfy are:

Non-negativity - a distance cannot be negative, a point that needs no further explanation I reckon
 Symmetry - that is, the distance between point A and point B is the same as the distance between point B and point A.
 Identity- the distance between a point and itself is zero.

Triangle inequality - that is the sum of distances between point A and B and points B and C must be less than or equal to the distance between point A and C.

Any mathematical object that displays the above properties is akin to a distance. Such generalized distances are called metrics and the mathematical space in which they live is called a metric space. Metrics are defined using special mathematical functions designed to satisfy the above conditions. These functions are known as kernels.

The essence of the kernel trick lies in mapping the classification problem to a metric space in which the problem is rendered separable via a separation boundary that is simple in the new space, but complex – as it has to be – in the original one. Generally, the transformed space has a higher dimensionality, with each of the dimensions being (possibly complex) combinations of the original problem variables. However, this is not necessarily a problem because in practice one doesn't actually mess around with transformations, one just tries different kernels (the transformation being implicit in the kernel) and sees which one does the job. The check is simple: we simply test the predictions resulting from using different kernels against a held out subset of the data (as one would for any machine learning algorithm).

It turns out that a particular function – called the radial basis function kernel (RBF kernel) – is very effective in many cases. The RBF kernel is essentially a Gaussian (or Normal) function with the Euclidean distance between pairs of points as the variable (see equation 1 below). The basic rationale behind the RBF kernel is that it creates separation boundaries that it tends to classify points close together (in the Euclidean sense) in the original space in the same way. This is reflected in the fact that the kernel decays (i.e. drops off to zero) as the Euclidean distance between points increases.

The rate at which a kernel decays is governed by the parameter γ – the higher the value of γ , the more rapid the decay. This serves to illustrate that the RBF kernel is extremely flexible....but the flexibility comes at a price – the danger of overfitting for large values of γ . One should choose appropriate values of C and γ so as to ensure that the resulting kernel represents the best possible balance between flexibility and accuracy. We'll discuss how this is done in practice later in this article.

Finally, though it is probably obvious, it is worth mentioning that the separation boundaries for arbitrary kernels are also defined through support vectors as in Figure 3. To reiterate a point made earlier, this means that a solution that has fewer support vectors is likely to be more robust than one with many. Why? Because the data points defining support vectors are ones that are most sensitive to noise- therefore the fewer, the better.

There are many other types of kernels, each with their own pros and cons. However, I'll leave these for adventurous readers to explore by themselves. Finally, for a much more detailed....and dare I say, better... explanation of the kernel trick, I highly recommend this article by Eric Kim.

7.4 Support vector machines in R

In this demo we'll use the `svm` interface that is implemented in the `e1071` R package. This interface provides R programmers access to the comprehensive `libsvm` library written by Chang and Lin. I'll use two toy datasets: the famous iris dataset available with the base R package and the sonar dataset from the `mlbench` package. I won't describe details of the datasets as they are discussed at length in the documentation that I have linked to. However, it is worth mentioning the reasons why I chose these datasets:

As mentioned earlier, no real life dataset is linearly separable, but the iris dataset is almost so. Consequently, it is a good illustration of using linear SVMs. Although one almost never uses these in practice, I have illustrated their use primarily for pedagogical reasons. The sonar dataset is a good illustration of the benefits of using RBF kernels in cases where the dataset is hard to visualise (60 variables in this case!). In general, one would almost always use RBF (or other nonlinear) kernels in practice.

With that said, let's get right to it. I assume you have R and RStudio installed. For instructions on how to do this, have a look at the first article in this series. The processing preliminaries – loading libraries, data and creating training and test datasets are much the same as in my previous articles so I won't dwell on these here. For completeness, however, I'll list all the code so you can run it directly in R or R studio (a complete listing of the code can be found [here](#)):

7.5 SVM on `iris` dataset

7.5.1 Training and test datasets

```
#load required library
library(e1071)

#load built-in iris dataset
data(iris)

#set seed to ensure reproducible results
set.seed(42)

#split into training and test sets
iris[, "train"] <- ifelse(runif(nrow(iris)) < 0.8, 1, 0)

#separate training and test sets
trainset <- iris[iris$train == 1,]
testset <- iris[iris$train == 0,]

#get column index of train flag
trainColNum <- grep("train", names(trainset))

#remove train flag column from train and test sets
trainset <- trainset[,-trainColNum]
testset <- testset[,-trainColNum]

dim(trainset)
#> [1] 115  5
dim(testset)
#> [1] 35  5
```

7.5.2 Build the SVM model

```
#get column index of predicted variable in dataset
typeColNum <- grep("Species", names(iris))

#build model - linear kernel and C-classification (soft margin) with default cost (C=1)
svm_model <- svm(Species~., data = trainset,
                  method = "C-classification",
                  kernel = "linear")

svm_model
#>
#> Call:
#> svm(formula = Species ~ ., data = trainset, method = "C-classification",
#>       kernel = "linear")
#>
#>
#> Parameters:
#>   SVM-Type: C-classification
#>   SVM-Kernel: linear
#>   cost: 1
#>   gamma: 0.25
#>
#> Number of Support Vectors: 24
```

The output from the SVM model show that there are 24 support vectors. If desired, these can be examined using the SV variable in the model – i.e via `svm_model$SV`.

7.5.3 Support Vectors

```
# support vectors
svm_model$SV
#> Sepal.Length Sepal.Width Petal.Length Petal.Width
#> 19      -0.2564    1.7668     -1.323     -1.305
#> 42      -1.7006   -1.7045     -1.559     -1.305
#> 45      -0.9785    1.7668     -1.205     -1.171
#> 53      1.1878    0.1469      0.568      0.309
#> 55      0.7064   -0.5474      0.390      0.309
#> 57      0.4657    0.6097      0.450      0.443
#> 58      -1.2192   -1.4730     -0.378     -0.364
#> 69      0.3453   -1.9359      0.331      0.309
#> 71      -0.0157   0.3783      0.509      0.712
#> 73      0.4657   -1.2416      0.568      0.309
#> 78      0.9471   -0.0845      0.627      0.578
#> 84      0.1046   -0.7788      0.686      0.443
#> 85      -0.6174   -0.0845      0.331      0.309
#> 86      0.1046    0.8412      0.331      0.443
#> 99      -0.9785   -1.2416     -0.555     -0.229
#> 107     -1.2192   -1.2416      0.331      0.578
#> 111     0.7064    0.3783      0.686      0.981
#> 117     0.7064   -0.0845      0.922      0.712
#> 124     0.4657   -0.7788      0.568      0.712
#> 130     1.5488   -0.0845      1.099      0.443
```

```
#> 138      0.5860    0.1469    0.922    0.712
#> 139      0.1046   -0.0845    0.509    0.712
#> 147      0.4657   -1.2416    0.627    0.847
#> 150     -0.0157   -0.0845    0.686    0.712
```

The test prediction accuracy indicates that the linear performs quite well on this dataset, confirming that it is indeed near linearly separable. To check performance by class, one can create a confusion matrix as described in my post on random forests. I'll leave this as an exercise for you. Another point is that we have used a soft-margin classification scheme with a cost C=1. You can experiment with this by explicitly changing the value of C. Again, I'll leave this for you an exercise.

7.5.4 Predictions on training model

```
# training set predictions
pred_train <- predict(svm_model, trainset)
mean(pred_train == trainset$Species)
#> [1] 0.983
# [1] 0.9826087
```

7.5.5 Predictions on test model

```
# test set predictions
pred_test <- predict(svm_model, testset)
mean(pred_test == testset$Species)
#> [1] 0.914
# [1] 0.9142857
```

7.5.6 Confusion matrix and Accuracy

```
# confusion matrix
cm <- table(pred_test, testset$Species)
cm
#>
#> pred_test    setosa versicolor virginica
#>   setosa      18       0       0
#>   versicolor   0       5       3
#>   virginica    0       0       9

# accuracy
sum(diag(cm)) / sum(cm)
#> [1] 0.914
```

7.6 SVM with Radial Basis Function kernel. Linear

7.6.1 Training and test sets

```
#load required library (assuming e1071 is already loaded)
library(mlbench)

#load Sonar dataset
data(Sonar)
#set seed to ensure reproducible results
set.seed(42)
#split into training and test sets
Sonar[, "train"] <- ifelse(runif(nrow(Sonar)) < 0.8, 1, 0)

#separate training and test sets
trainset <- Sonar[Sonar$train == 1,]
testset <- Sonar[Sonar$train == 0,]

#get column index of train flag
trainColNum <- grep("train", names(trainset))
#remove train flag column from train and test sets
trainset <- trainset[, -trainColNum]
testset <- testset[, -trainColNum]

#get column index of predicted variable in dataset
typeColNum <- grep("Class", names(Sonar))
```

7.6.2 Predictions on Training model

```
#build model - linear kernel and C-classification with default cost (C=1)
svm_model <- svm(Class ~ ., data=trainset,
                  method="C-classification",
                  kernel="linear")

#training set predictions
pred_train <- predict(svm_model, trainset)
mean(pred_train == trainset$Class)
#> [1] 0.97
```

7.6.3 Predictions on test model

```
#test set predictions
pred_test <- predict(svm_model, testset)
mean(pred_test == testset$Class)
#> [1] 0.605
```

I'll leave you to examine the contents of the model. The important point to note here is that the performance of the model with the test set is quite dismal compared to the previous case. This simply indicates that the linear kernel is not appropriate here. Let's take a look at what happens if we use the RBF kernel with default values for the parameters:

7.7 SVM with Radial Basis Function kernel. Non-linear

7.7.1 Predictions on training model

```
#build model: radial kernel, default params
svm_model <- svm(Class~ ., data=trainset,
                  method="C-classification",
                  kernel="radial")

# print params
svm_model$cost
#> [1] 1
svm_model$gamma
#> [1] 0.0167

#training set predictions
pred_train <- predict(svm_model,trainset)
mean(pred_train==trainset$Class)
#> [1] 0.988
```

7.7.2 Predictions on test model

```
#test set predictions
pred_test <- predict(svm_model,testset)
mean(pred_test==testset$Class)
#> [1] 0.767
```

That's a pretty decent improvement from the linear kernel. Let's see if we can do better by doing some parameter tuning. To do this we first invoke tune.svm and use the parameters it gives us in the call to svm:

7.7.3 Tuning of parameters

```
# find optimal parameters in a specified range
tune_out <- tune.svm(x = trainset[, -typeColNum],
                      y = trainset[, typeColNum],
                      gamma = 10^{(-3:3)},
                      cost = c(0.01, 0.1, 1, 10, 100, 1000),
                      kernel = "radial")

#print best values of cost and gamma
tune_out$best.parameters$cost
#> [1] 10
tune_out$best.parameters$gamma
#> [1] 0.01

#build model
svm_model <- svm(Class~ ., data = trainset,
                  method = "C-classification",
                  kernel = "radial",
                  cost = tune_out$best.parameters$cost,
                  gamma = tune_out$best.parameters$gamma)
```

7.7.4 Prediction on training model with new parameters

```
# training set predictions
pred_train <- predict(svm_model,trainset)
mean(pred_train==trainset$Class)
#> [1] 1
```

7.7.5 Prediction on test model with new parameters

```
# test set predictions
pred_test <- predict(svm_model,testset)
mean(pred_test==testset$Class)
#> [1] 0.814
```

Which is fairly decent improvement on the un-optimised case.

7.8 Wrapping up

This bring us to the end of this introductory exploration of SVMs in R. To recap, the distinguishing feature of SVMs in contrast to most other techniques is that they attempt to construct optimal separation boundaries between different categories.

SVMs are quite versatile and have been applied to a wide variety of domains ranging from chemistry to pattern recognition. They are best used in binary classification scenarios. This brings up a question as to where SVMs are to be preferred to other binary classification techniques such as logistic regression. The honest response is, “it depends” – but here are some points to keep in mind when choosing between the two. A general point to keep in mind is that SVM algorithms tend to be expensive both in terms of memory and computation, issues that can start to hurt as the size of the dataset increases.

Given all the above caveats and considerations, the best way to figure out whether an SVM approach will work for your problem may be to do what most machine learning practitioners do: try it out!

Chapter 8

Classification with SVM. Social Network dataset

8.1 Introduction

Source: <https://www.geeksforgeeks.org/classifying-data-using-support-vector-machinessvms-in-r/>

8.2 Data Operations

8.2.1 Load libraries

```
# load packages  
library(dplyr)  
library(caTools)  
library(e1071)  
library(ElemStatLearn)
```

8.2.2 Importing dataset

```

#> 1 15624510 Male      19       19000      0
#> 2 15810944 Male      35       20000      0
#> 3 15668575 Female    26       43000      0
#> 4 15603246 Female    27       57000      0
#> 5 15804002 Male      19       76000      0
#> 6 15728773 Male      27       58000      0
#> # ... with 394 more rows

# Taking columns 3-5
dataset = dataset[3:5]
tibble::as_tibble(dataset)
#> # A tibble: 400 x 3
#>   Age   EstimatedSalary Purchased
#>   <int>       <int>     <int>
#> 1 19       19000      0
#> 2 35       20000      0
#> 3 26       43000      0
#> 4 27       57000      0
#> 5 19       76000      0
#> 6 27       58000      0
#> # ... with 394 more rows

# Encoding the target feature as factor
dataset$Purchased = factor(dataset$Purchased, levels = c(0, 1))
str(dataset)
#> 'data.frame': 400 obs. of 3 variables:
#> $ Age : int 19 35 26 27 19 27 27 32 25 35 ...
#> $ EstimatedSalary: int 19000 20000 43000 57000 76000 58000 84000 150000 33000 65000 ...
#> $ Purchased : Factor w/ 2 levels "0","1": 1 1 1 1 1 1 1 2 1 1 ...

# Splitting the dataset into the Training set and Test set
set.seed(123)
split = sample.split(dataset$Purchased, SplitRatio = 0.75)

training_set = subset(dataset, split == TRUE)
test_set = subset(dataset, split == FALSE)

dim(training_set)
#> [1] 300 3
dim(test_set)
#> [1] 100 3

# Feature Scaling
training_set[-3] = scale(training_set[-3])
test_set[-3] = scale(test_set[-3])

# Fitting SVM to the Training set
classifier = svm(formula = Purchased ~ .,
                 data = training_set,
                 type = 'C-classification',
                 kernel = 'linear')

classifier
#>
#> Call:
#> svm(formula = Purchased ~ ., data = training_set, type = "C-classification",

```

```

#>      kernel = "linear")
#>
#>
#> Parameters:
#>   SVM-Type: C-classification
#>   SVM-Kernel: linear
#>   cost: 1
#>   gamma: 0.5
#>
#> Number of Support Vectors: 116

summary(classifier)
#>
#> Call:
#> sum(formula = Purchased ~ ., data = training_set, type = "C-classification",
#>   kernel = "linear")
#>
#>
#> Parameters:
#>   SVM-Type: C-classification
#>   SVM-Kernel: linear
#>   cost: 1
#>   gamma: 0.5
#>
#> Number of Support Vectors: 116
#>
#> ( 58 58 )
#>
#>
#> Number of Classes: 2
#>
#> Levels:
#> 0 1

# Predicting the Test set results
y_pred = predict(classifier, newdata = test_set[-3])
y_pred
#>  2   4   5   9  12  18  19  20  22  29  32  34  35  38  45  46  48  52
#>  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0
#>  66  69  74  75  82  84  85  86  87  89 103 104 107 108 109 117 124 126
#>  0   0   0   0   0   0   0   0   0   0   0   0   1   0   0   0   0   0   0
#> 127 131 134 139 148 154 156 159 162 163 170 175 176 193 199 200 208 213
#>  0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   0   1
#> 224 226 228 229 230 234 236 237 239 241 255 264 265 266 273 274 281 286
#>  1   0   1   0   1   1   1   0   1   1   1   0   1   1   1   1   1   1   0
#> 292 299 302 305 307 310 316 324 326 332 339 341 343 347 353 363 364 367
#>  1   1   1   0   1   0   0   0   0   1   0   1   0   1   1   0   1   1
#> 368 369 372 373 380 383 389 392 395 400
#>  1   0   1   0   1   1   0   0   0   0
#> Levels: 0 1

# Making the Confusion Matrix
cm = table(test_set[, 3], y_pred)
cm

```

```

#>      y_pred
#>      0  1
#>  0 57  7
#>  1 13 23

xtable::xtable(cm)
#> % latex table generated in R 3.6.0 by xtable 1.8-4 package
#> % Fri Sep 20 14:33:21 2019
#> \begin{table}[ht]
#> \centering
#> \begin{tabular}{rrr}
#>   \hline
#> & 0 & 1 \\
#> \hline
#> 0 & 57 & 7 \\
#> 1 & 13 & 23 \\
#> \hline
#> \end{tabular}
#> \end{table}

# installing library ElemStatLearn
# library(ElemStatLearn)

# Plotting the training data set results
set = training_set
X1 = seq(min(set[, 1]) - 1, max(set[, 1]) + 1, by = 0.01)
X2 = seq(min(set[, 2]) - 1, max(set[, 2]) + 1, by = 0.01)

grid_set = expand.grid(X1, X2)
colnames(grid_set) = c('Age', 'EstimatedSalary')
y_grid = predict(classifier, newdata = grid_set)

plot(set[, -3],
     main = 'SVM (Training set)',
     xlab = 'Age', ylab = 'Estimated Salary',
     xlim = range(X1), ylim = range(X2))

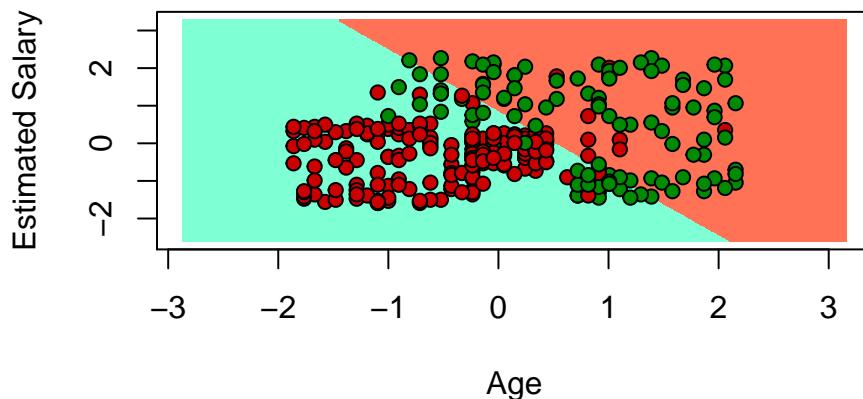
contour(X1, X2, matrix(as.numeric(y_grid), length(X1), length(X2)), add = TRUE)

points(grid_set, pch = '.', col = ifelse(y_grid == 1, 'coral1', 'aquamarine'))

points(set, pch = 21, bg = ifelse(set[, 3] == 1, 'green4', 'red3'))

```

SVM (Training set)



```

set = test_set
X1 = seq(min(set[, 1]) - 1, max(set[, 1]) + 1, by = 0.01)
X2 = seq(min(set[, 2]) - 1, max(set[, 2]) + 1, by = 0.01)

grid_set = expand.grid(X1, X2)
colnames(grid_set) = c('Age', 'EstimatedSalary')
y_grid = predict(classifier, newdata = grid_set)

plot(set[, -3], main = 'SVM (Test set)',
      xlab = 'Age', ylab = 'Estimated Salary',
      xlim = range(X1), ylim = range(X2))

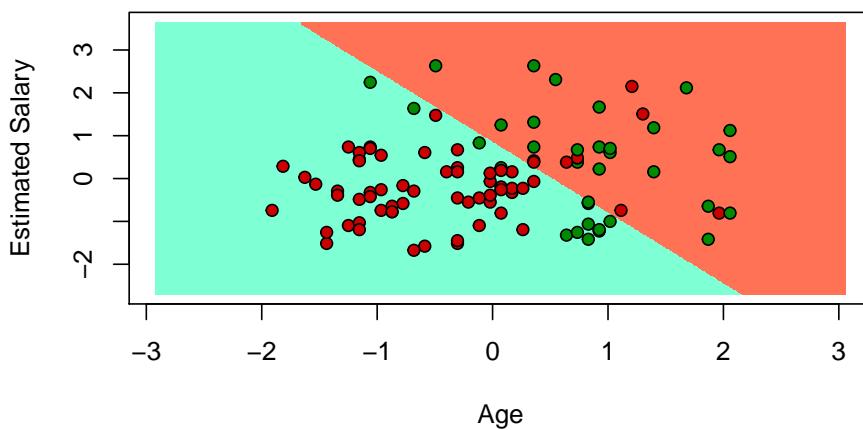
contour(X1, X2, matrix(as.numeric(y_grid), length(X1), length(X2)), add = TRUE)

points(grid_set, pch = '.', col = ifelse(y_grid == 1, 'coral1', 'aquamarine'))

points(set, pch = 21, bg = ifelse(set[, 3] == 1, 'green4', 'red3'))

```

SVM (Test set)



Chapter 9

Broad view of SVM

9.1 Introduction

Source: <http://uc-r.github.io/svm>

```
# set pseudorandom number generator
set.seed(10)

# Attach Packages
library(tidyverse)      # data manipulation and visualization
#> Registered S3 methods overwritten by 'ggplot2':
#>   method           from
#>   [.quosures       rlang
#>   c.quosures       rlang
#>   print.quosures  rlang
#> Registered S3 method overwritten by 'rvest':
#>   method           from
#>   read_xml.response xml2
#> -- Attaching packages ----- tidyverse 1.2.1 --
#> v ggplot2 3.1.1      v purrrr  0.3.2
#> v tibble  2.1.1      v dplyr    0.8.0.1
#> v tidyrr  0.8.3      v stringr  1.4.0
#> v readr   1.3.1      vforcats  0.4.0
#> -- Conflicts ----- tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()   masks stats::lag()
library(kernlab)        # SVM methodology
#>
#> Attaching package: 'kernlab'
#> The following object is masked from 'package:purrr':
#>
#>   cross
#> The following object is masked from 'package:ggplot2':
#>
#>   alpha
library(e1071)          # SVM methodology
library(ISLR)            # contains example data set "Khan"
library(RColorBrewer)    # customized coloring of plots
```

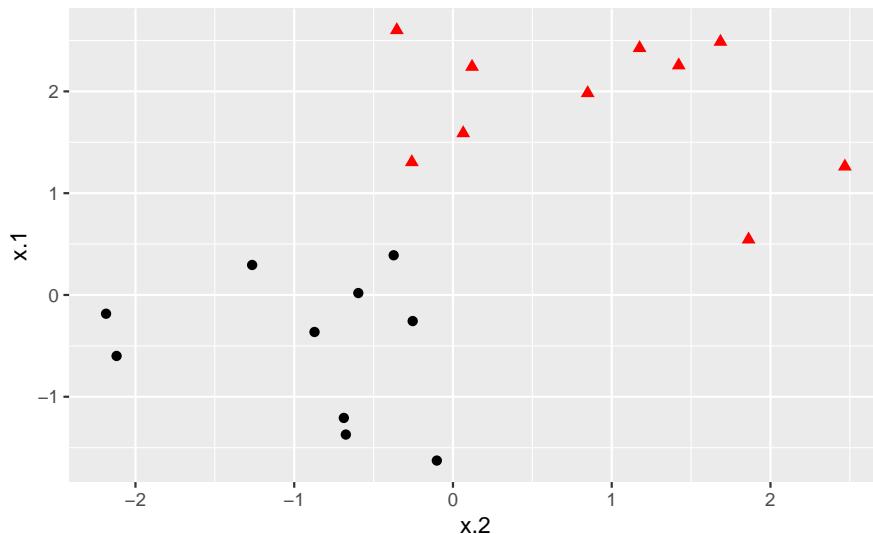
The data sets used in the tutorial (with the exception of Khan) will be generated using built-in R commands. The Support Vector Machine methodology is sound for any number of dimensions, but becomes difficult to visualize for more than 2. As previously mentioned, SVMs are robust for any number of classes, but we will stick to no more than 3 for the duration of this tutorial.

9.2 Maximal Margin Classifier

If the classes are separable by a linear boundary, we can use a Maximal Margin Classifier to find the classification boundary. To visualize an example of separated data, we generate 40 random observations and assign them to two classes. Upon visual inspection, we can see that infinitely many lines exist that split the two classes.

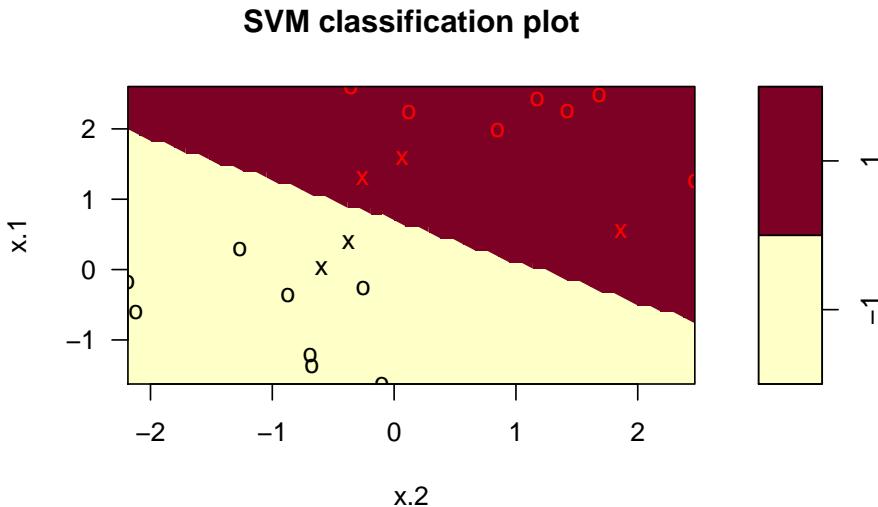
```
# Construct sample data set - completely separated
x <- matrix(rnorm(20*2), ncol = 2)
y <- c(rep(-1,10), rep(1,10))
x[y==1,] <- x[y==1,] + 3/2
dat <- data.frame(x=x, y=as.factor(y))

# Plot data
ggplot(data = dat, aes(x = x.2, y = x.1, color = y, shape = y)) +
  geom_point(size = 2) +
  scale_color_manual(values=c("#000000", "#FF0000")) +
  theme(legend.position = "none")
```



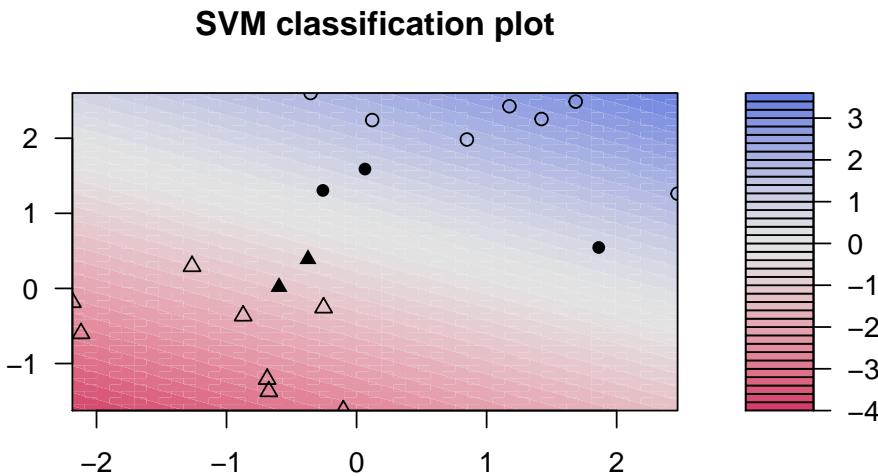
The goal of the maximal margin classifier is to identify the linear boundary that maximizes the total distance between the line and the closest point in each class. We can use the `svm()` function in the `e1071` package to find this boundary.

```
# Fit Support Vector Machine model to data set
svmfit <- svm(y~., data = dat, kernel = "linear", scale = FALSE)
# Plot Results
plot(svmfit, dat)
```



In the plot, points that are represented by an “X” are the support vectors, or the points that directly affect the classification line. The points marked with an “o” are the other points, which don’t affect the calculation of the line. This principle will lay the foundation for support vector machines. The same plot can be generated using the kernlab package, with the following results:

```
# fit model and produce plot
kernfit <- ksvm(x, y, type = "C-svc", kernel = 'vanilladot')
#> Setting default kernel parameters
plot(kernfit, data = x)
```



`kernlab` shows a little more detail than `e1071`, showing a color gradient that indicates how confidently a new point would be classified based on its features. Just as in the first plot, the support vectors are marked, in this case as filled-in points, while the classes are denoted by different shapes.

9.3 Support Vector Classifiers

As convenient as the maximal marginal classifier is to understand, most real data sets will not be fully separable by a linear boundary. To handle such data, we must use modified methodology. We simulate a new data set where the classes are more mixed.

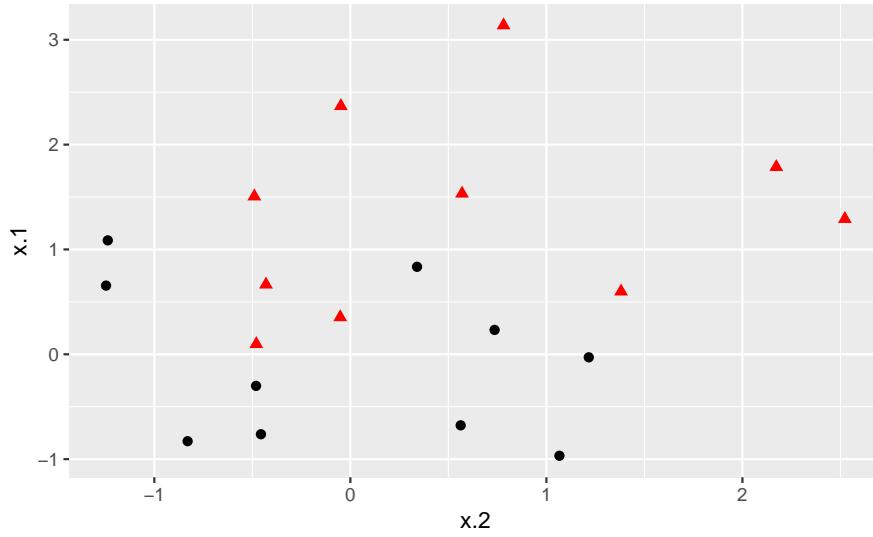
```
# Construct sample data set - not completely separated
x <- matrix(rnorm(20*2), ncol = 2)
```

```

y <- c(rep(-1,10), rep(1,10))
x[y==1,] <- x[y==1,] + 1
dat <- data.frame(x=x, y=as.factor(y))

# Plot data set
ggplot(data = dat, aes(x = x.2, y = x.1, color = y, shape = y)) +
  geom_point(size = 2) +
  scale_color_manual(values=c("#000000", "#FF0000")) +
  theme(legend.position = "none")

```

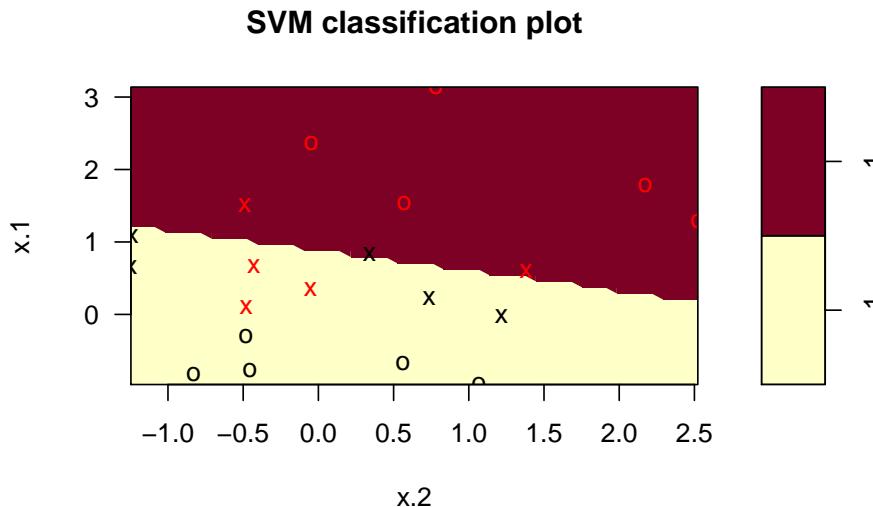


Whether the data is separable or not, the `svm()` command syntax is the same. In the case of data that is not linearly separable, however, the `cost` = argument takes on real importance. This quantifies the penalty associated with having an observation on the wrong side of the classification boundary. We can plot the fit in the same way as the completely separable case. We first use `e1071`:

```

# Fit Support Vector Machine model to data set
svmfite <- svm(y~., data = dat, kernel = "linear", cost = 10)
# Plot Results
plot(svmfite, dat)

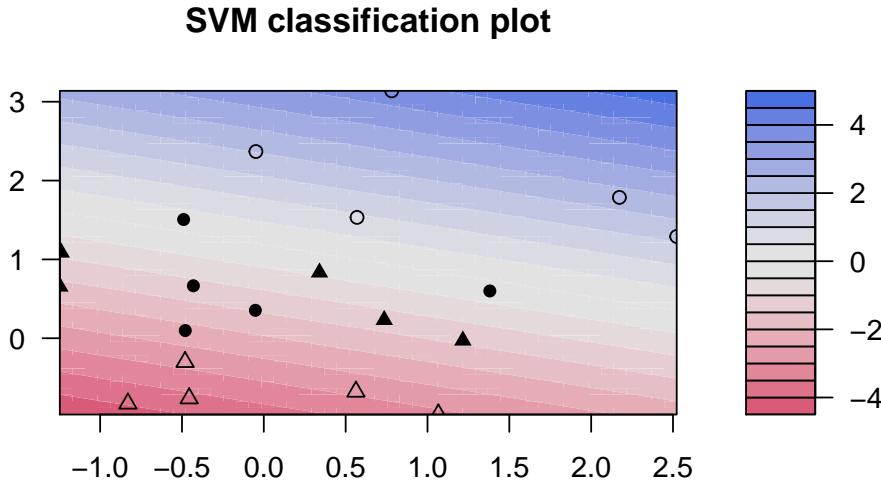
```



By upping the cost of misclassification from 10 to 100, you can see the difference in the classification line.

We repeat the process of plotting the SVM using the `kernlab` package:

```
# Fit Support Vector Machine model to data set
kernfit <- ksvm(x,y, type = "C-svc", kernel = 'vanilladot', C = 100)
##> Setting default kernel parameters
# Plot results
plot(kernfit, data = x)
```



But how do we decide how costly these misclassifications actually are? Instead of specifying a cost up front, we can use the `tune()` function from `e1071` to test various costs and identify which value produces the best fitting model.

```
# find optimal cost of misclassification
tune.out <- tune(svm, y~., data = dat, kernel = "linear",
                  ranges = list(cost = c(0.001, 0.01, 0.1, 1, 5, 10, 100)))
# extract the best model
(bestmod <- tune.out$best.model)
##>
##> Call:
##> best.tune(method = svm, train.x = y ~ ., data = dat, ranges = list(cost = c(0.001,
##>     0.01, 0.1, 1, 5, 10, 100)), kernel = "linear")
##>
##>
##> Parameters:
##>   SVM-Type: C-classification
##>   SVM-Kernel: linear
##>     cost: 0.1
##>     gamma: 0.5
##>
##> Number of Support Vectors: 16
```

For our data set, the optimal cost (from amongst the choices we provided) is calculated to be 0.1, which doesn't penalize the model much for misclassified observations. Once this model has been identified, we can construct a table of predicted classes against true classes using the `predict()` command as follows:

```
# Create a table of misclassified observations
ypred <- predict(bestmod, dat)
(misclass <- table(predict = ypred, truth = dat$y))
##>      truth
##> predict -1 1
##>      -1 9 3
```

```
#>      1  1  7
```

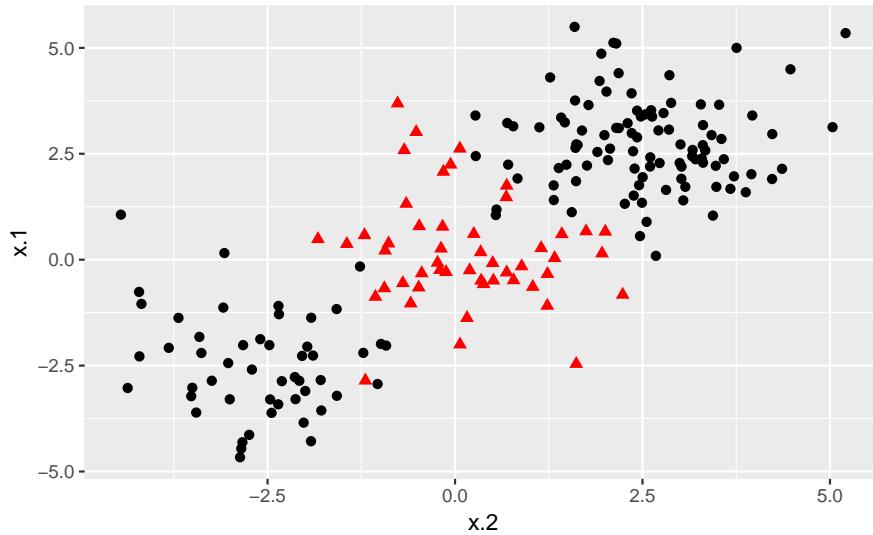
Using this support vector classifier, 80% of the observations were correctly classified, which matches what we see in the plot. If we wanted to test our classifier more rigorously, we could split our data into training and testing sets and then see how our SVC performed with the observations not used to construct the model. We will use this training-testing method later in this tutorial to validate our SVMs.

9.4 Support Vector Machines

Support Vector Classifiers are a subset of the group of classification structures known as Support Vector Machines. Support Vector Machines can construct classification boundaries that are nonlinear in shape. The options for classification structures using the `svm()` command from the e1071 package are linear, polynomial, radial, and sigmoid. To demonstrate a nonlinear classification boundary, we will construct a new data set.

```
# construct larger random data set
x <- matrix(rnorm(200*2), ncol = 2)
x[1:100,] <- x[1:100,] + 2.5
x[101:150,] <- x[101:150,] - 2.5
y <- c(rep(1,150), rep(2,50))
dat <- data.frame(x=x,y=as.factor(y))

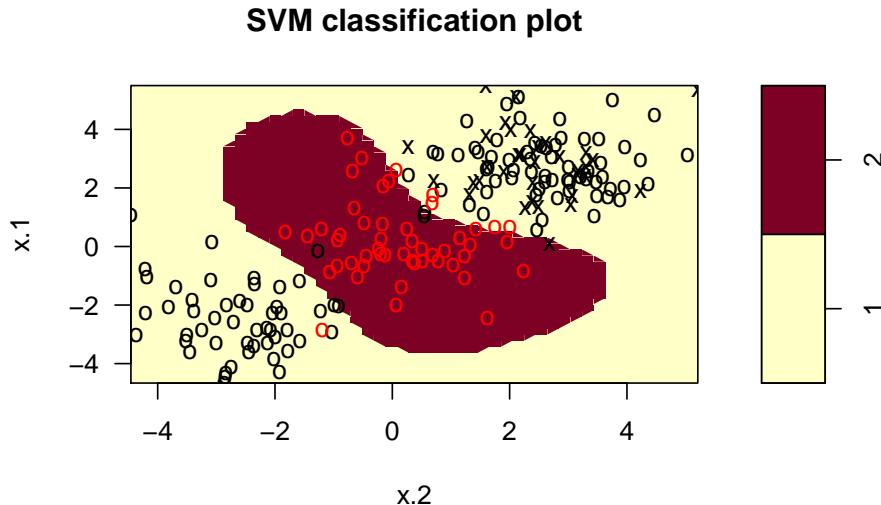
# Plot data
ggplot(data = dat, aes(x = x.2, y = x.1, color = y, shape = y)) +
  geom_point(size = 2) +
  scale_color_manual(values=c("#000000", "#FF0000")) +
  theme(legend.position = "none")
```



Notice that the data is not linearly separable, and furthermore, isn't all clustered together in a single group. There are two sections of class 1 observations with a cluster of class 2 observations in between. To demonstrate the power of SVMs, we'll take 100 random observations from the set and use them to construct our boundary. We set kernel = "radial" based on the shape of our data and plot the results.

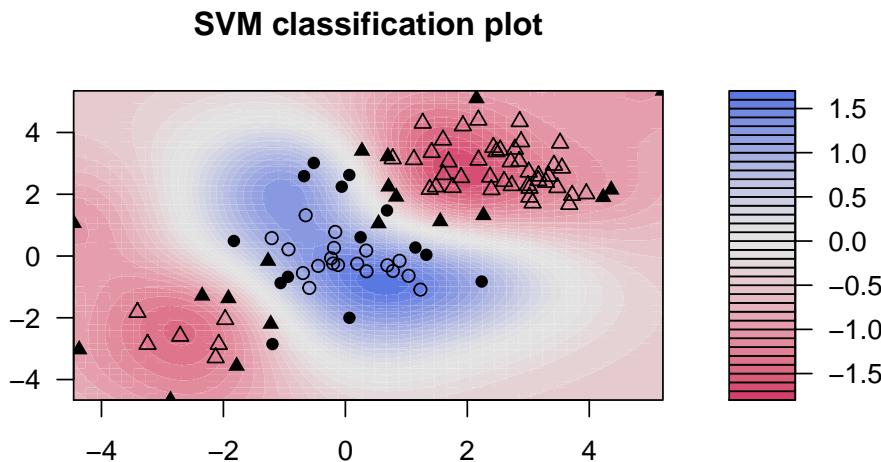
```
# set pseudorandom number generator
set.seed(123)
# sample training data and fit model
train <- base::sample(200,100, replace = FALSE)
svmfit <- svm(y~., data = dat[train,], kernel = "radial", gamma = 1, cost = 1)
```

```
# plot classifier
plot(svmfit, dat)
```



The same procedure can be run using the kernlab package, which has far more kernel options than the corresponding function in e1071. In addition to the four choices in e1071, this package allows use of a hyperbolic tangent, Laplacian, Bessel, Spline, String, or ANOVA RBF kernel. To fit this data, we set the cost to be the same as it was before, 1.

```
# Fit radial-based SVM in kernlab
kernfit <- ksvm(x[train],y[train], type = "C-svc", kernel = 'rbfdot', C = 1, scaled = c())
# Plot training data
plot(kernfit, data = x[train,])
```



We see that, at least visually, the SVM does a reasonable job of separating the two classes. To fit the model, we used `cost = 1`, but as mentioned previously, it isn't usually obvious which cost will produce the optimal classification boundary. We can use the `tune()` command to try several different values of cost as well as several different values of γ , a scaling parameter used to fit nonlinear boundaries.

```
# tune model to find optimal cost, gamma values
tune.out <- tune(svm, y~, data = dat[train], kernel = "radial",
                  ranges = list(cost = c(0.1,1,10,100,1000),
                                gamma = c(0.5,1,2,3,4)))
# show best model
tune.out$best.model
```

```
#>
#> Call:
#> best.tune(method = "svm", train.x = y ~ ., data = dat[train, ],
#>             ranges = list(cost = c(0.1, 1, 10, 100, 1000), gamma = c(0.5,
#>                 1, 2, 3, 4)), kernel = "radial")
#>
#>
#> Parameters:
#>   SVM-Type: C-classification
#>   SVM-Kernel: radial
#>     cost: 1
#>     gamma: 0.5
#>
#> Number of Support Vectors: 30
```

The model that reduces the error the most in the training data uses a cost of 1 and γ value of 0.5. We can now see how well the SVM performs by predicting the class of the 100 testing observations:

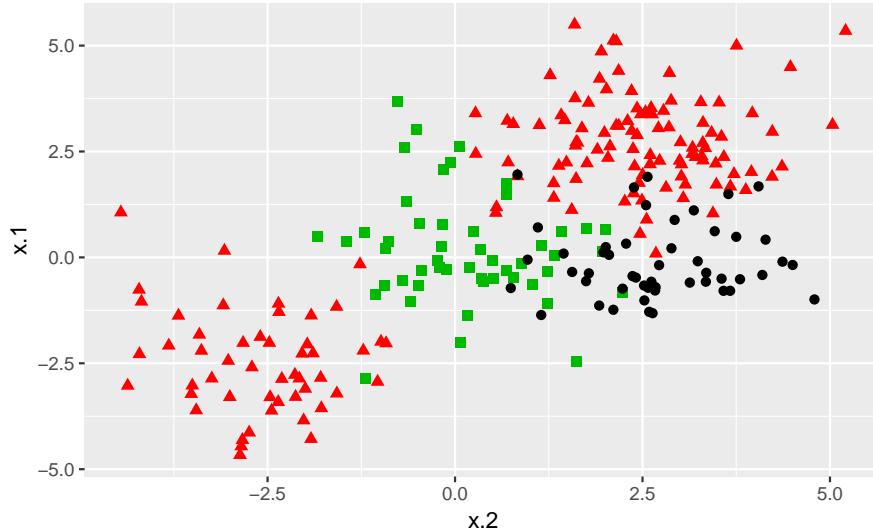
```
# validate model performance
(valid <- table(true = dat[-train, "y"], pred = predict(tune.out$best.model,
newx = dat[-train,])))
#> pred
#> true 1 2
#>    1 55 28
#>    2 12  5
## pred
## true 1 2
##    1 58 19
##    2 16  7
```

Our best-fitting model produces 65% accuracy in identifying classes. For such a complicated shape of observations, this performed reasonably well. We can challenge this method further by adding additional classes of observations.

9.5 SVMs for Multiple Classes

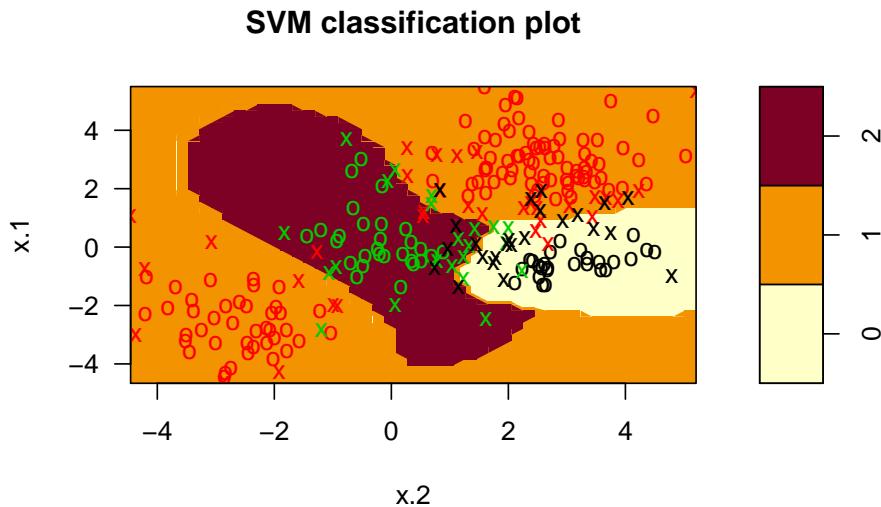
The procedure does not change for data sets that involve more than two classes of observations. We construct our data set the same way as we have previously, only now specifying three classes instead of two:

```
# construct data set
x <- rbind(x, matrix(rnorm(50*2), ncol = 2))
y <- c(y, rep(0, 50))
x[y==0, 2] <- x[y==0, 2] + 2.5
dat <- data.frame(x=x, y=as.factor(y))
# plot data set
ggplot(data = dat, aes(x = x.2, y = x.1, color = y, shape = y)) +
  geom_point(size = 2) +
  scale_color_manual(values=c("#000000", "#FF0000", "#00BA00")) +
  theme(legend.position = "none")
```



The commands don't change for the e1071 package. We specify a cost and tuning parameter γ and fit a support vector machine. The results and interpretation are similar to two-class classification.

```
# fit model
svmfit <- svm(y~., data = dat, kernel = "radial", cost = 10, gamma = 1)
# plot results
plot(svmfit, dat)
```



We can check to see how well our model fit the data by using the `predict()` command, as follows:

```
#construct table
ypred <- predict(svmfit, dat)
(misclass <- table(predict = ypred, truth = dat$y))
##          truth
## predict   0   1   2
##       0 38   2   5
##       1   7 145   2
##       2   5   3  43
##          truth
## predict   0   1   2
##       0 38   2   4
##       1   8 143   4
```

```
##      2   4   5  42
```

As shown in the resulting table, 89% of our training observations were correctly classified. However, since we didn't break our data into training and testing sets, we didn't truly validate our results.

The kernlab package, on the other hand, can fit more than 2 classes, but cannot plot the results. To visualize the results of the ksvm function, we take the steps listed below to create a grid of points, predict the value of each point, and plot the results:

```
# fit and plot
kernfit <- ksvm(as.matrix(dat[,2:1]),dat$y, type = "C-svc", kernel = 'rbfdot',
                 C = 100, scaled = c())

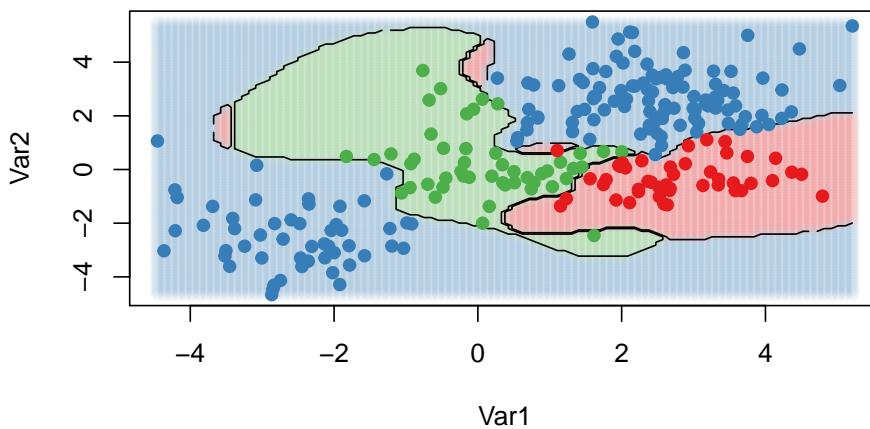
# Create a fine grid of the feature space
x.1 <- seq(from = min(dat$x.1), to = max(dat$x.1), length = 100)
x.2 <- seq(from = min(dat$x.2), to = max(dat$x.2), length = 100)
x.grid <- expand.grid(x.2, x.1)

# Get class predictions over grid
pred <- predict(kernfit, newdata = x.grid)

# Plot the results
cols <- brewer.pal(3, "Set1")
plot(x.grid, pch = 19, col = adjustcolor(cols[pred], alpha.f = 0.05))

classes <- matrix(pred, nrow = 100, ncol = 100)
contour(x = x.2, y = x.1, z = classes, levels = 1:3, labels = "", add = TRUE)

points(dat[, 2:1], pch = 19, col = cols[predict(kernfit)])
```



9.6 Application

The Khan data set contains data on 83 tissue samples with 2308 gene expression measurements on each sample. These were split into 63 training observations and 20 testing observations, and there are four distinct classes in the set. It would be impossible to visualize such data, so we choose the simplest classifier (linear) to construct our model. We will use the svm command from e1071 to conduct our analysis.

```
# fit model
dat <- data.frame(x = Khan$xtrain, y=as.factor(Khan$ytrain))
(out <- svm(y~, data = dat, kernel = "linear", cost=10))
```

```
#>
#> Call:
#> sum(formula = y ~ ., data = dat, kernel = "linear", cost = 10)
#>
#>
#> Parameters:
#>   SVM-Type: C-classification
#>   SVM-Kernel: linear
#>   cost: 10
#>   gamma: 0.000433
#>
#> Number of Support Vectors: 58
```

First of all, we can check how well our model did at classifying the training observations. This is usually high, but again, doesn't validate the model. If the model doesn't do a very good job of classifying the training set, it could be a red flag. In our case, all 63 training observations were correctly classified.

```
# check model performance on training set
table(out$fitted, dat$y)
#>
#>     1  2  3  4
#> 1  8  0  0  0
#> 2  0 23  0  0
#> 3  0  0 12  0
#> 4  0  0  0 20
```

To perform validation, we can check how the model performs on the testing set:

```
# validate model performance
dat.te <- data.frame(x=Khan$xtest, y=as.factor(Khan$ytest))
pred.te <- predict(out, newdata=dat.te)
table(pred.te, dat.te$y)
#>
#> pred.te 1 2 3 4
#>     1 3 0 0 0
#>     2 0 6 2 0
#>     3 0 0 4 0
#>     4 0 0 0 5
```

The model correctly identifies 18 of the 20 testing observations. SVMs and the boundaries they impose are more difficult to interpret at higher dimensions, but these results seem to suggest that our model is a good classifier for the gene data.

Chapter 10

Sonar Standalone Model with Random Forest

Classification problem

10.1 Introduction

- `mtry`: Number of variables randomly sampled as candidates at each split.
- `ntree`: Number of trees to grow.

10.2 Load libraries

```
# load packages
library(caret)
library(mlbench)
library(randomForest)
library(tictoc)

# load dataset
data(Sonar)
set.seed(7)
```

10.3 Explore data

```
dplyr::glimpse(Sonar)
#> Observations: 208
#> Variables: 61
#> $ V1    <dbl> 0.0200, 0.0453, 0.0262, 0.0100, 0.0762, 0.0286, 0.0317, ...
#> $ V2    <dbl> 0.0371, 0.0523, 0.0582, 0.0171, 0.0666, 0.0453, 0.0956, ...
#> $ V3    <dbl> 0.0428, 0.0843, 0.1099, 0.0623, 0.0481, 0.0277, 0.1321, ...
#> $ V4    <dbl> 0.0207, 0.0689, 0.1083, 0.0205, 0.0394, 0.0174, 0.1408, ...
#> $ V5    <dbl> 0.0954, 0.1183, 0.0974, 0.0205, 0.0590, 0.0384, 0.1674, ...
```

```
#> $ V6      <dbl> 0.0986, 0.2583, 0.2280, 0.0368, 0.0649, 0.0990, 0.1710, ...
#> $ V7      <dbl> 0.1539, 0.2156, 0.2431, 0.1098, 0.1209, 0.1201, 0.0731, ...
#> $ V8      <dbl> 0.1601, 0.3481, 0.3771, 0.1276, 0.2467, 0.1833, 0.1401, ...
#> $ V9      <dbl> 0.3109, 0.3337, 0.5598, 0.0598, 0.3564, 0.2105, 0.2083, ...
#> $ V10     <dbl> 0.2111, 0.2872, 0.6194, 0.1264, 0.4459, 0.3039, 0.3513, ...
#> $ V11     <dbl> 0.1609, 0.4918, 0.6333, 0.0881, 0.4152, 0.2988, 0.1786, ...
#> $ V12     <dbl> 0.1582, 0.6552, 0.7060, 0.1992, 0.3952, 0.4250, 0.0658, ...
#> $ V13     <dbl> 0.2238, 0.6919, 0.5544, 0.0184, 0.4256, 0.6343, 0.0513, ...
#> $ V14     <dbl> 0.0645, 0.7797, 0.5320, 0.2261, 0.4135, 0.8198, 0.3752, ...
#> $ V15     <dbl> 0.0660, 0.7464, 0.6479, 0.1729, 0.4528, 1.0000, 0.5419, ...
#> $ V16     <dbl> 0.2273, 0.9444, 0.6931, 0.2131, 0.5326, 0.9988, 0.5440, ...
#> $ V17     <dbl> 0.3100, 1.0000, 0.6759, 0.0693, 0.7306, 0.9508, 0.5150, ...
#> $ V18     <dbl> 0.300, 0.887, 0.755, 0.228, 0.619, 0.902, 0.426, 0.120, ...
#> $ V19     <dbl> 0.508, 0.802, 0.893, 0.406, 0.203, 0.723, 0.202, 0.668, ...
#> $ V20     <dbl> 0.4797, 0.7818, 0.8619, 0.3973, 0.4636, 0.5122, 0.4233, ...
#> $ V21     <dbl> 0.578, 0.521, 0.797, 0.274, 0.415, 0.207, 0.772, 0.783, ...
#> $ V22     <dbl> 0.507, 0.405, 0.674, 0.369, 0.429, 0.399, 0.974, 0.535, ...
#> $ V23     <dbl> 0.433, 0.396, 0.429, 0.556, 0.573, 0.589, 0.939, 0.681, ...
#> $ V24     <dbl> 0.555, 0.391, 0.365, 0.485, 0.540, 0.287, 0.556, 0.917, ...
#> $ V25     <dbl> 0.671, 0.325, 0.533, 0.314, 0.316, 0.204, 0.527, 0.761, ...
#> $ V26     <dbl> 0.641, 0.320, 0.241, 0.533, 0.229, 0.578, 0.683, 0.822, ...
#> $ V27     <dbl> 0.7104, 0.3271, 0.5070, 0.5256, 0.6995, 0.5389, 0.5713, ...
#> $ V28     <dbl> 0.8080, 0.2767, 0.8533, 0.2520, 1.0000, 0.3750, 0.5429, ...
#> $ V29     <dbl> 0.6791, 0.4423, 0.6036, 0.2090, 0.7262, 0.3411, 0.2177, ...
#> $ V30     <dbl> 0.3857, 0.2028, 0.8514, 0.3559, 0.4724, 0.5067, 0.2149, ...
#> $ V31     <dbl> 0.131, 0.379, 0.851, 0.626, 0.510, 0.558, 0.581, 0.132, ...
#> $ V32     <dbl> 0.2604, 0.2947, 0.5045, 0.7340, 0.5459, 0.4778, 0.6323, ...
#> $ V33     <dbl> 0.512, 0.198, 0.186, 0.612, 0.288, 0.330, 0.296, 0.099, ...
#> $ V34     <dbl> 0.7547, 0.2341, 0.2709, 0.3497, 0.0981, 0.2198, 0.1873, ...
#> $ V35     <dbl> 0.8537, 0.1306, 0.4232, 0.3953, 0.1951, 0.1407, 0.2969, ...
#> $ V36     <dbl> 0.851, 0.418, 0.304, 0.301, 0.418, 0.286, 0.516, 0.105, ...
#> $ V37     <dbl> 0.669, 0.384, 0.612, 0.541, 0.460, 0.381, 0.615, 0.192, ...
#> $ V38     <dbl> 0.6097, 0.1057, 0.6756, 0.8814, 0.3217, 0.4158, 0.4283, ...
#> $ V39     <dbl> 0.4943, 0.1840, 0.5375, 0.9857, 0.2828, 0.4054, 0.5479, ...
#> $ V40     <dbl> 0.2744, 0.1970, 0.4719, 0.9167, 0.2430, 0.3296, 0.6133, ...
#> $ V41     <dbl> 0.0510, 0.1674, 0.4647, 0.6121, 0.1979, 0.2707, 0.5017, ...
#> $ V42     <dbl> 0.2834, 0.0583, 0.2587, 0.5006, 0.2444, 0.2650, 0.2377, ...
#> $ V43     <dbl> 0.2825, 0.1401, 0.2129, 0.3210, 0.1847, 0.0723, 0.1957, ...
#> $ V44     <dbl> 0.4256, 0.1628, 0.2222, 0.3202, 0.0841, 0.1238, 0.1749, ...
#> $ V45     <dbl> 0.2641, 0.0621, 0.2111, 0.4295, 0.0692, 0.1192, 0.1304, ...
#> $ V46     <dbl> 0.1386, 0.0203, 0.0176, 0.3654, 0.0528, 0.1089, 0.0597, ...
#> $ V47     <dbl> 0.1051, 0.0530, 0.1348, 0.2655, 0.0357, 0.0623, 0.1124, ...
#> $ V48     <dbl> 0.1343, 0.0742, 0.0744, 0.1576, 0.0085, 0.0494, 0.1047, ...
#> $ V49     <dbl> 0.0383, 0.0409, 0.0130, 0.0681, 0.0230, 0.0264, 0.0507, ...
#> $ V50     <dbl> 0.0324, 0.0061, 0.0106, 0.0294, 0.0046, 0.0081, 0.0159, ...
#> $ V51     <dbl> 0.0232, 0.0125, 0.0033, 0.0241, 0.0156, 0.0104, 0.0195, ...
#> $ V52     <dbl> 0.0027, 0.0084, 0.0232, 0.0121, 0.0031, 0.0045, 0.0201, ...
#> $ V53     <dbl> 0.0065, 0.0089, 0.0166, 0.0036, 0.0054, 0.0014, 0.0248, ...
#> $ V54     <dbl> 0.0159, 0.0048, 0.0095, 0.0150, 0.0105, 0.0038, 0.0131, ...
#> $ V55     <dbl> 0.0072, 0.0094, 0.0180, 0.0085, 0.0110, 0.0013, 0.0070, ...
#> $ V56     <dbl> 0.0167, 0.0191, 0.0244, 0.0073, 0.0015, 0.0089, 0.0138, ...
#> $ V57     <dbl> 0.0180, 0.0140, 0.0316, 0.0050, 0.0072, 0.0057, 0.0092, ...
#> $ V58     <dbl> 0.0084, 0.0049, 0.0164, 0.0044, 0.0048, 0.0027, 0.0143, ...
```

```
#> $ V59 <dbl> 0.0090, 0.0052, 0.0095, 0.0040, 0.0107, 0.0051, 0.0036, ...
#> $ V60 <dbl> 0.0032, 0.0044, 0.0078, 0.0117, 0.0094, 0.0062, 0.0103, ...
#> $ Class <fct> R, ...
```

```
tibble::as_tibble(Sonar)
#> # A tibble: 208 x 61
#>   V1     V2     V3     V4     V5     V6     V7     V8     V9     V10    V11
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1 0.02  0.0371 0.0428 0.0207 0.0954 0.0986 0.154  0.160  0.311  0.211  0.161
#> 2 0.0453 0.0523 0.0843 0.0689 0.118  0.258  0.216  0.348  0.334  0.287  0.492
#> 3 0.0262 0.0582 0.110  0.108  0.0974 0.228  0.243  0.377  0.560  0.619  0.633
#> 4 0.01   0.0171 0.0623 0.0205 0.0205 0.0368 0.110  0.128  0.0598 0.126  0.0881
#> 5 0.0762 0.0666 0.0481 0.0394 0.059  0.0649 0.121  0.247  0.356  0.446  0.415
#> 6 0.0286 0.0453 0.0277 0.0174 0.0384 0.099  0.120  0.183  0.210  0.304  0.299
#> # ... with 202 more rows, and 50 more variables: V12 <dbl>, V13 <dbl>,
#> #   V14 <dbl>, V15 <dbl>, V16 <dbl>, V17 <dbl>, V18 <dbl>, V19 <dbl>,
#> #   V20 <dbl>, V21 <dbl>, V22 <dbl>, V23 <dbl>, V24 <dbl>, V25 <dbl>,
#> #   V26 <dbl>, V27 <dbl>, V28 <dbl>, V29 <dbl>, V30 <dbl>, V31 <dbl>,
#> #   V32 <dbl>, V33 <dbl>, V34 <dbl>, V35 <dbl>, V36 <dbl>, V37 <dbl>,
#> #   V38 <dbl>, V39 <dbl>, V40 <dbl>, V41 <dbl>, V42 <dbl>, V43 <dbl>,
#> #   V44 <dbl>, V45 <dbl>, V46 <dbl>, V47 <dbl>, V48 <dbl>, V49 <dbl>,
#> #   V50 <dbl>, V51 <dbl>, V52 <dbl>, V53 <dbl>, V54 <dbl>, V55 <dbl>,
#> #   V56 <dbl>, V57 <dbl>, V58 <dbl>, V59 <dbl>, V60 <dbl>, Class <fct>
```

```
# create 80%/20% for training and validation datasets
validationIndex <- createDataPartition(Sonar$Class, p=0.80, list=FALSE)
validation <- Sonar[-validationIndex,]
training <- Sonar[validationIndex,]
```

```
tic()
# train a model and summarize model
set.seed(7)
trainControl <- trainControl(method="repeatedcv", number=10, repeats=3)
fit.rf <- train(Class~., data=training,
                 method = "rf",
                 metric = "Accuracy",
                 trControl = trainControl,
                 ntree = 2000)
toc()
#> 61.175 sec elapsed
print(fit.rf)
#> Random Forest
#>
#> 167 samples
#> 60 predictor
#> 2 classes: 'M', 'R'
#>
#> No pre-processing
#> Resampling: Cross-Validated (10 fold, repeated 3 times)
#> Summary of sample sizes: 150, 150, 150, 151, 151, 150, ...
#> Resampling results across tuning parameters:
#>
#>   mtry  Accuracy  Kappa
#>   2      0.845    0.682
```

```

#>   31    0.828    0.651
#>   60    0.808    0.611
#>
#> Accuracy was used to select the optimal model using the largest value.
#> The final value used for the model was mtry = 2.
print(fit.rf$finalModel)
#>
#> Call:
#>   randomForest(x = x, y = y, ntree = 2000, mtry = param$mtry)
#>           Type of random forest: classification
#>           Number of trees: 2000
#> No. of variables tried at each split: 2
#>
#>       OOB estimate of error rate: 14.4%
#> Confusion matrix:
#>   M  R class.error
#> M 84 5      0.0562
#> R 19 59     0.2436

```

Accuracy: 85.26% at mtry=2

10.4 Apply tuning parameters for final model

```

# create standalone model using all training data
set.seed(7)
finalModel <- randomForest(Class~, training, mtry=2, ntree=2000)

# make a predictions on "new data" using the final model
finalPredictions <- predict(finalModel, validation[,1:60])
confusionMatrix(finalPredictions, validation$Class)
#> Confusion Matrix and Statistics
#>
#>             Reference
#> Prediction M  R
#>           M 20 4
#>           R  2 15
#>
#>       Accuracy : 0.854
#>           95% CI : (0.708, 0.944)
#>   No Information Rate : 0.537
#> P-Value [Acc > NIR] : 1.88e-05
#>
#>           Kappa : 0.704
#>
#> Mcnemar's Test P-Value : 0.683
#>
#> Sensitivity : 0.909
#> Specificity : 0.789
#> Pos Pred Value : 0.833
#> Neg Pred Value : 0.882
#> Prevalence : 0.537
#> Detection Rate : 0.488

```

```
#>      Detection Prevalence : 0.585
#>      Balanced Accuracy : 0.849
#>
#>      'Positive' Class : M
#>
```

Accuracy: 82.93%

10.5 Save model

```
# save the model to disk
saveRDS(finalModel, file.path(model_out_dir, "sonar-finalModel.rds"))
```

10.6 Use the saved model

```
# load the model
superModel <- readRDS(file.path(model_out_dir, "sonar-finalModel.rds"))
print(superModel)
#>
#> Call:
#>   randomForest(formula = Class ~ ., data = training, mtry = 2,      ntree = 2000)
#>           Type of random forest: classification
#>           Number of trees: 2000
#> No. of variables tried at each split: 2
#>
#>           OOB estimate of error rate: 16.2%
#> Confusion matrix:
#>   M  R class.error
#> M 81  8    0.0899
#> R 19  59   0.2436
```

10.7 Make prediction with new data

```
# make a predictions on "new data" using the final model
finalPredictions <- predict(superModel, validation[,1:60])
confusionMatrix(finalPredictions, validation$Class)
#> Confusion Matrix and Statistics
#>
#>           Reference
#> Prediction  M  R
#>           M 20  4
#>           R  2  15
#>
#>           Accuracy : 0.854
#>             95% CI : (0.708, 0.944)
#>   No Information Rate : 0.537
#>   P-Value [Acc > NIR] : 1.88e-05
```

```
#>                               Kappa : 0.704
#>
#> Mcnemar's Test P-Value : 0.683
#>
#>                               Sensitivity : 0.909
#>                               Specificity : 0.789
#> Pos Pred Value : 0.833
#> Neg Pred Value : 0.882
#> Prevalence : 0.537
#> Detection Rate : 0.488
#> Detection Prevalence : 0.585
#> Balanced Accuracy : 0.849
#>
#> 'Positive' Class : M
#>
```

Chapter 11

Glass classification

<https://cran.r-project.org/web/packages/e1071/vignettes/svmdoc.pdf>

In this example, we use the glass data from the UCI Repository of Machine Learning Databases for classification. The task is to predict the type of a glass on basis of its chemical analysis. We start by splitting the data into a train and test set:

```
library(caret)
#> Loading required package: lattice
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures    rlang
#>   c.quosures    rlang
#>   print.quosures rlang
library(e1071)
library(rpart)

data(Glass, package="mlbench")
str(Glass)
#> 'data.frame': 214 obs. of 10 variables:
#> $ RI : num 1.52 1.52 1.52 1.52 1.52 ...
#> $ Na : num 13.6 13.9 13.5 13.2 13.3 ...
#> $ Mg : num 4.49 3.6 3.55 3.69 3.62 3.61 3.6 3.61 3.58 3.6 ...
#> $ Al : num 1.1 1.36 1.54 1.29 1.24 1.62 1.14 1.05 1.37 1.36 ...
#> $ Si : num 71.8 72.7 73 72.6 73.1 ...
#> $ K : num 0.06 0.48 0.39 0.57 0.55 0.64 0.58 0.57 0.56 0.57 ...
#> $ Ca : num 8.75 7.83 7.78 8.22 8.07 8.07 8.17 8.24 8.3 8.4 ...
#> $ Ba : num 0 0 0 0 0 0 0 0 0 0 ...
#> $ Fe : num 0 0 0 0 0 0.26 0 0 0 0.11 ...
#> $ Type: Factor w/ 6 levels "1","2","3","5",...: 1 1 1 1 1 1 1 1 1 1 ...

## split data into a train and test set
index <- 1:nrow(Glass)
testindex <- sample(index, trunc(length(index)/3))
testset <- Glass[testindex,]
trainset <- Glass[-testindex,]
```

Both for the SVM and the partitioning tree (via `rpart()`), we fit the model and try to predict the test set values:

```
## svm
svm.model <- svm(Type ~ ., data = trainset, cost = 100, gamma = 1)
svm.pred <- predict(svm.model, testset[,-10])
```

(The dependent variable, Type, has column number 10. cost is a general penalizing parameter for C-classification and gamma is the radial basis function-specific kernel parameter.)

```
## rpart
rpart.model <- rpart(Type ~ ., data = trainset)
rpart.pred <- predict(rpart.model, testset[,-10], type = "class")
```

A cross-tabulation of the true versus the predicted values yields:

```
## compute svm confusion matrix
table(pred = svm.pred, true = testset[,10])
#>      true
#> pred  1  2  3  5  6  7
#>   1 20  3  3  0  0  0
#>   2  6 13  5  4  2  4
#>   3  2  1  0  0  0  0
#>   5  0  0  0  1  0  0
#>   6  0  0  0  0  0  0
#>   7  0  0  0  0  0  7

## compute rpart confusion matrix
table(pred = rpart.pred, true = testset[,10])
#>      true
#> pred  1  2  3  5  6  7
#>   1 22  0  3  0  0  0
#>   2  5 12  4  0  0  0
#>   3  0  2  1  0  0  0
#>   5  0  2  0  5  2  1
#>   6  0  0  0  0  0  0
#>   7  1  1  0  0  0 10
```

11.0.1 Comparison test sets

```
confusionMatrix(svm.pred, testset$Type)
#> Confusion Matrix and Statistics
#>
#>      Reference
#> Prediction  1  2  3  5  6  7
#>   1 20  3  3  0  0  0
#>   2  6 13  5  4  2  4
#>   3  2  1  0  0  0  0
#>   5  0  0  0  1  0  0
#>   6  0  0  0  0  0  0
#>   7  0  0  0  0  0  7
#>
#> Overall Statistics
#>
#>     Accuracy : 0.577
#>     95% CI : (0.454, 0.694)
#>     No Information Rate : 0.394
```

```

#>      P-Value [Acc > NIR] : 0.00137
#>
#>      Kappa : 0.413
#>
#> McNemar's Test P-Value : NA
#>
#> Statistics by Class:
#>
#>          Class: 1 Class: 2 Class: 3 Class: 5 Class: 6 Class: 7
#> Sensitivity      0.714    0.765    0.0000   0.2000    0.0000   0.6364
#> Specificity      0.860    0.611    0.9524   1.0000    1.0000   1.0000
#> Pos Pred Value   0.769    0.382    0.0000   1.0000      NaN    1.0000
#> Neg Pred Value   0.822    0.892    0.8824   0.9429   0.9718   0.9375
#> Prevalence        0.394    0.239    0.1127   0.0704   0.0282   0.1549
#> Detection Rate   0.282    0.183    0.0000   0.0141    0.0000   0.0986
#> Detection Prevalence  0.366    0.479    0.0423   0.0141    0.0000   0.0986
#> Balanced Accuracy 0.787    0.688    0.4762   0.6000   0.5000   0.8182

confusionMatrix(rpart.pred, testset$type)
#> Confusion Matrix and Statistics
#>
#>      Reference
#> Prediction 1 2 3 5 6 7
#>       1 22 0 3 0 0 0
#>       2 5 12 4 0 0 0
#>       3 0 2 1 0 0 0
#>       5 0 2 0 5 2 1
#>       6 0 0 0 0 0 0
#>       7 1 1 0 0 0 10
#>
#> Overall Statistics
#>
#>      Accuracy : 0.704
#>      95% CI : (0.584, 0.807)
#>      No Information Rate : 0.394
#>      P-Value [Acc > NIR] : 1.23e-07
#>
#>      Kappa : 0.605
#>
#> McNemar's Test P-Value : NA
#>
#> Statistics by Class:
#>
#>          Class: 1 Class: 2 Class: 3 Class: 5 Class: 6 Class: 7
#> Sensitivity      0.786    0.706    0.1250   1.0000    0.0000   0.909
#> Specificity      0.930    0.833    0.9683   0.9242    1.0000   0.967
#> Pos Pred Value   0.880    0.571    0.3333   0.5000      NaN   0.833
#> Neg Pred Value   0.870    0.900    0.8971   1.0000   0.9718   0.983
#> Prevalence        0.394    0.239    0.1127   0.0704   0.0282   0.155
#> Detection Rate   0.310    0.169    0.0141   0.0704   0.0000   0.141
#> Detection Prevalence  0.352    0.296    0.0423   0.1408   0.0000   0.169
#> Balanced Accuracy 0.858    0.770    0.5466   0.9621   0.5000   0.938

```

11.0.2 Comparison with resamples

Finally, we compare the performance of the two methods by computing the respective accuracy rates and the kappa indices (as computed by `classAgreement()` also contained in package `e1071`). In Table 1, we summarize the results of 10 replications—Support Vector Machines show better results.

```
set.seed(1234567)

# SVM
fit.svm <- train(Type ~ ., data = trainset,
                   method = "svmRadial")

# Random Forest
fit.rpart <- train(Type ~ ., data = trainset,
                     method="rpart")

# collect resamples
results <- resamples(list(svm = fit.svm,
                           rpart = fit.rpart))

summary(results)
#>
#> Call:
#> summary.resamples(object = results)
#>
#> Models: sum, rpart
#> Number of resamples: 25
#>
#> Accuracy
#>      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> sum    0.510  0.565  0.600 0.599   0.625 0.704    0
#> rpart  0.462  0.519  0.554 0.558   0.600 0.660    0
#>
#> Kappa
#>      Min. 1st Qu. Median Mean 3rd Qu. Max. NA's
#> sum    0.267  0.376  0.406 0.410   0.446 0.559    0
#> rpart  0.135  0.299  0.358 0.363   0.443 0.545    0
```

Chapter 12

Ozone SVM

<https://cran.r-project.org/web/packages/e1071/vignettes/svmdoc.pdf>

```
library(e1071)
library(rpart)

data(Ozone, package="mlbench")
## split data into a train and test set
index <- 1:nrow(Ozone)
testindex <- sample(index, trunc(length(index)/3))
testset <- na.omit(Ozone[testindex,-3])
trainset <- na.omit(Ozone[-testindex,-3])

## svm
svm.model <- svm(V4 ~ ., data = trainset, cost = 1000, gamma = 0.0001)
svm.pred <- predict(svm.model, testset[,-3])
crossprod(svm.pred - testset[,3]) / length(testindex)
#>      [,1]
#> [1,] 10.7

## rpart
rpart.model <- rpart(V4 ~ ., data = trainset)
rpart.pred <- predict(rpart.model, testset[,-3])
crossprod(rpart.pred - testset[,3]) / length(testindex)
#>      [,1]
#> [1,] 11.9
```


Chapter 13

A gentle introduction to support vector machines using R

<https://eight2late.wordpress.com/2017/02/07/a-gentle-introduction-to-support-vector-machines-using-r/>

13.1 Support vector machines in R

In this demo we'll use the `svm` interface that is implemented in the `e1071` R package. This interface provides R programmers access to the comprehensive `libsvm` library written by Chang and Lin. I'll use two toy datasets: the famous `iris` dataset available with the base R package and the `sonar` dataset from the `mlbench` package. I won't describe details of the datasets as they are discussed at length in the documentation that I have linked to. However, it is worth mentioning the reasons why I chose these datasets:

As mentioned earlier, no real life dataset is linearly separable, but the `iris` dataset is almost so. Consequently, it is a good illustration of using linear SVMs. Although one almost never uses these in practice, I have illustrated their use primarily for pedagogical reasons. The `sonar` dataset is a good illustration of the benefits of using RBF kernels in cases where the dataset is hard to visualise (60 variables in this case!). In general, one would almost always use RBF (or other nonlinear) kernels in practice.

With that said, let's get right to it. I assume you have R and RStudio installed. For instructions on how to do this, have a look at the first article in this series. The processing preliminaries – loading libraries, data and creating training and test datasets are much the same as in my previous articles so I won't dwell on these here. For completeness, however, I'll list all the code so you can run it directly in R or R studio (a complete listing of the code can be found [here](#)):

13.2 SVM on `iris` dataset

13.2.1 Training and test datasets

```
#load required library
library(e1071)

#load built-in iris dataset
data(iris)
```

```
#set seed to ensure reproducible results
set.seed(42)

#split into training and test sets
iris[, "train"] <- ifelse(runif(nrow(iris)) < 0.8, 1, 0)

#separate training and test sets
trainset <- iris[iris$train == 1,]
testset <- iris[iris$train == 0,]

#get column index of train flag
trainColNum <- grep("train", names(trainset))

#remove train flag column from train and test sets
trainset <- trainset[,-trainColNum]
testset <- testset[,-trainColNum]

dim(trainset)
#> [1] 115   5
dim(testset)
#> [1] 35   5
```

13.2.2 Build the SVM model

```
#get column index of predicted variable in dataset
typeColNum <- grep("Species", names(iris))

#build model - linear kernel and C-classification (soft margin) with default cost (C=1)
svm_model <- svm(Species ~ ., data = trainset,
                  method = "C-classification",
                  kernel = "linear")
svm_model
#>
#> Call:
#> sum(formula = Species ~ ., data = trainset, method = "C-classification",
#>       kernel = "linear")
#>
#>
#> Parameters:
#>   SVM-Type:  C-classification
#>   SVM-Kernel: linear
#>     cost:  1
#>     gamma:  0.25
#>
#> Number of Support Vectors:  24
```

The output from the SVM model show that there are 24 support vectors. If desired, these can be examined using the SV variable in the model – i.e via `svm_model$SV`.

13.2.3 Support Vectors

```
# support vectors
svm_model$SV
#> Sepal.Length Sepal.Width Petal.Length Petal.Width
#> 19 -0.2564 1.7668 -1.323 -1.305
#> 42 -1.7006 -1.7045 -1.559 -1.305
#> 45 -0.9785 1.7668 -1.205 -1.171
#> 53 1.1878 0.1469 0.568 0.309
#> 55 0.7064 -0.5474 0.390 0.309
#> 57 0.4657 0.6097 0.450 0.443
#> 58 -1.2192 -1.4730 -0.378 -0.364
#> 69 0.3453 -1.9359 0.331 0.309
#> 71 -0.0157 0.3783 0.509 0.712
#> 73 0.4657 -1.2416 0.568 0.309
#> 78 0.9471 -0.0845 0.627 0.578
#> 84 0.1046 -0.7788 0.686 0.443
#> 85 -0.6174 -0.0845 0.331 0.309
#> 86 0.1046 0.8412 0.331 0.443
#> 99 -0.9785 -1.2416 -0.555 -0.229
#> 107 -1.2192 -1.2416 0.331 0.578
#> 111 0.7064 0.3783 0.686 0.981
#> 117 0.7064 -0.0845 0.922 0.712
#> 124 0.4657 -0.7788 0.568 0.712
#> 130 1.5488 -0.0845 1.099 0.443
#> 138 0.5860 0.1469 0.922 0.712
#> 139 0.1046 -0.0845 0.509 0.712
#> 147 0.4657 -1.2416 0.627 0.847
#> 150 -0.0157 -0.0845 0.686 0.712
```

The test prediction accuracy indicates that the linear performs quite well on this dataset, confirming that it is indeed near linearly separable. To check performance by class, one can create a confusion matrix as described in my post on random forests. I'll leave this as an exercise for you. Another point is that we have used a soft-margin classification scheme with a cost C=1. You can experiment with this by explicitly changing the value of C. Again, I'll leave this for you an exercise.

13.2.4 Predictions on training model

```
# training set predictions
pred_train <- predict(svm_model, trainset)
mean(pred_train == trainset$Species)
#> [1] 0.983
# [1] 0.9826087
```

13.2.5 Predictions on test model

```
# test set predictions
pred_test <- predict(svm_model, testset)
mean(pred_test == testset$Species)
#> [1] 0.914
# [1] 0.9142857
```

13.2.6 Confusion matrix and Accuracy

```
# confusion matrix
cm <- table(pred_test, testset$Species)
cm
#>
#> pred_test      setosa versicolor virginica
#>   setosa        18       0       0
#>   versicolor     0       5       3
#>   virginica     0       0       9

# accuracy
sum(diag(cm)) / sum(cm)
#> [1] 0.914
```

13.3 SVM with Radial Basis Function kernel. Linear

13.3.1 Training and test sets

```
#load required library (assuming e1071 is already loaded)
library(mlbench)

#load Sonar dataset
data(Sonar)
#set seed to ensure reproducible results
set.seed(42)
#split into training and test sets
Sonar[, "train"] <- ifelse(runif(nrow(Sonar)) < 0.8, 1, 0)

#separate training and test sets
trainset <- Sonar[Sonar$train == 1,]
testset <- Sonar[Sonar$train == 0,]

#get column index of train flag
trainColNum <- grep("train", names(trainset))
#remove train flag column from train and test sets
trainset <- trainset[,-trainColNum]
testset <- testset[,-trainColNum]

#get column index of predicted variable in dataset
typeColNum <- grep("Class", names(Sonar))
```

13.3.2 Predictions on Training model

```
#build model - linear kernel and C-classification with default cost (C=1)
svm_model <- svm(Class ~ ., data=trainset,
                  method="C-classification",
                  kernel="linear")
```

```
#training set predictions
pred_train <-predict(svm_model,trainset)
mean(pred_train==trainset$Class)
#> [1] 0.97
```

13.3.3 Predictions on test model

```
#test set predictions
pred_test <-predict(svm_model,testset)
mean(pred_test==testset$Class)
#> [1] 0.605
```

I'll leave you to examine the contents of the model. The important point to note here is that the performance of the model with the test set is quite dismal compared to the previous case. This simply indicates that the linear kernel is not appropriate here. Let's take a look at what happens if we use the RBF kernel with default values for the parameters:

13.4 SVM with Radial Basis Function kernel. Non-linear

13.4.1 Predictions on training model

```
#build model: radial kernel, default params
svm_model <- svm(Class~ ., data=trainset,
                  method="C-classification",
                  kernel="radial")

# print params
svm_model$cost
#> [1] 1
svm_model$gamma
#> [1] 0.0167

#training set predictions
pred_train <-predict(svm_model,trainset)
mean(pred_train==trainset$Class)
#> [1] 0.988
```

13.4.2 Predictions on test model

```
#test set predictions
pred_test <-predict(svm_model,testset)
mean(pred_test==testset$Class)
#> [1] 0.767
```

That's a pretty decent improvement from the linear kernel. Let's see if we can do better by doing some parameter tuning. To do this we first invoke tune.svm and use the parameters it gives us in the call to svm:

13.4.3 Tuning of parameters

```
# find optimal parameters in a specified range
tune_out <- tune.svm(x = trainset[, -typeColNum],
                      y = trainset[, typeColNum],
                      gamma = 10^{(-3:3)},
                      cost = c(0.01, 0.1, 1, 10, 100, 1000),
                      kernel = "radial")

#print best values of cost and gamma
tune_out$best.parameters$cost
#> [1] 10
tune_out$best.parameters$gamma
#> [1] 0.01

#build model
svm_model <- svm(Class~ ., data = trainset,
                   method = "C-classification",
                   kernel = "radial",
                   cost = tune_out$best.parameters$cost,
                   gamma = tune_out$best.parameters$gamma)
```

13.4.4 Prediction on training model with new parameters

```
# training set predictions
pred_train <- predict(svm_model, trainset)
mean(pred_train==trainset$Class)
#> [1] 1
```

13.4.5 Prediction on test model with new parameters

```
# test set predictions
pred_test <- predict(svm_model, testset)
mean(pred_test==testset$Class)
#> [1] 0.814
```

Which is fairly decent improvement on the un-optimised case.

13.5 Wrapping up

This bring us to the end of this introductory exploration of SVMs in R. To recap, the distinguishing feature of SVMs in contrast to most other techniques is that they attempt to construct optimal separation boundaries between different categories.

SVMs are quite versatile and have been applied to a wide variety of domains ranging from chemistry to pattern recognition. They are best used in binary classification scenarios. This brings up a question as to where SVMs are to be preferred to other binary classification techniques such as logistic regression. The honest response is, “it depends” – but here are some points to keep in mind when choosing between the two. A general point to keep in mind is that SVM algorithms tend to be expensive both in terms of memory and computation, issues that can start to hurt as the size of the dataset increases.

Given all the above caveats and considerations, the best way to figure out whether an SVM approach will work for your problem may be to do what most machine learning practitioners do: try it out!

Chapter 14

SMS spam. Naive Bayes. Classification

Dataset: https://github.com/stedy/Machine-Learning-with-R-datasets/blob/master/sms_spam.csv

Instructions: Machine Learning with R. Page 104.

```
library(tictoc)

sms_raw <- read.csv(file.path(data_raw_dir, "sms_spam.csv"), stringsAsFactors = FALSE)

str(sms_raw)
#> 'data.frame': 5574 obs. of 2 variables:
#> $ type: chr "ham" "ham" "spam" "ham" ...
#> $ text: chr "Go until jurong point, crazy.. Available only in bugis n great world la e buffet... C
```

14.0.1 convert type to a factor

```
sms_raw$type <- factor(sms_raw$type)

str(sms_raw$type)
#> Factor w/ 2 levels "ham", "spam": 1 1 2 1 1 2 1 1 2 2 ...
```

How many email of type ham or spam:

```
table(sms_raw$type)
#>
#> ham spam
#> 4827 747
```

Create the corpus:

```
library(tm)
#> Loading required package: NLP

sms_corpus <- VCorpus(VectorSource(sms_raw$text))
print(sms_corpus)
#> <<VCorpus>>
#> Metadata: corpus specific: 0, document level (indexed): 0
#> Content: documents: 5574
```

Let's see a couple of documents:

```
inspect(sms_corpus[1:2])
#> <<VCorpus>>
#> Metadata: corpus specific: 0, document level (indexed): 0
#> Content: documents: 2
#>
#> [[1]]
#> <<PlainTextDocument>>
#> Metadata: 7
#> Content: chars: 111
#>
#> [[2]]
#> <<PlainTextDocument>>
#> Metadata: 7
#> Content: chars: 29

# show some text
as.character(sms_corpus[[1]])
#> [1] "Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cine there ...

# show three documents
lapply(sms_corpus[1:3], as.character)
#> $`1`
#> [1] "Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cine there ...
#>
#> $`2`
#> [1] "Ok lar... Joking wif u oni..."
#>
#> $`3`
#> [1] "Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to 87121 to receive ..."
```

14.1 Some conversion

```
# convert to lowercase
sms_corpus_clean <- tm_map(sms_corpus, content_transformer(tolower))

as.character(sms_corpus[[1]])
#> [1] "Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cine there ...

# converted to lowercase
as.character(sms_corpus_clean[[1]])
#> [1] "go until jurong point, crazy.. available only in bugis n great world la e buffet... cine there ...

# remove numbers
sms_corpus_clean <- tm_map(sms_corpus_clean, removeNumbers)
```

What transformations are available

```
# what transformations are available
getTransformations()
#> [1] "removeNumbers"      "removePunctuation" "removeWords"
#> [4] "stemDocument"       "stripWhitespace"
```

```
# remove stop words
sms_corpus_clean <- tm_map(sms_corpus_clean, removeWords, stopwords())

# remove punctuation
sms_corpus_clean <- tm_map(sms_corpus_clean, removePunctuation)
```

Stemming:

```
library(SnowballC)
wordStem(c("learn", "learned", "learning", "learns"))
#> [1] "learn" "learn" "learn" "learn"

# stemming corpus
sms_corpus_clean <- tm_map(sms_corpus_clean, stemDocument)

# remove white spaces
sms_corpus_clean <- tm_map(sms_corpus_clean, stripWhitespace)
```

Show what we've got so far

```
# show what we've got so far
lapply(sms_corpus[1:3], as.character)
#> $`1`
#> [1] "Go until jurong point, crazy.. Available only in bugis n great world la e buffet... Cine there...
#>
#> $`2`
#> [1] "Ok lar... Joking wif u oni..."
#>
#> $`3`
#> [1] "Free entry in 2 a wkly comp to win FA Cup final tkts 21st May 2005. Text FA to 87121 to receive...

lapply(sms_corpus_clean[1:3], as.character)
#> $`1`
#> [1] "go jurong point crazi avail bugi n great world la e buffet cine got amor wat"
#>
#> $`2`
#> [1] "ok lar joke wif u oni"
#>
#> $`3`
#> [1] "free entri wkli comp win fa cup final tkts st may text fa receiv entri questionstd txt ratetc a...
```

14.2 Convert to Document Term Matrix (dtm)

```
)  
sms_dtm <- DocumentTermMatrix(sms_corpus_clean)  
sms_dtm  
#> <<DocumentTermMatrix (documents: 5574, terms: 6592)>>  
#> Non-/sparse entries: 42608/36701200  
#> Sparsity : 100%  
#> Maximal term length: 40  
#> Weighting : term frequency (tf)
```

14.3 split in training and test datasets

```
sms_dtm_train <- sms_dtm[1:4169, ]
sms_dtm_test  <- sms_dtm[4170:5559, ]
```

14.3.1 separate the labels

```
sms_train_labels <- sms_raw[1:4169, ]$type
sms_test_labels  <- sms_raw[4170:5559, ]$type

prop.table(table(sms_train_labels))
#> sms_train_labels
#>   ham  spam
#> 0.865 0.135

prop.table(table(sms_test_labels))
#> sms_test_labels
#>   ham  spam
#> 0.87 0.13

# convert dtm to matrix
sms_mat_train <- as.matrix(t(sms_dtm_train))
dtm.rs <- sort(rowSums(sms_mat_train), decreasing=TRUE)

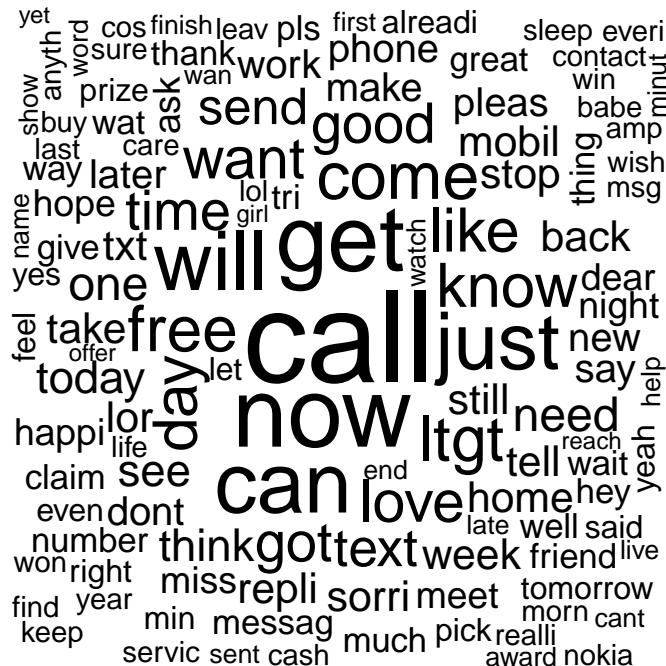
# dataframe with word-frequency
dtm.df <- data.frame(word = names(dtm.rs), freq = as.integer(dtm.rs),
                      stringsAsFactors = FALSE)
```

14.4 plot wordcloud

```
library(wordcloud)
#> Loading required package: RColorBrewer
wordcloud(sms_corpus_clean, min.freq = 50, random.order = FALSE)
#> Warning in wordcloud(sms_corpus_clean, min.freq = 50, random.order =
#> FALSE): tone could not be fit on page. It will not be plotted.
#> Warning in wordcloud(sms_corpus_clean, min.freq = 50, random.order =
#> FALSE): also could not be fit on page. It will not be plotted.
#> Warning in wordcloud(sms_corpus_clean, min.freq = 50, random.order =
#> FALSE): look could not be fit on page. It will not be plotted.
#> Warning in wordcloud(sms_corpus_clean, min.freq = 50, random.order =
#> FALSE): start could not be fit on page. It will not be plotted.
#> Warning in wordcloud(sms_corpus_clean, min.freq = 50, random.order =
#> FALSE): smile could not be fit on page. It will not be plotted.
#> Warning in wordcloud(sms_corpus_clean, min.freq = 50, random.order =
#> FALSE): urgent could not be fit on page. It will not be plotted.
#> Warning in wordcloud(sms_corpus_clean, min.freq = 50, random.order =
#> FALSE): use could not be fit on page. It will not be plotted.
#> Warning in wordcloud(sms_corpus_clean, min.freq = 50, random.order =
#> FALSE): someth could not be fit on page. It will not be plotted.
#> Warning in wordcloud(sms_corpus_clean, min.freq = 50, random.order =
```



```
#> Warning in wordcloud(sms_corpus_clean, min.freq = 50, random.order =
#> FALSE): shop could not be fit on page. It will not be plotted.
#> Warning in wordcloud(sms_corpus_clean, min.freq = 50, random.order =
#> FALSE): hello could not be fit on page. It will not be plotted.
#> Warning in wordcloud(sms_corpus_clean, min.freq = 50, random.order =
#> FALSE): hour could not be fit on page. It will not be plotted.
#> Warning in wordcloud(sms_corpus_clean, min.freq = 50, random.order =
#> FALSE): mean could not be fit on page. It will not be plotted.
#> Warning in wordcloud(sms_corpus_clean, min.freq = 50, random.order =
#> FALSE): month could not be fit on page. It will not be plotted.
#> Warning in wordcloud(sms_corpus_clean, min.freq = 50, random.order =
#> FALSE): guarante could not be fit on page. It will not be plotted.
#> Warning in wordcloud(sms_corpus_clean, min.freq = 50, random.order =
#> FALSE): peopl could not be fit on page. It will not be plotted.
#> Warning in wordcloud(sms_corpus_clean, min.freq = 50, random.order =
#> FALSE): happen could not be fit on page. It will not be plotted.
#> Warning in wordcloud(sms_corpus_clean, min.freq = 50, random.order =
#> FALSE): thk could not be fit on page. It will not be plotted.
```



```
spam <- subset(sms_raw, type == "spam")
ham <- subset(sms_raw, type == "ham")
```

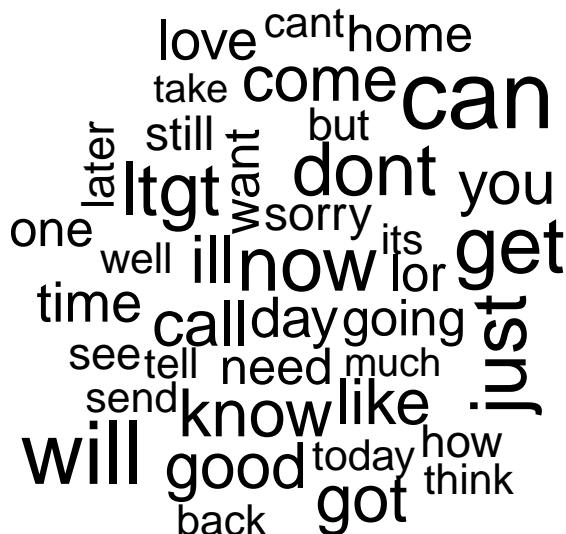
Words related to **spam**

```
wordcloud(spam$text, max.words = 40, scale = c(3, 0.5))
#> Warning in tm_map.SimpleCorpus(corpus, tm::removePunctuation):
#> transformation drops documents
#> Warning in tm_map.SimpleCorpus(corpus, function(x) tm::removeWords(x,
#> tm::stopwords())): transformation drops documents
```



Words related to ham

```
wordcloud(ham$text, max.words = 40, scale = c(3, 0.5))
#> Warning in tm_map.SimpleCorpus(corpus, tm::removePunctuation):
#> transformation drops documents
#> Warning in tm_map.SimpleCorpus(corpus, function(x) tm::removeWords(x,
#> tm::stopwords())): transformation drops documents
```



14.5 Limit Frequent words

```
# words that appear at least in 5 messages
sms_freq_words <- findFreqTerms(sms_dtm_train, 6)

str(sms_freq_words)
#> chr [1:997] "abiola" "abl" "abt" "accept" "access" "account" "across" ...
```

14.5.1 get only frequent words

```
sms_dtm_freq_train<- sms_dtm_train[ , sms_freq_words]
sms_dtm_freq_test <- sms_dtm_test[ , sms_freq_words]
```

14.5.2 function to change value to Yes/No

```

convert_counts <- function(x) {
  x <- ifelse(x > 0, "Yes", "No")
}

# change from number to Yes/No
# also the result returns a matrix
sms_train <- apply(sms_dtm_freq_train, MARGIN = 2,
                    convert_counts)
sms_test <- apply(sms_dtm_freq_test, MARGIN = 2,
                   convert_counts)

# matrix of
# 4169 documents as rows
# 1159 terms as columns
dim(sms_train)
#> [1] 4169 997
length(sms_train_labels)
#> [1] 4169

# this is how the matrix looks
sms_train[1:10, 10:15]
#>      Terms
#> Docs add address admir advanc aft afternoon
#> 1 "No" "No" "No" "No" "No" "No"
#> 2 "No" "No" "No" "No" "No" "No"
#> 3 "No" "No" "No" "No" "No" "No"
#> 4 "No" "No" "No" "No" "No" "No"
#> 5 "No" "No" "No" "No" "No" "No"
#> 6 "No" "No" "No" "No" "No" "No"
#> 7 "No" "No" "No" "No" "No" "No"
#> 8 "No" "No" "No" "No" "No" "No"
#> 9 "No" "No" "No" "No" "No" "No"
#> 10 "No" "No" "No" "No" "No" "No"

library(e1071)
sms_classifier <- naiveBayes(sms_train, sms_train_labels)

tic()
sms_test_pred <- predict(sms_classifier, sms_test)
toc()
#> 20.665 sec elapsed

library(gmodels)
CrossTable(sms_test_pred, sms_test_labels,
           prop.chisq = FALSE, prop.t = FALSE,
           dnn = c('predicted', 'actual'))
#>
#>
#>      Cell Contents
#> /-----/
#> |                               N   |
#> |                               N / Row Total   |
#> |                               N / Col Total   |

```

```
#> /-----/
#>
#>
#> Total Observations in Table: 1390
#>
#>
#>           / actual
#>   predicted /      ham /      spam / Row Total /
#> -----/-----/-----/-----/
#>   ham /    1202 /      21 /    1223 /
#>   /    0.983 /    0.017 /    0.880 /
#>   /    0.994 /    0.116 /      /
#> -----/-----/-----/-----/
#>   spam /      7 /    160 /    167 /
#>   /    0.042 /    0.958 /    0.120 /
#>   /    0.006 /    0.884 /      /
#> -----/-----/-----/-----/
#> Column Total /    1209 /    181 /    1390 /
#>   /    0.870 /    0.130 /      /
#> -----/-----/-----/-----/
#>
#>
```

Misclassified: 20+9 (frequency = 5) 25+7 (freq=4) 23+7 (freq=3) 25+8 (freq=2) 21+7 (freq=6)

Decreasing the minimum word frequency doesn't make the model better.

14.6 Improve model performance

```
sms_classifier2 <- naiveBayes(sms_train, sms_train_labels,
                                laplace = 1)
```

```
tic()
sms_test_pred2 <- predict(sms_classifier2, sms_test)
toc()
#> 20.166 sec elapsed
```

```
CrossTable(sms_test_pred2, sms_test_labels,
            prop.chisq = FALSE, prop.t = FALSE, prop.r = FALSE,
            dnn = c('predicted', 'actual'))
#>
#>
#>   Cell Contents
#> /-----/-----/
#> /           N /           /
#> /     N / Col Total /           /
#> /-----/-----/
#>
#>
#> Total Observations in Table: 1390
#>
#>
#>           / actual
```

```
#>   predicted /      ham /      spam / Row Total /
#>   -----/-----/-----/-----/
#>     ham /    1203 /      28 /    1231 /
#>     /    0.995 /  0.155 /      /
#>   -----/-----/-----/-----/
#>     spam /       6 /    153 /    159 /
#>     /  0.005 /  0.845 /      /
#>   -----/-----/-----/-----/
#> Column Total /    1209 /    181 /    1390 /
#>     /  0.870 /  0.130 /      /
#>   -----/-----/-----/-----/
#>
#>
```

Misclassified: 28+7

Chapter 15

Classification Tree: Vehicle example

- Dataset: Vehicle (mlbench)
- Instructions: book “Applied Predictive Modeling Techniques”, Lewis, N.D.

15.1 Load packages

```
library(tree)
library(mlbench)

data(Vehicle)
str(Vehicle)
#> 'data.frame': 846 obs. of 19 variables:
#> $ Comp      : num  95 91 104 93 85 107 97 90 86 93 ...
#> $ Circ      : num  48 41 50 41 44 57 43 43 34 44 ...
#> $ D.Circ    : num  83 84 106 82 70 106 73 66 62 98 ...
#> $ Rad.Ra    : num  178 141 209 159 205 172 173 157 140 197 ...
#> $ Pr.Axis.Ra: num  72 57 66 63 103 50 65 65 61 62 ...
#> $ Max.L.Ra  : num  10 9 10 9 52 6 6 9 7 11 ...
#> $ Scat.Ra   : num  162 149 207 144 149 255 153 137 122 183 ...
#> $ Elong     : num  42 45 32 46 45 26 42 48 54 36 ...
#> $ Pr.Axis.Rect: num  20 19 23 19 19 28 19 18 17 22 ...
#> $ Max.L.Rect : num  159 143 158 143 144 169 143 146 127 146 ...
#> $ Sc.Var.Maxis: num  176 170 223 160 241 280 176 162 141 202 ...
#> $ Sc.Var.maxis: num  379 330 635 309 325 957 361 281 223 505 ...
#> $ Ra.Gyr    : num  184 158 220 127 188 264 172 164 112 152 ...
#> $ Skew.Maxis : num  70 72 73 63 127 85 66 67 64 64 ...
#> $ Skew.maxis : num  6 9 14 6 9 5 13 3 2 4 ...
#> $ Kurt.maxis : num  16 14 9 10 11 9 1 3 14 14 ...
#> $ Kurt.Maxis : num  187 189 188 199 180 181 200 193 200 195 ...
#> $ Holl.Ra   : num  197 199 196 207 183 183 204 202 208 204 ...
#> $ Class     : Factor w/ 4 levels "bus","opel","saab",...: 4 4 3 4 1 1 1 4 4 3 ...

summary(Vehicle[1])
#>          Comp
#> Min.    : 73.0
#> 1st Qu.: 87.0
#> Median  : 93.0
```

```
#> Mean    : 93.7
#> 3rd Qu.:100.0
#> Max.    :119.0

summary(Vehicle[2])
#>      Circ
#> Min.   :33.0
#> 1st Qu.:40.0
#> Median  :44.0
#> Mean    :44.9
#> 3rd Qu.:49.0
#> Max.    :59.0

attributes(Vehicle$Class)
#> $levels
#> [1] "bus"  "opel" "saab" "van"
#>
#> $class
#> [1] "factor"
```

15.2 Prepare data

```
set.seed(107)
N = nrow(Vehicle)
train <- sample(1:N, 500, FALSE)

# training and test sets
trainset <- Vehicle[train,]
testset  <- Vehicle[-train,]
```

15.3 Estimate the decision tree

```
fit <- tree(Class ~., data = trainset, split = "deviance")
fit
#> node), split, n, deviance, yval, (yprob)
#>      * denotes terminal node
#>
#> 1) root 500 1000 opel ( 0 0 0 0 )
#>    2) Elong < 41.5 215 500 saab ( 0 0 0 0 )
#>       4) Max.L.Ra < 7.5 51 50 bus ( 1 0 0 0 )
#>          8) Comp < 93.5 12 20 bus ( 0 0 0 0 )
#>             16) Pr.Axis.Ra < 67.5 7 8 saab ( 0 0 1 0 ) *
#>             17) Pr.Axis.Ra > 67.5 5 0 bus ( 1 0 0 0 ) *
#>             9) Comp > 93.5 39 9 bus ( 1 0 0 0 ) *
#>             5) Max.L.Ra > 7.5 164 200 opel ( 0 1 0 0 )
#>                10) Sc.Var.maxis < 723 149 200 saab ( 0 0 1 0 )
#>                   20) Comp < 109.5 137 200 opel ( 0 1 0 0 ) *
#>                   21) Comp > 109.5 12 0 saab ( 0 0 1 0 ) *
#>                   11) Sc.Var.maxis > 723 15 7 opel ( 0 1 0 0 ) *
#>                   3) Eelong > 41.5 285 700 van ( 0 0 0 0 )
```

```

#>      6) Sc.Var.maxis < 305.5 116 200 van ( 0 0 0 1 )
#>      12) Max.L.Rect < 128.5 40 90 saab ( 0 0 0 0 )
#>      24) Scat.Ra < 120.5 15 30 van ( 0 0 0 1 ) *
#>      25) Scat.Ra > 120.5 25 30 saab ( 0 0 1 0 ) *
#>      13) Max.L.Rect > 128.5 76 90 van ( 0 0 0 1 )
#>      26) Max.L.Rect < 138.5 38 60 van ( 0 0 0 1 )
#>      52) Circ < 37.5 17 10 van ( 0 0 0 1 ) *
#>      53) Circ > 37.5 21 40 opel ( 0 0 0 0 ) *
#>      27) Max.L.Rect > 138.5 38 20 van ( 0 0 0 1 ) *
#>      7) Sc.Var.maxis > 305.5 169 400 bus ( 0 0 0 0 )
#>      14) Max.L.Ra < 8.5 116 200 bus ( 1 0 0 0 )
#>      28) D.Circ < 76.5 97 100 bus ( 1 0 0 0 )
#>      56) Skew.maxis < 10.5 87 70 bus ( 1 0 0 0 )
#>      112) Max.L.Rect < 134.5 12 20 bus ( 0 0 0 0 ) *
#>      113) Max.L.Rect > 134.5 75 20 bus ( 1 0 0 0 ) *
#>      57) Skew.maxis > 10.5 10 20 opel ( 0 0 0 0 ) *
#>      29) D.Circ > 76.5 19 30 opel ( 0 1 0 0 ) *
#>      15) Max.L.Ra > 8.5 53 20 van ( 0 0 0 1 ) *

```

```

# fit <- tree(Class ~ ., data = Vehicle[train,], split ="deviance")
# fit

```

We use deviance as the splitting criteria, a common alternative is to use split="gini".

At each branch of the tree (after root) we see in order: 1. The branch number (e.g. in this case 1,2,14 and 15); 2. the split (e.g. Elong < 41.5); 3. the number of samples going along that split (e.g. 229); 4. the deviance associated with that split (e.g. 489.1); 5. the predicted class (e.g. opel); 6. the associated probabilities (e.g. (0.222707 0.410480 0.366812 0.000000)); 7. and for a terminal node (or leaf), the symbol "*".

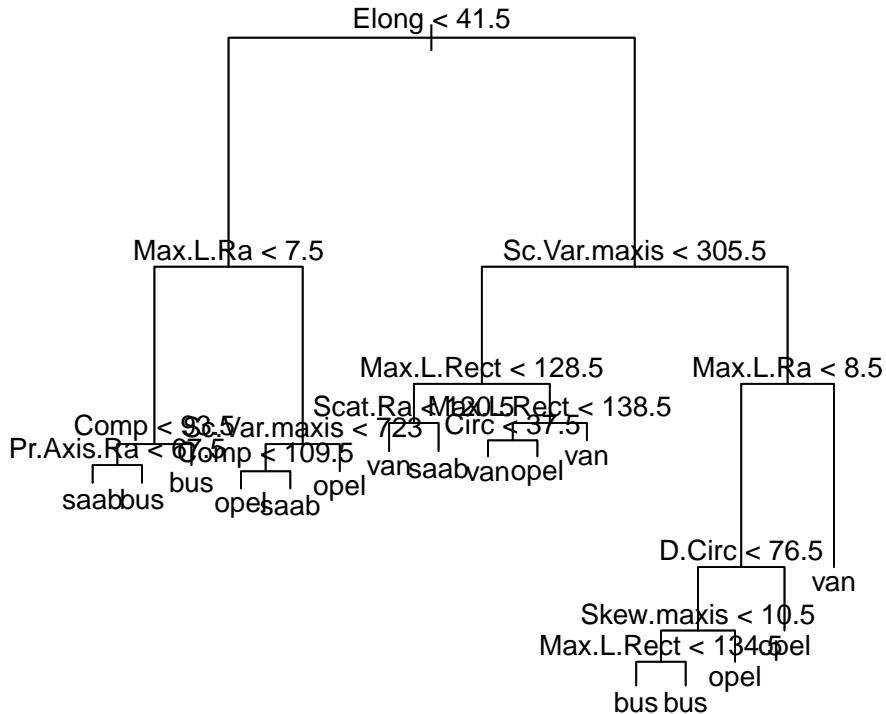
```

summary(fit)
#>
#> Classification tree:
#> tree(formula = Class ~ ., data = trainset, split = "deviance")
#> Variables actually used in tree construction:
#> [1] "Elong"          "Max.L.Ra"        "Comp"           "Pr.Axis.Ra"
#> [5] "Sc.Var.maxis"   "Max.L.Rect"     "Scat.Ra"        "Circ"
#> [9] "D.Circ"         "Skew.maxis"
#> Number of terminal nodes: 16
#> Residual mean deviance: 0.943 = 456 / 484
#> Misclassification error rate: 0.252 = 126 / 500

```

Notice that summary(fit) shows: 1. The type of tree, in this case a Classification tree; 2. the formula used to fit the tree; 3. the variables used to fit the tree; 4. the number of terminal nodes in this case 15; 5. the residual mean deviance - 0.9381; 6. the misclassification error rate 0.232 or 23.2%.

```
plot(fit); text(fit)
```



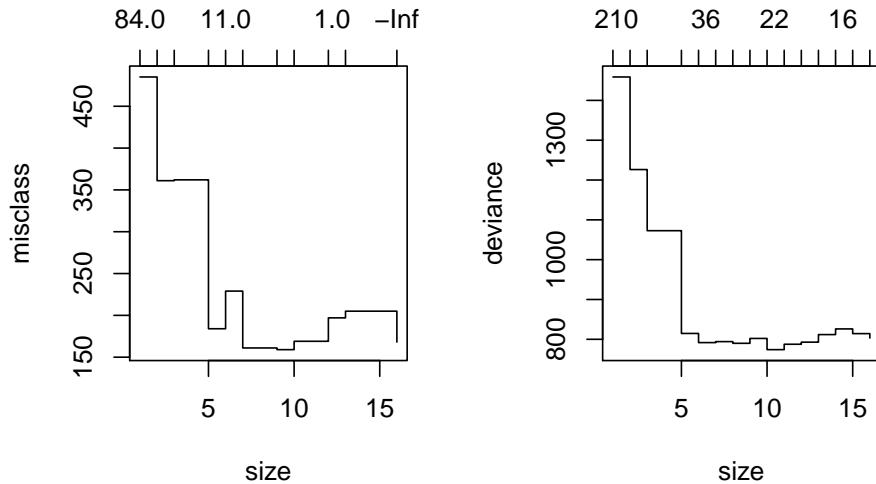
15.4 Assess model

Unfortunately, classification trees have a tendency to overfit the data. One approach to reduce this risk is to use cross-validation. For each hold out sample we fit the model and note at what level the tree gives the best results (using deviance or the misclassification rate). Then we hold out a different sample and repeat. This can be carried out using the `cv.tree()` function. We use a leave-one-out cross-validation using the misclassification rate and deviance (`FUN=prune.misclass`, followed by `FUN=prune.tree`).

```
fitM.cv <- cv.tree(fit, K=346, FUN = prune.misclass)
fitP.cv <- cv.tree(fit, K=346, FUN = prune.tree)
```

The results are plotted out side by side in Figure 1.2. The jagged lines shows where the minimum deviance / misclassification occurred with the cross-validated tree. Since the cross validated misclassification and deviance both reach their minimum close to the number of branches in the original fitted tree there is little to be gained from pruning this tree

```
par(mfrow = c(1, 2))
plot(fitM.cv)
plot(fitP.cv)
```



15.5 Make predictions

We use the validation data set and the fitted decision tree to predict vehicle classes; then we display the confusion matrix and calculate the error rate of the fitted tree. Overall, the model has an error rate of 32%.

```
testLabels <- Vehicle$Class[-train]
testLabels
#> [1] van bus bus van van bus bus saab opel bus van saab van saab
#> [15] saab van saab opel van saab saab saab bus bus saab opel bus opel
#> [29] bus opel van opel opel saab saab bus bus van van saab opel
#> [43] bus opel van opel saab bus van bus opel van saab bus opel bus
#> [57] opel opel van bus van saab opel bus van saab opel opel saab saab
#> [71] saab opel bus van bus opel bus saab bus bus bus opel opel van
#> [85] saab bus bus bus van saab opel van van bus bus opel bus opel
#> [99] saab opel bus opel bus saab van van saab saab bus van opel van
#> [113] saab opel saab saab van van van bus bus opel bus bus van
#> [127] saab bus opel bus bus bus opel van saab saab bus opel van
#> [141] bus saab bus van bus opel van saab opel saab opel van saab van
#> [155] saab opel bus van bus saab saab opel opel bus bus opel van van
#> [169] bus van van saab bus saab opel saab opel bus bus saab bus
#> [183] opel opel saab saab saab van van opel opel van van opel bus saab
#> [197] bus van opel opel bus bus bus opel saab opel van bus opel opel
#> [211] saab opel bus opel opel opel van opel van saab saab van saab saab
#> [225] saab saab van van van saab bus van van bus saab opel saab saab
#> [239] opel saab saab saab saab van saab opel bus saab bus opel opel opel
#> [253] saab bus van opel saab opel bus bus saab van opel bus saab van
#> [267] opel saab saab saab saab van opel bus bus bus opel saab saab saab
#> [281] van saab bus opel saab van opel bus saab saab opel opel van saab
#> [295] bus opel bus van van opel bus bus saab bus van saab bus van
#> [309] saab van opel bus bus opel saab opel bus bus saab van saab saab
#> [323] bus opel opel opel bus saab bus van bus van saab opel saab van
#> [337] opel opel van bus saab saab van saab opel saab

# Levels: bus opel saab van

# Confusion Matrix
pred <- predict(fit, newdata = testset)
# find column which has the maximum of all rows
```

```

pred.class <- colnames(pred)[max.col(pred, ties.method = c("random"))]
cm <- table(testLabels, pred.class,
             dnn = c("Observed Class", "Predicted Class"))
cm
#>           Predicted Class
#> Observed Class bus opel saab van
#>       bus     85    1    1   5
#>       opel     3   70   10   2
#>       saab     7   67   14   7
#>       van     1    4    5  64

# Sensitivity
sum(diag(cm)) / sum(cm)
#> [1] 0.673

# pred <- predict(fit, newdata = Vehicle[-train,])
# pred.class <- colnames(pred)[max.col(pred, ties.method = c("random"))]
# table(Vehicle$Class[-train], pred.class,
#       dnn = c("Observed Class", "Predicted Class"))

error_rate = (1 - sum(pred.class == testset) / nrow(testset))
round(error_rate, 3)
#> [1] 0.327

# error_rate = (1 - sum(pred.class == Vehicle$Class[-train])/346)
# round(error_rate,3)

```

Chapter 16

Bike sharing demand

```
#loading the required libraries
library(rpart)
library(rattle)
#> Rattle: A free graphical interface for data science with R.
#> Version 5.2.0 Copyright (c) 2006-2018 Togaware Pty Ltd.
#> Type 'rattle()' to shake, rattle, and roll your data.
library(rpart.plot)
library(RColorBrewer)
library(randomForest)
#> randomForest 4.6-14
#> Type rfNews() to see new features/changes/bug fixes.
#>
#> Attaching package: 'randomForest'
#> The following object is masked from 'package:rattle':
#>
#>     importance
library(corrplot)
#> corrplot 0.84 loaded
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following object is masked from 'package:randomForest':
#>
#>     combine
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
library(tictoc)
```

Source: <https://www.analyticsvidhya.com/blog/2015/06/solution-kaggle-competition-bike-sharing-demand/>

16.1 Hypothesis Generation

Before exploring the data to understand the relationship between variables, I'd recommend you to focus on hypothesis generation first. Now, this might sound counter-intuitive for solving a data science problem, but if there is one thing I have learnt over years, it is this. Before exploring data, you should spend some time thinking about the business problem, gaining the domain knowledge and may be gaining first hand experience of the problem (only if I could travel to North America!)

How does it help? This practice usually helps you form better features later on, which are not biased by the data available in the dataset. At this stage, you are expected to posses structured thinking i.e. a thinking process which takes into consideration all the possible aspects of a particular problem.

Here are some of the hypothesis which I thought could influence the demand of bikes:

- **Hourly trend:** There must be high demand during office timings. Early morning and late evening can have different trend (cyclist) and low demand during 10:00 pm to 4:00 am.
- **Daily Trend:** Registered users demand more bike on weekdays as compared to weekend or holiday.
- **Rain:** The demand of bikes will be lower on a rainy day as compared to a sunny day. Similarly, higher humidity will cause to lower the demand and vice versa.
- **Temperature:** Would high or low temperature encourage or disencourage bike riding?
- **Pollution:** If the pollution level in a city starts soaring, people may start using Bike (it may be influenced by government / company policies or increased awareness).
- **Time:** Total demand should have higher contribution of registered user as compared to casual because registered user base would increase over time.
- **Traffic:** It can be positively correlated with Bike demand. Higher traffic may force people to use bike as compared to other road transport medium like car, taxi etc

16.2 Understanding the Data Set

The dataset shows hourly rental data for two years (2011 and 2012). The training data set is for the **first 19 days of each month**. The test dataset is from **20th day to month's end**. We are required to predict the total count of bikes rented during each hour covered by the test set.

In the training data set, they have separately given bike demand by registered, casual users and sum of both is given as count.

Training data set has 12 variables (see below) and Test has 9 (excluding registered, casual and count).

16.2.1 Independent variables

```

datetime: date and hour in "mm/dd/yyyy hh:mm" format
season: Four categories-> 1 = spring, 2 = summer, 3 = fall, 4 = winter
holiday: whether the day is a holiday or not (1/0)
workingday: whether the day is neither a weekend nor holiday (1/0)
weather: Four Categories of weather
        1-> Clear, Few clouds, Partly cloudy, Partly cloudy
        2-> Mist + Cloudy, Mist + Broken clouds, Mist + Few clouds, Mist
        3-> Light Snow and Rain + Thunderstorm + Scattered clouds, Light Rain + Scattered clouds
        4-> Heavy Rain + Ice Pallets + Thunderstorm + Mist, Snow + Fog
temp: hourly temperature in Celsius
atemp: "feels like" temperature in Celsius

```

```
humidity: relative humidity
windspeed: wind speed
```

16.2.2 Dependent variables

```
registered: number of registered user
casual:     number of non-registered user
count:      number of total rentals (registered + casual)
```

16.3 Importing the dataset and Data Exploration

For this solution, I have used R (R Studio 0.99.442) in Windows Environment.

Below are the steps to import and perform data exploration. If you are new to this concept, you can refer this guide on Data Exploration in R

1. Import Train and Test Data Set

```
# https://www.kaggle.com/c/bike-sharing-demand/data
train = read.csv(file.path(data_raw_dir, "bike_train.csv"))
test = read.csv(file.path(data_raw_dir, "bike_test.csv"))

glimpse(train)
#> Observations: 10,886
#> Variables: 12
#> $ datetime <fct> 2011-01-01 00:00:00, 2011-01-01 01:00:00, 2011-01-0...
#> $ season <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
#> $ holiday <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
#> $ workingday <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
#> $ weather <int> 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 2, 2, 2, ...
#> $ temp <dbl> 9.84, 9.02, 9.02, 9.84, 9.84, 9.84, 9.02, 8.20, 9.8...
#> $ atemp <dbl> 14.4, 13.6, 13.6, 14.4, 14.4, 12.9, 13.6, 12.9, 14....
#> $ humidity <int> 81, 80, 80, 75, 75, 75, 80, 86, 75, 76, 76, 81, 77, ...
#> $ windspeed <dbl> 0, 0, 0, 0, 6, 0, 0, 0, 17, 19, 19, 20, 19, 2...
#> $ casual <int> 3, 8, 5, 3, 0, 0, 2, 1, 1, 8, 12, 26, 29, 47, 35, 4...
#> $ registered <int> 13, 32, 27, 10, 1, 1, 0, 2, 7, 6, 24, 30, 55, 47, 7...
#> $ count <int> 16, 40, 32, 13, 1, 1, 2, 3, 8, 14, 36, 56, 84, 94, ...

glimpse(test)
#> Observations: 6,493
#> Variables: 9
#> $ datetime <fct> 2011-01-20 00:00:00, 2011-01-20 01:00:00, 2011-01-2...
#> $ season <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
#> $ holiday <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
#> $ workingday <int> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, ...
#> $ weather <int> 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 2, 2, 2, ...
#> $ temp <dbl> 10.66, 10.66, 10.66, 10.66, 10.66, 9.84, 9.02, 9.02...
#> $ atemp <dbl> 11.4, 13.6, 13.6, 12.9, 12.9, 11.4, 10.6, 10.6, 10....
#> $ humidity <int> 56, 56, 56, 56, 60, 60, 55, 55, 52, 48, 45, 42, ...
#> $ windspeed <dbl> 26, 0, 0, 11, 11, 15, 15, 15, 19, 15, 20, 11, 0, 7, ...
```

2. Combine both Train and Test Data set (to understand the distribution of independent variable together).

```
# add variables to test dataset before merging
test$registered=0
test$casual=0
test$count=0

data = rbind(train,test)
```

3. Variable Type Identification

```
str(data)
#> 'data.frame': 17379 obs. of 12 variables:
#> $ datetime : Factor w/ 17379 levels "2011-01-01 00:00:00",...: 1 2 3 4 5 6 7 8 9 10 ...
#> $ season   : int 1 1 1 1 1 1 1 1 1 ...
#> $ holiday  : int 0 0 0 0 0 0 0 0 0 ...
#> $ workingday: int 0 0 0 0 0 0 0 0 0 ...
#> $ weather   : int 1 1 1 1 2 1 1 1 ...
#> $ temp      : num 9.84 9.02 9.02 9.84 9.84 ...
#> $ atemp     : num 14.4 13.6 13.6 14.4 14.4 ...
#> $ humidity  : int 81 80 80 75 75 75 80 86 75 76 ...
#> $ windspeed : num 0 0 0 0 0 ...
#> $ casual    : num 3 8 5 3 0 0 2 1 1 8 ...
#> $ registered: num 13 32 27 10 1 1 0 2 7 6 ...
#> $ count     : num 16 40 32 13 1 1 2 3 8 14 ...
```

4. Find missing values in the dataset if any

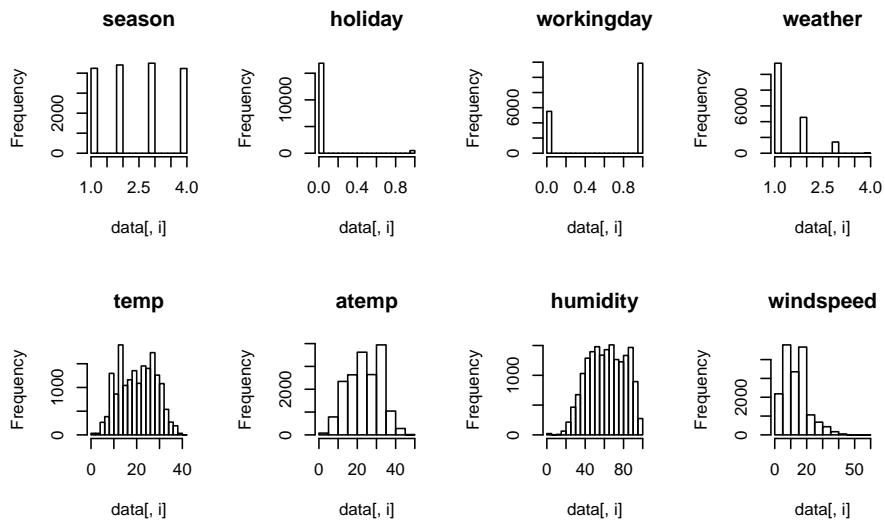
```
table(is.na(data))
#>
#> FALSE
#> 208548
```

No NAs in the dataset.

5. Understand the distribution of numerical variables and generate a frequency table for numeric variables. Analyze the distribution.

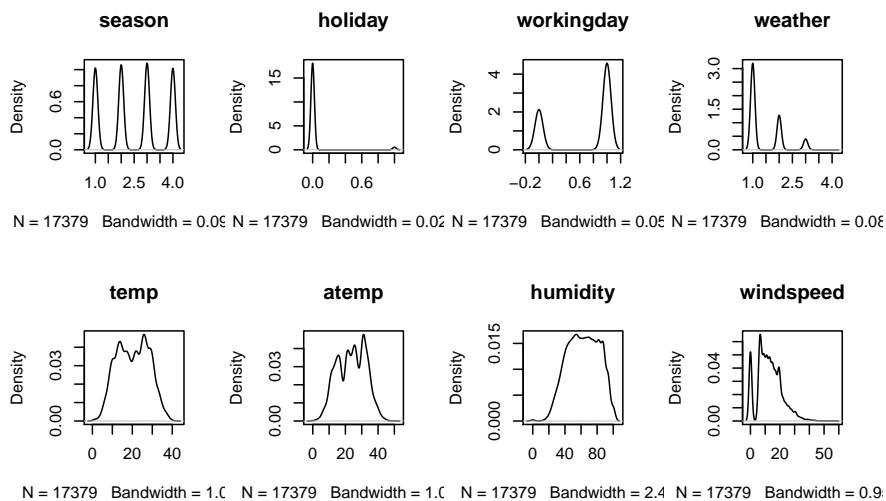
16.3.1 histograms

```
# histograms each attribute
par(mfrow=c(2,4))
for(i in 2:9) {
  hist(data[,i], main = names(data)[i])
}
```



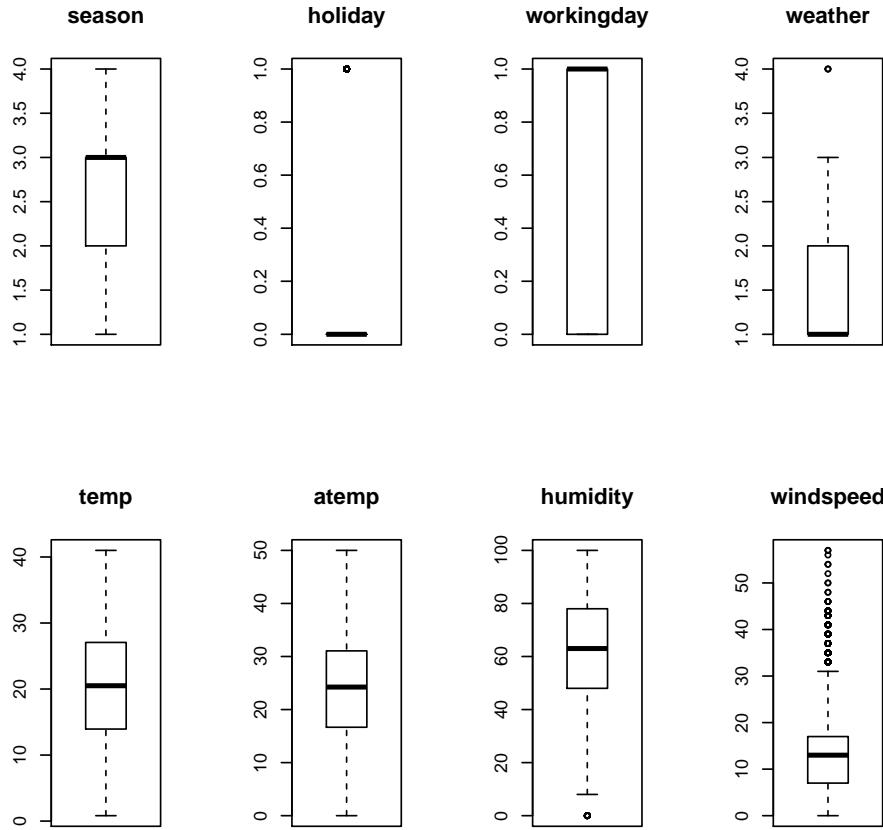
16.3.2 density plots

```
# density plot for each attribute
par(mfrow=c(2,4))
for(i in 2:9) {
  plot(density(data[,i]), main=names(data)[i])
}
```



16.3.3 boxplots

```
# boxplots for each attribute
par(mfrow=c(2,4))
for(i in 2:9) {
  boxplot(data[,i], main=names(data)[i])
}
```



16.3.4 Unique values of discrete variables

```
# the discrete variables in this case are integers
ints <- unlist(lapply(data, is.integer))
names(data)[ints]
#> [1] "season"      "holiday"       "workingday"    "weather"      "humidity"
```

Humidity should not be an integer or discrete variable; it is a continuous or numeric variable.

```
# convert humidity to numeric
data$humidity <- as.numeric(data$humidity)
```

```
# list unique values of integer variables
ints <- unlist(lapply(data, is.integer))
int_vars <- names(data)[ints]

sapply(int_vars, function(x) unique(data[x]))
#> $season.season
#> [1] 1 2 3 4
#>
#> $holiday.holiday
#> [1] 0 1
#>
#> $workingday.workingday
#> [1] 0 1
#>
#> $weather.weather
```

```
#> [1] 1 2 3 4
```

16.3.5 Inferences

1. The variables `season`, `holiday`, `workingday` and `weather` are discrete (integer).
2. Activity is even through all seasons.
3. Most of the activity happens during non-holidays.
4. Activity doubles during the working days.
5. Activity happens mostly during clear (1) weather.
6. `temp`, `atemp` and `humidity` are continuous variables (numeric).

16.4 Hypothesis Testing (using multivariate analysis)

Till now, we have got a fair understanding of the data set. Now, let's test the hypothesis which we had generated earlier. Here I have added some additional hypothesis from the dataset. Let's test them one by one:

16.4.1 Hourly trend

There must be high demand during office timings. Early morning and late evening can have different trend (cyclist) and low demand during 10:00 pm to 4:00 am.

We don't have the variable 'hour' with us. But we can extract it using the datetime column.

```
head(data$datetime)
#> [1] 2011-01-01 00:00:00 2011-01-01 01:00:00 2011-01-01 02:00:00
#> [4] 2011-01-01 03:00:00 2011-01-01 04:00:00 2011-01-01 05:00:00
#> 17379 Levels: 2011-01-01 00:00:00 2011-01-01 01:00:00 ... 2012-12-31 23:00:00

class(data$datetime)
#> [1] "factor"

# show hour and day from the variable datetime
head(substr(data$datetime, 12, 13)) # hour
#> [1] "00" "01" "02" "03" "04" "05"
head(substr(data$datetime, 9, 10)) # day
#> [1] "01" "01" "01" "01" "01" "01"

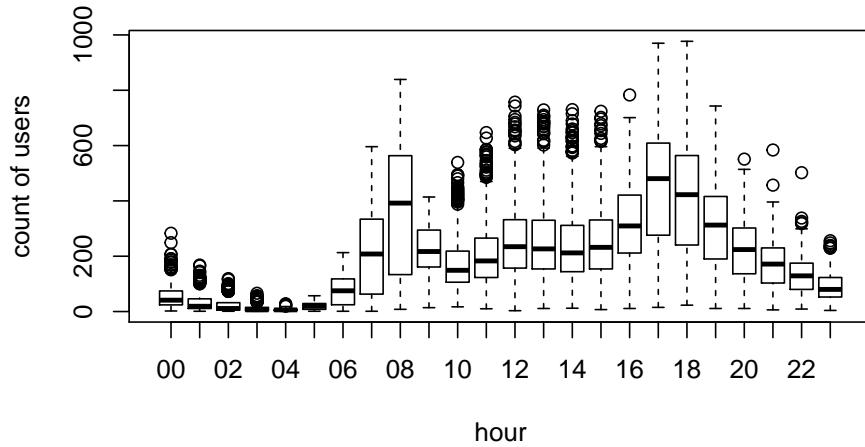
# extracting hour
data$hour = substr(data$datetime, 12, 13)
data$hour = as.factor(data$hour)
head(data$hour)
#> [1] 00 01 02 03 04 05
#> 24 Levels: 00 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 ... 23

### dividing again in train and test
# the train dataset is for the first 19 days
train = data[as.integer(substr(data$datetime, 9, 10)) < 20,]

# the test dataset is from day 20 to the end of the month
test = data[as.integer(substr(data$datetime, 9, 10)) > 19,]
```

16.4.2 boxplot count vs hour in training set

```
boxplot(train$count ~ train$hour, xlab="hour", ylab="count of users")
```



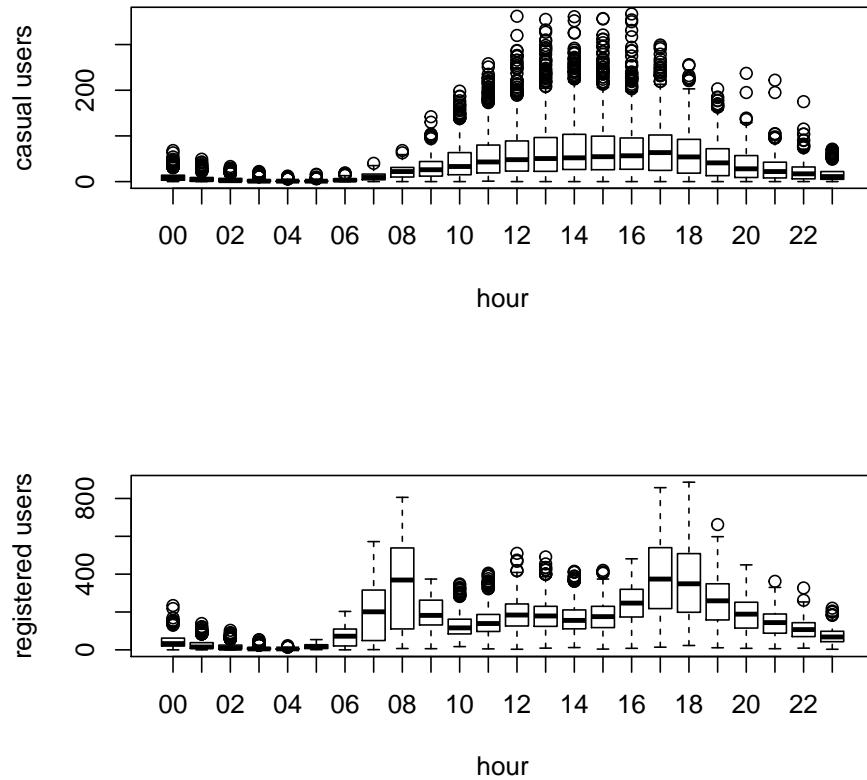
Rides increase from 6 am to 6pm, during office hours.

```
# casual users
casual <- data[data$casual > 0, ]
registered <- data[data$registered > 0, ]

dim(casual)
#> [1] 9900    13
dim(registered)
#> [1] 10871    13
```

16.4.3 Boxplot hourly: casual vs registered users in the training set

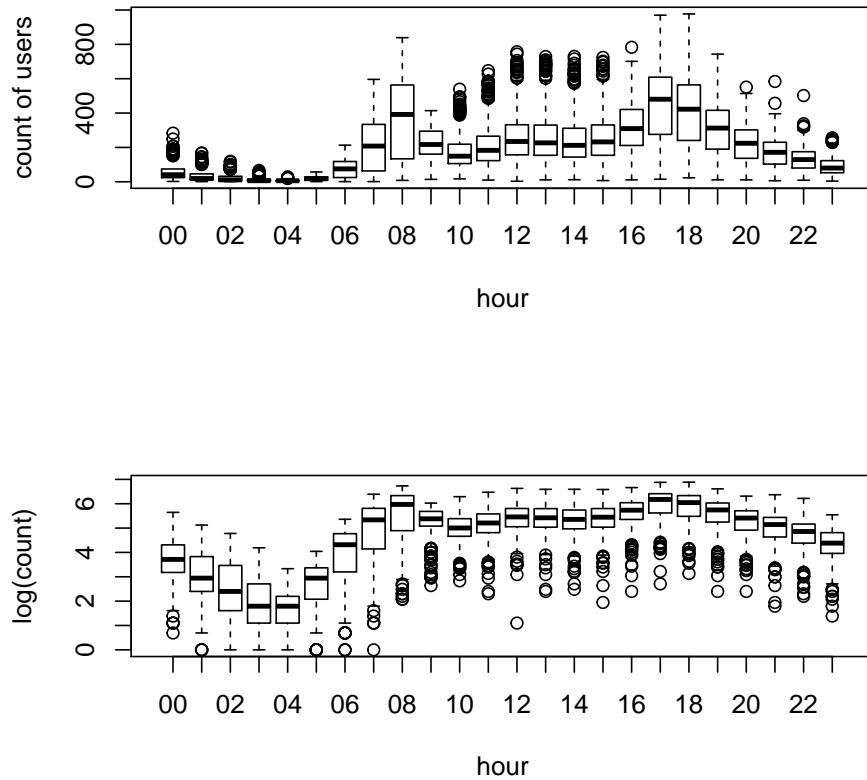
```
# by hour: casual vs registered users
par(mfrow=c(2,1))
boxplot(train$casual ~ train$hour, xlab="hour", ylab="casual users")
boxplot(train$registered ~ train$hour, xlab="hour", ylab="registered users")
```



Casual and Registered users have different distributions. Casual users tend to rent more during office hours.

16.4.4 outliers in the training set

```
par(mfrow=c(2,1))
boxplot(train$count ~ train$hour, xlab="hour", ylab="count of users")
boxplot(log(train$count) ~ train$hour,xlab="hour",ylab="log(count)")
```



16.4.5 Daily trend

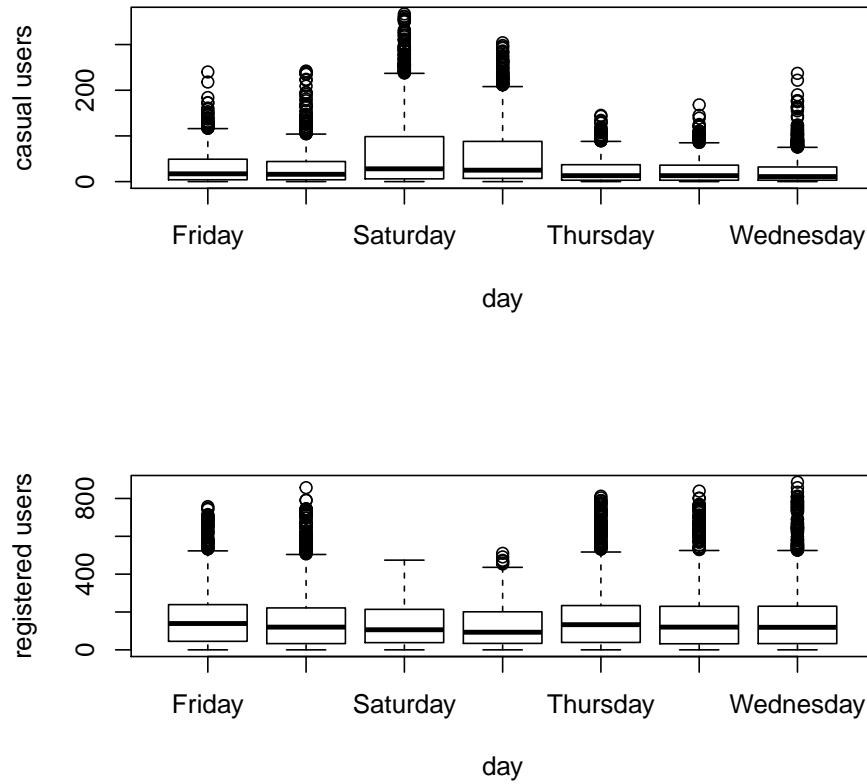
Registered users demand more bike on weekdays as compared to weekend or holiday.

```
# extracting days of week
date <- substr(data$datetime, 1, 10)
days <- weekdays(as.Date(date))
data$day <- days

# split the dataset again at day 20 of the month, before and after
train = data[as.integer(substr(data$datetime, 9, 10)) < 20,]
test = data[as.integer(substr(data$datetime, 9, 10)) > 19,]
```

16.4.6 Boxplot daily trend: casual vs registered users, training set

```
# creating boxplots for rentals with different variables to see the variation
par(mfrow=c(2,1))
boxplot(train$casual ~ train$day, xlab="day", ylab="casual users")
boxplot(train$registered ~ train$day, xlab="day", ylab="registered users")
```



Demand of casual users increases during the weekend, contrary of registered users.

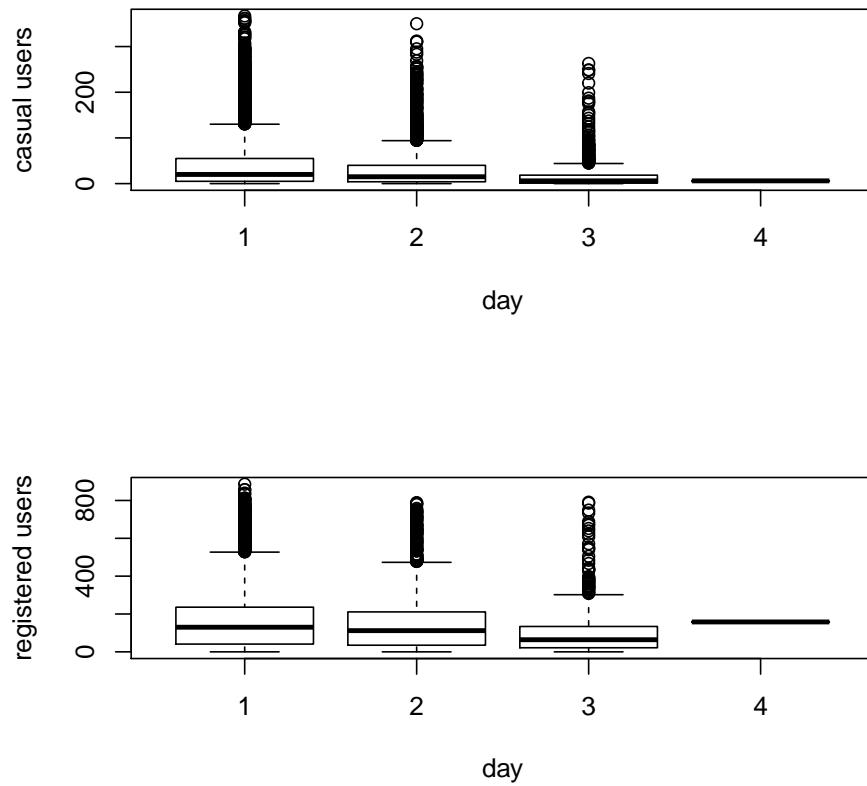
16.4.7 Rain

The demand of bikes will be lower on a rainy day as compared to a sunny day. Similarly, higher humidity will cause to lower the demand and vice versa.

We use the variable weather (1 to 4) to analyze riding under rain conditions.

16.4.7.1 Boxplot of rain effect on bike riding, training set

```
par(mfrow=c(2,1))
boxplot(train$casual ~ train$weather, xlab="day", ylab="casual users")
boxplot(train$registered ~ train$weather, xlab="day", ylab="registered users")
```



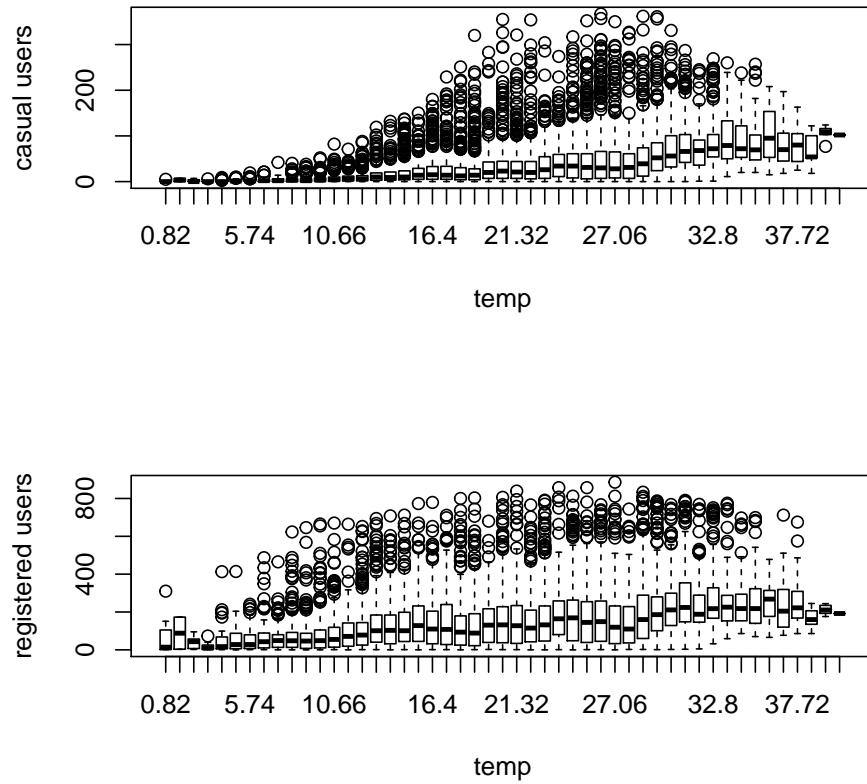
Registered used tend to ride even with rain.

16.4.8 Temperature

Would high or low temperature encourage or disencourage bike riding?

16.4.8.1 boxplot of temperature effect, training set

```
par(mfrow=c(2,1))
boxplot(train$casual ~ train$temp, xlab="temp", ylab="casual users")
boxplot(train$registered ~ train$temp, xlab="temp", ylab="registered users")
```



Casual users tend to ride with milder temperatures while registered users ride even at low temperatures.

16.4.9 Correlation

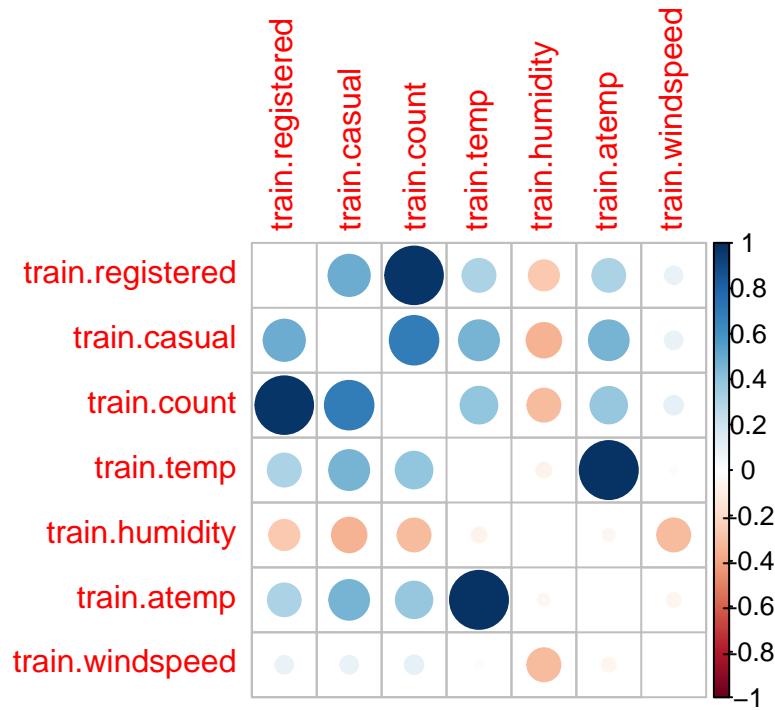
```

sub = data.frame(train$registered, train$casual, train$count, train$temp,
                 train$humidity, train$atemp, train$windspeed)
cor(sub)

#>          train.registered train.casual train.count train.temp
#> train.registered      1.0000     0.4972    0.971   0.3186
#> train.casual         0.4972     1.0000    0.690   0.4671
#> train.count           0.9709     0.6904    1.000   0.3945
#> train.temp            0.3186     0.4671    0.394   1.0000
#> train.humidity        -0.2655    -0.3482   -0.317  -0.0649
#> train.atemp           0.3146     0.4621    0.390   0.9849
#> train.windspeed       0.0911     0.0923    0.101  -0.0179
#>          train.humidity train.atemp train.windspeed
#> train.registered     -0.2655     0.3146    0.0911
#> train.casual         -0.3482     0.4621    0.0923
#> train.count           -0.3174     0.3898    0.1014
#> train.temp            -0.0649     0.9849   -0.0179
#> train.humidity        1.0000    -0.0435   -0.3186
#> train.atemp           -0.0435     1.0000   -0.0575
#> train.windspeed       -0.3186    -0.0575    1.0000

# do not show the diagonal
corrplot(cor(sub), diag = FALSE)

```



1. correlation between `casual` and `atemp`, `temp`.
2. Strong correlation between `temp` and `atemp`.

16.4.10 Activity by year

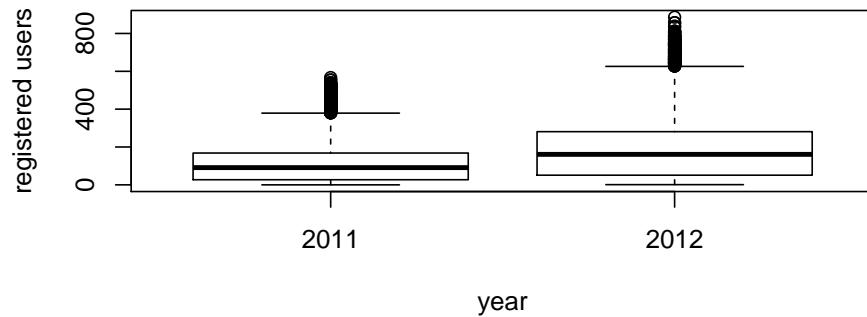
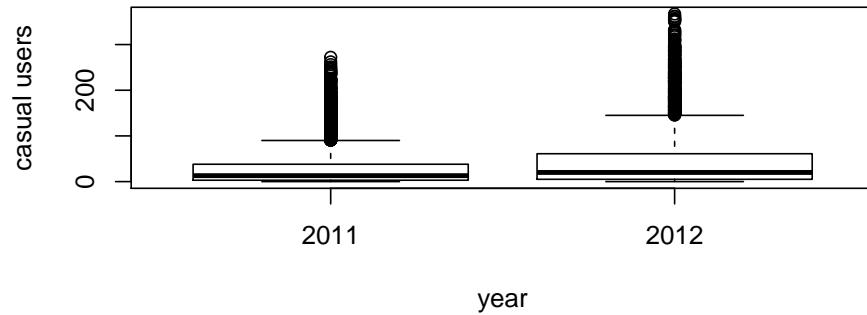
16.4.10.1 Year extraction

```
# extracting year
data$year = substr(data$datetime, 1, 4)
data$year = as.factor(data$year)

# ignore the division of data again and again, this could have been done together also
train = data[as.integer(substr(data$datetime,9,10)) < 20,]
test = data[as.integer(substr(data$datetime,9,10)) > 19,]
```

16.4.10.2 Trend by year, training set

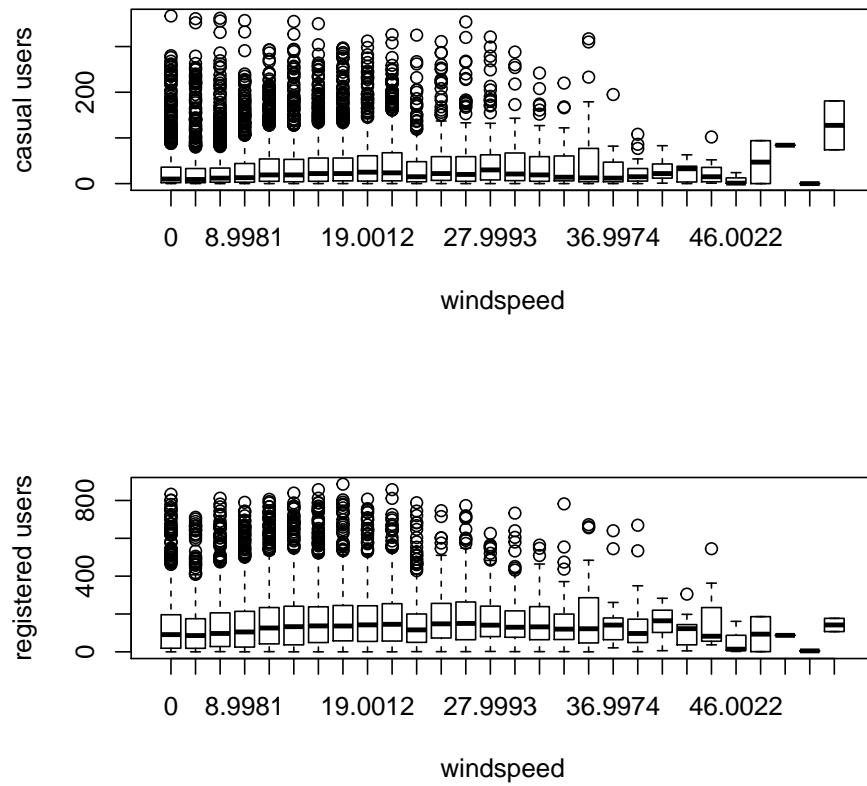
```
par(mfrow=c(2,1))
# again some boxplots with different variables
# these boxplots give important information about the dependent variable with respect to the independent
boxplot(train$casual ~ train$year, xlab="year", ylab="casual users")
boxplot(train$registered ~ train$year, xlab="year", ylab="registered users")
```



Activity increased in 2012.

16.4.10.3 trend by windspeed, training set

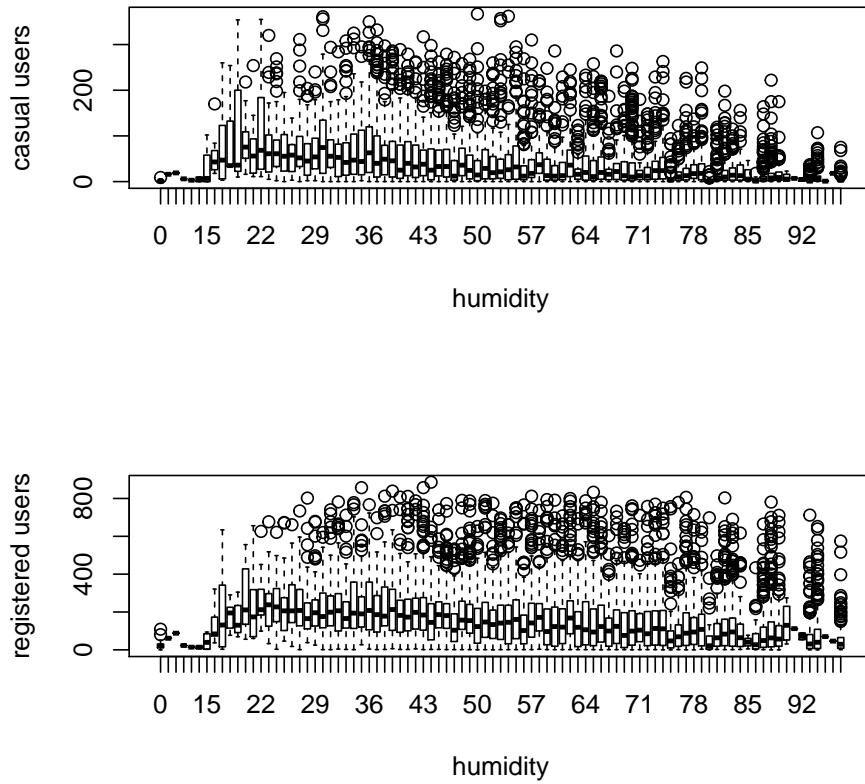
```
par(mfrow=c(2,1))
boxplot(train$casual ~ train$windspeed, xlab="windspeed", ylab="casual users")
boxplot(train$registered ~ train$windspeed, xlab="windspeed", ylab="registered users")
```



Casual users ride even with stron winds.

16.4.10.4 trend by humidity, training set

```
par(mfrow=c(2,1))
boxplot(train$casual ~ train$humidity, xlab="humidity", ylab="casual users")
boxplot(train$registered ~ train$humidity, xlab="humidity", ylab="registered users")
```



Casual users prefer not to ride with humid weather.

16.5 Feature Engineering

16.5.1 Prepare data

```
# factoring some variables from integer
data$season      <- as.factor(data$season)
data$weather     <- as.factor(data$weather)
data$holiday     <- as.factor(data$holiday)
data$workingday  <- as.factor(data$workingday)

# new column
data$hour <- as.integer(data$hour)

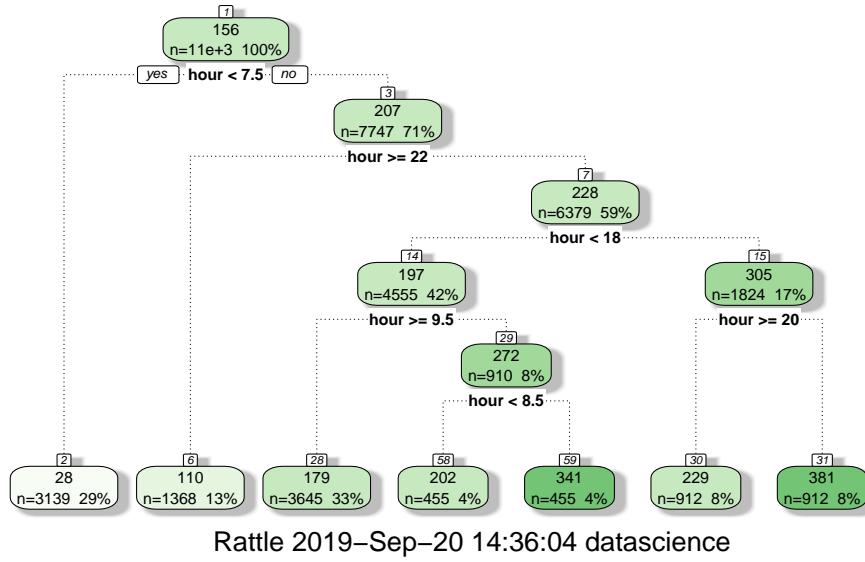
# created this variable to divide a day into parts, but did not finally use it
data$day_part <- 0

# split in training and test sets again
train <- data[as.integer(substr(data$datetime, 9, 10)) < 20,]
test  <- data[as.integer(substr(data$datetime, 9, 10)) > 19,]

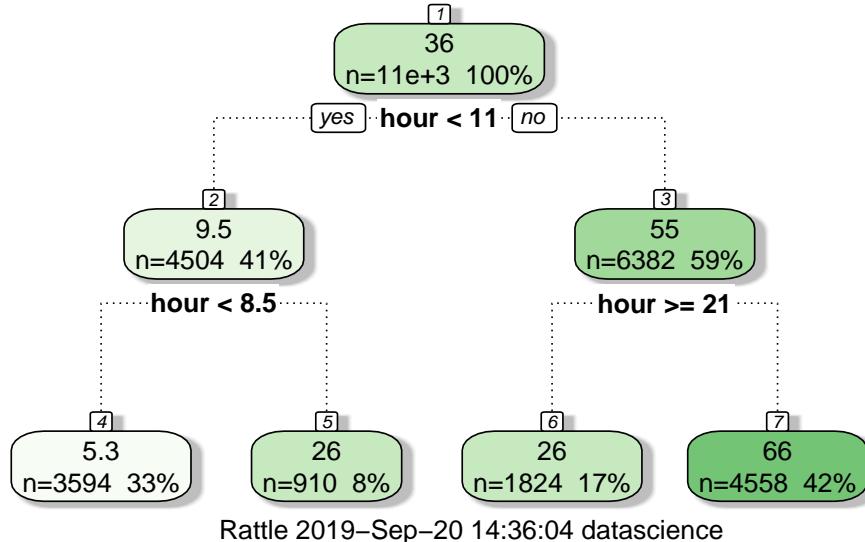
# combine the sets
data <- rbind(train, test)
```

16.5.2 Build hour bins

```
# for registered users
d = rpart(registered ~ hour, data = train)
fancyRpartPlot(d)
```



```
# for casual users
d = rpart(casual ~ hour, data = train)
fancyRpartPlot(d)
```



```
# Assign the timings according to tree
# fill the hour bins
data = rbind(train,test)

# create hour buckets for registered users
# 0,1,2,3,4,5,6,7 < 7.5
# 22,23,24 >=22
# 10,11,12,13,14,15,16,17: h>=9.5 & h<18
```

```

# h<9.5 & h<8.5 : 8
# h<9.5 & h>=8.5 : 9
# h>=20: 20,21
# h < 20: 18,19

data$dp_reg = 0
data$dp_reg[data$hour < 8] = 1
data$dp_reg[data$hour >= 22] = 2
data$dp_reg[data$hour > 9 & data$hour < 18] = 3
data$dp_reg[data$hour == 8] = 4
data$dp_reg[data$hour == 9] = 5
data$dp_reg[data$hour == 20 | data$hour == 21] = 6
data$dp_reg[data$hour == 19 | data$hour == 18] = 7

# casual users
# h<11, h<8.5: 0,1,2,3,4,5,6,7,8
# h>=8.5 & h<11: 9, 10
# h >=11 & h>=21: 21,22,23,24
# h >=11 & h<21: 11,12,13,14,15,16,17,18,19,20
data$dp_cas = 0
data$dp_cas[data$hour < 11 & data$hour >= 8] = 1
data$dp_cas[data$hour == 9 | data$hour == 10] = 2
data$dp_cas[data$hour >= 11 & data$hour < 21] = 3
data$dp_cas[data$hour >= 21] = 4

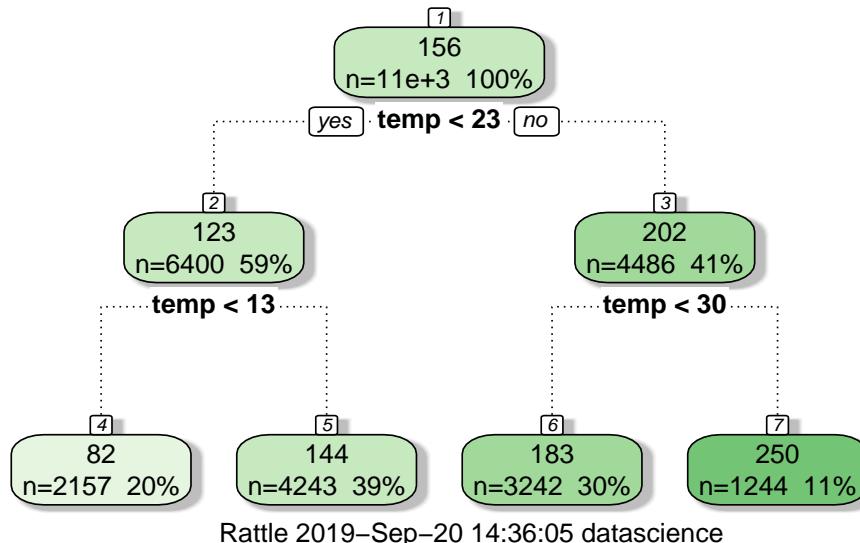
```

16.5.3 Temperature bins

```

# partition the data by temperature, registered users
f = rpart(registered ~ temp, data=train)
fancyRpartPlot(f)

```

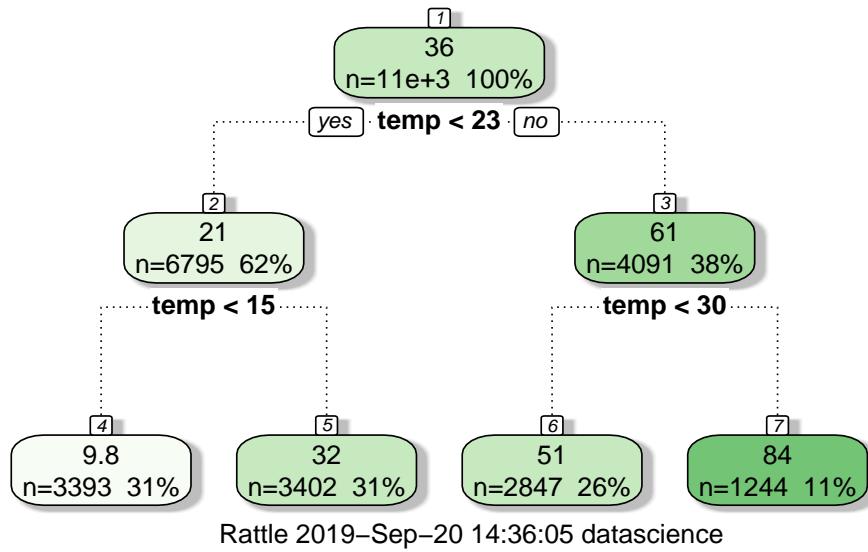


Rattle 2019–Sep–20 14:36:05 datascience

```

# partition the data by temperature,, casual users
f=rpart(casual ~ temp, data=train)
fancyRpartPlot(f)

```



16.5.3.1 Assign temperature ranges according to trees

```

data$temp_reg = 0
data$temp_reg[data$temp < 13] = 1
data$temp_reg[data$temp >= 13 & data$temp < 23] = 2
data$temp_reg[data$temp >= 23 & data$temp < 30] = 3
data$temp_reg[data$temp >= 30] = 4

data$temp_cas = 0
data$temp_cas[data$temp < 15] = 1
data$temp_cas[data$temp >= 15 & data$temp < 23] = 2
data$temp_cas[data$temp >= 23 & data$temp < 30] = 3
data$temp_cas[data$temp >= 30] = 4
  
```

16.5.4 Year bins by quarter

```

# add new variable with the month number
data$month <- substr(data$datetime, 6, 7)
data$month <- as.integer(data$month)

# bin by quarter manually
data$year_part[data$year=='2011'] = 1
data$year_part[data$year=='2011' & data$month>3] = 2
data$year_part[data$year=='2011' & data$month>6] = 3
data$year_part[data$year=='2011' & data$month>9] = 4
data$year_part[data$year=='2012'] = 5
data$year_part[data$year=='2012' & data$month>3] = 6
data$year_part[data$year=='2012' & data$month>6] = 7
data$year_part[data$year=='2012' & data$month>9] = 8
table(data$year_part)
#>
#>   1   2   3   4   5   6   7   8
#> 2067 2183 2192 2203 2176 2182 2208 2168
  
```

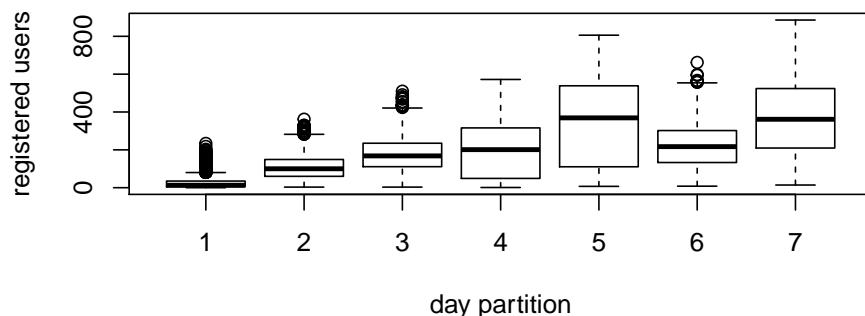
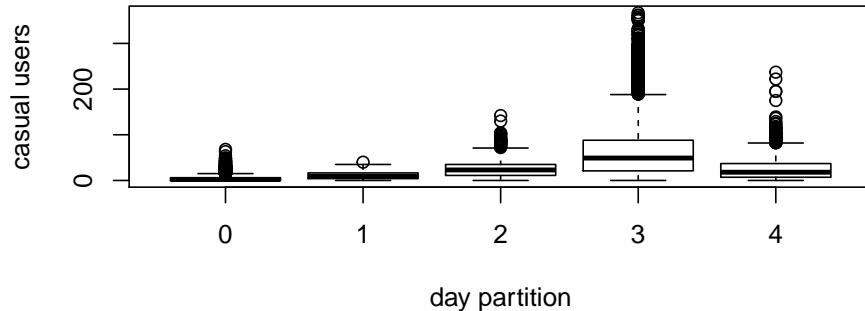
16.5.5 Day Type

Created a variable having categories like “weekday”, “weekend” and “holiday”.

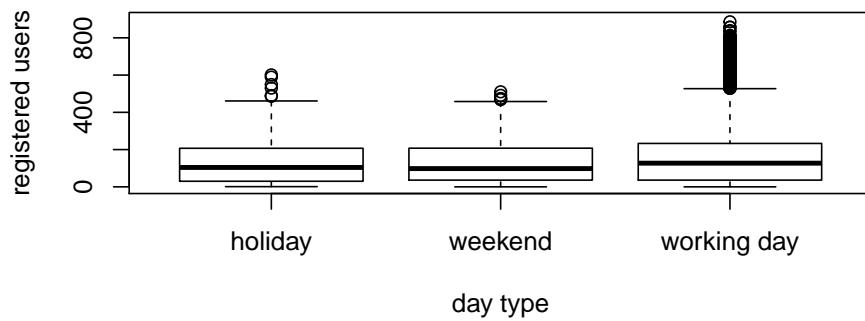
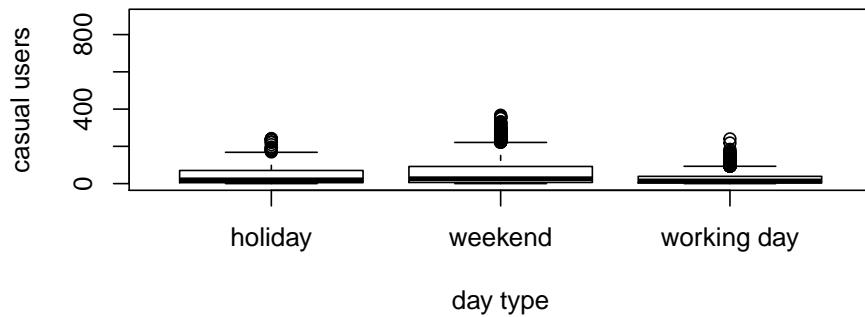
```
# creating another variable day_type which may affect our accuracy as weekends and weekdays are important
data$day_type = 0
data$day_type[data$holiday==0 & data$workingday==0] = "weekend"
data$day_type[data$holiday==1] = "holiday"
data$day_type[data$holiday==0 & data$workingday==1] = "working day"

# split dataset again
train = data[as.integer(substr(data$datetime,9,10)) < 20,]
test = data[as.integer(substr(data$datetime,9,10)) > 19,]

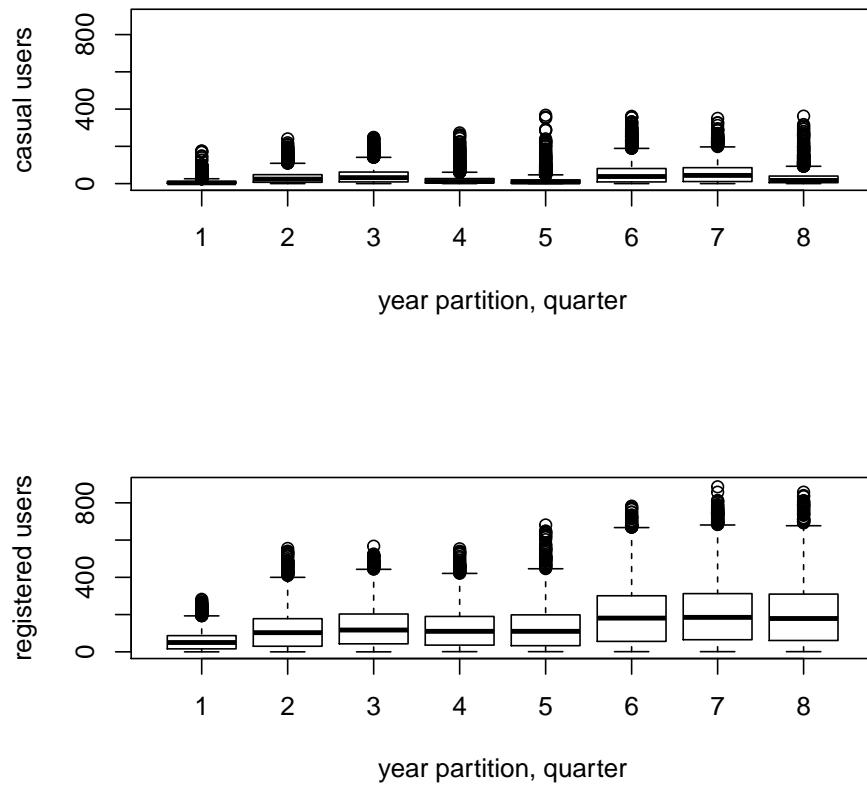
par(mfrow=c(2,1))
boxplot(train$casual ~ train$dp_cas, xlab = "day partition", ylab="casual users")
boxplot(train$registered ~ train$dp_reg, xlab = "day partition", ylab="registered users")
```



```
par(mfrow=c(2,1))
boxplot(train$casual ~ train$day_type, xlab = "day type",
        ylab="casual users", ylim = c(0,900))
boxplot(train$registered ~ train$day_type, xlab = "day type",
        ylab="registered users", ylim = c(0,900))
```

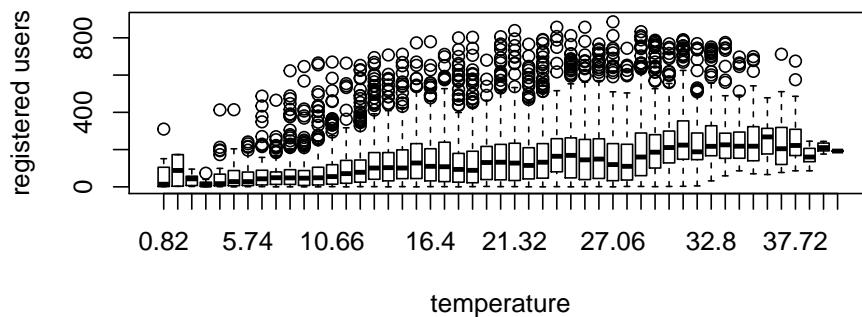
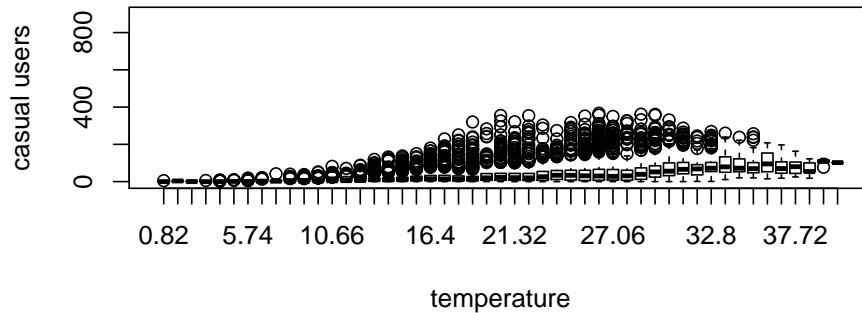


```
par(mfrow=c(2,1))
boxplot(train$casual ~ train$year_part, xlab = "year partition, quarter",
       ylab="casual users", ylim = c(0,900))
boxplot(train$registered ~ train$year_part, xlab = "year partition, quarter",
       ylab="registered users", ylim = c(0,900))
```

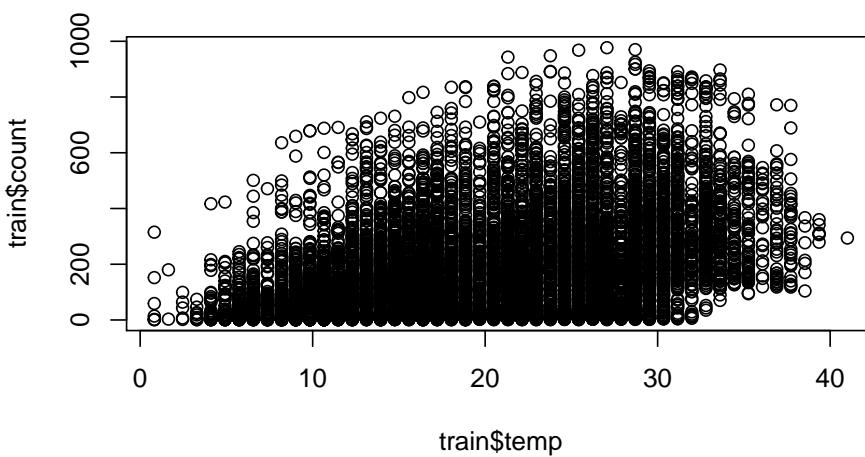


16.5.6 Temperatures

```
par(mfrow=c(2,1))
boxplot(train$casual ~ train$temp, xlab = "temperature",
       ylab="casual users", ylim = c(0,900))
boxplot(train$registered ~ train$temp, xlab = "temperature",
       ylab="registered users", ylim = c(0,900))
```



```
plot(train$temp, train$count)
data <- rbind(train, test)
# data$month <- substr(data$datetime, 6, 7)
# data$month <- as.integer(data$month)
```



16.5.7 Imputing missing data to wind speed

```
# dividing total data depending on windspeed to impute/predict the missing values
table(data$windspeed == 0)
#>
#> FALSE TRUE
#> 15199 2180
# FALSE TRUE
```

```

# 15199 2180

k = data$windspeed == 0

wind_0 = subset(data, k)      # windspeed is zero
wind_1 = subset(data, !k)     # windspeed not zero

tic()
# predicting missing values in windspeed using a random forest model
# this is a different approach to impute missing values rather than
# just using the mean or median or some other statistic for imputation

set.seed(415)
fit <- randomForest(windspeed ~ season + weather + humidity + month + temp +
                     year + atemp,
                     data = wind_1,
                     importance = TRUE,
                     ntree = 250)

pred = predict(fit, wind_0)
wind_0$windspeed = pred        # fill with wind speed predictions
toc()
#> 52.33 sec elapsed

# recompose the whole dataset
data = rbind(wind_0, wind_1)

# how many zero values now?
sum(data$windspeed == 0)
#> [1] 0

```

16.5.8 Weekend variable

Created a separate variable for weekend (0/1)

```

data$weekend = 0
data$weekend[data$day=="Sunday" | data$day=="Saturday"] = 1

```

16.6 Model Building

As this was our first attempt, we applied decision tree, conditional inference tree and random forest algorithms and found that random forest is performing the best. You can also go with regression, boosted regression, neural network and find which one is working well for you.

Before executing the random forest model code, I have followed following steps:

Convert discrete variables into factor (weather, season, hour, holiday, working day, month, day)

```

str(data)
#> 'data.frame': 17379 obs. of 24 variables:
#> $ datetime : Factor w/ 17379 levels "2011-01-01 00:00:00",...: 1 2 3 4 5 7 8 9 10 65 ...
#> $ season    : Factor w/ 4 levels "1","2","3","4": 1 1 1 1 1 1 1 1 ...
#> $ holiday   : Factor w/ 2 levels "0","1": 1 1 1 1 1 1 1 1 ...

```

```
#> $ workingday: Factor w/ 2 levels "0","1": 1 1 1 1 1 1 1 1 1 2 ...
#> $ weather : Factor w/ 4 levels "1","2","3","4": 1 1 1 1 1 1 1 1 1 1 ...
#> $ temp    : num 9.84 9.02 9.02 9.84 9.84 ...
#> $ atemp   : num 14.4 13.6 13.6 14.4 14.4 ...
#> $ humidity : num 81 80 80 75 75 80 86 75 76 47 ...
#> $ windspeed : num 9.03 9.05 9.05 9.15 9.15 ...
#> $ casual   : num 3 8 5 3 0 2 1 1 8 8 ...
#> $ registered: num 13 32 27 10 1 0 2 7 6 102 ...
#> $ count    : num 16 40 32 13 1 2 3 8 14 110 ...
#> $ hour     : int 1 2 3 4 5 7 8 9 10 20 ...
#> $ day      : chr "Saturday" "Saturday" "Saturday" "Saturday" ...
#> $ year     : Factor w/ 2 levels "2011","2012": 1 1 1 1 1 1 1 1 1 1 ...
#> $ day_part : num 0 0 0 0 0 0 0 0 0 0 ...
#> $ dp_reg   : num 1 1 1 1 1 4 5 3 6 ...
#> $ dp_cas   : num 0 0 0 0 0 0 1 2 2 3 ...
#> $ temp_reg : num 1 1 1 1 1 1 1 1 2 1 ...
#> $ temp_cas : num 1 1 1 1 1 1 1 1 1 1 ...
#> $ month    : int 1 1 1 1 1 1 1 1 1 1 ...
#> $ year_part: num 1 1 1 1 1 1 1 1 1 1 ...
#> $ day_type : chr "weekend" "weekend" "weekend" "weekend" ...
#> $ weekend  : num 1 1 1 1 1 1 1 1 1 0 ...
```

16.6.1 Convert variables to factors

```
# converting all relevant categorical variables into factors to feed to our random forest model
data$season      = as.factor(data$season)
data$holiday     = as.factor(data$holiday)
data$workingday  = as.factor(data$workingday)
data$weather     = as.factor(data$weather)
data$hour        = as.factor(data$hour)
data$month       = as.factor(data$month)
data$day_part    = as.factor(data$dp_cas)
data$day_type    = as.factor(data$dp_reg)
data$day         = as.factor(data$day)
data$temp_cas   = as.factor(data$temp_cas)
data$temp_reg   = as.factor(data$temp_reg)

str(data)
#> 'data.frame': 17379 obs. of 24 variables:
#> $ datetime : Factor w/ 17379 levels "2011-01-01 00:00:00",...: 1 2 3 4 5 7 8 9 10 65 ...
#> $ season   : Factor w/ 4 levels "1","2","3","4": 1 1 1 1 1 1 1 1 1 1 ...
#> $ holiday  : Factor w/ 2 levels "0","1": 1 1 1 1 1 1 1 1 1 1 ...
#> $ workingday: Factor w/ 2 levels "0","1": 1 1 1 1 1 1 1 1 1 2 ...
#> $ weather   : Factor w/ 4 levels "1","2","3","4": 1 1 1 1 1 1 1 1 1 1 ...
#> $ temp      : num 9.84 9.02 9.02 9.84 9.84 ...
#> $ atemp     : num 14.4 13.6 13.6 14.4 14.4 ...
#> $ humidity  : num 81 80 80 75 75 80 86 75 76 47 ...
#> $ windspeed : num 9.03 9.05 9.05 9.15 9.15 ...
#> $ casual    : num 3 8 5 3 0 2 1 1 8 8 ...
#> $ registered: num 13 32 27 10 1 0 2 7 6 102 ...
#> $ count     : num 16 40 32 13 1 2 3 8 14 110 ...
#> $ hour      : Factor w/ 24 levels "1","2","3","4",...: 1 2 3 4 5 7 8 9 10 20 ...
```

```
#> $ day      : Factor w/ 7 levels "Friday","Monday",...: 3 3 3 3 3 3 3 3 3 2 ...
#> $ year     : Factor w/ 2 levels "2011","2012": 1 1 1 1 1 1 1 1 1 1 ...
#> $ day_part : Factor w/ 5 levels "0","1","2","3",...: 1 1 1 1 1 1 2 3 3 4 ...
#> $ dp_reg   : num  1 1 1 1 1 1 4 5 3 6 ...
#> $ dp_cas   : num  0 0 0 0 0 1 2 2 3 ...
#> $ temp_reg : Factor w/ 4 levels "1","2","3","4": 1 1 1 1 1 1 1 1 2 1 ...
#> $ temp_cas : Factor w/ 4 levels "1","2","3","4": 1 1 1 1 1 1 1 1 1 1 ...
#> $ month    : Factor w/ 12 levels "1","2","3","4",...: 1 1 1 1 1 1 1 1 1 1 ...
#> $ year_part: num  1 1 1 1 1 1 1 1 1 ...
#> $ day_type : Factor w/ 7 levels "1","2","3","4",...: 1 1 1 1 1 1 4 5 3 6 ...
#> $ weekend  : num  1 1 1 1 1 1 1 1 0 ...
```

- As we know that dependent variables have natural outliers so we will predict log of dependent variables.
- Predict bike demand registered and casual users separately. $y1 = \log(casual + 1)$ and $y2 = \log(registered + 1)$, Here we have added 1 to deal with zero values in the casual and registered columns.

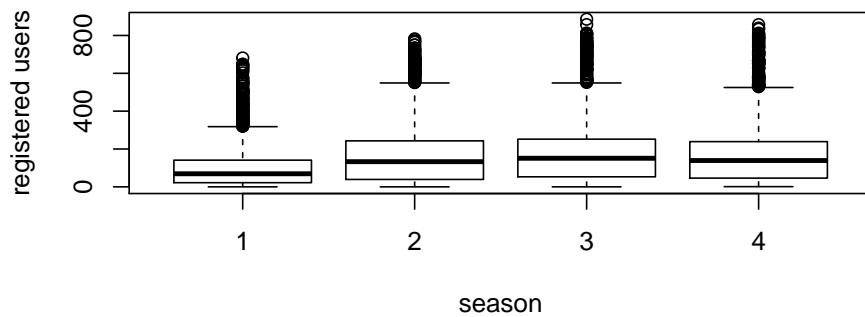
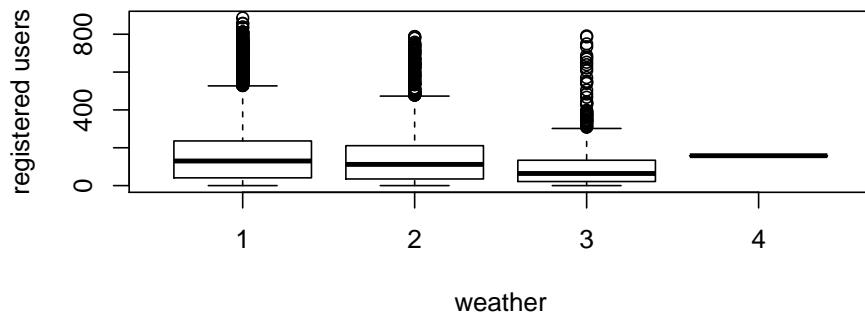
```
# separate again as train and test set
train = data[as.integer(substr(data$datetime, 9, 10)) < 20,]
test = data[as.integer(substr(data$datetime, 9, 10)) > 19,]
```

16.6.2 Log transform

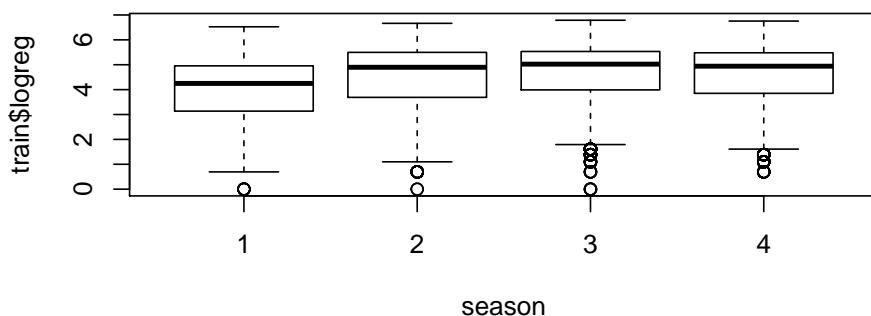
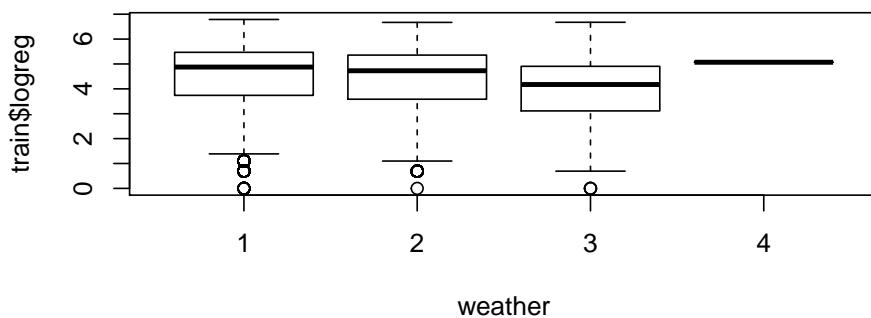
```
# log transformation for some skewed variables,
# which can be seen from their distribution
train$reg1  = train$registered + 1
train$cas1  = train$casual + 1
train$logcas = log(train$cas1)
train$logreg = log(train$reg1)
test$logreg = 0
test$logcas = 0
```

16.6.2.1 Plot by weather, by season

```
# cartesian plot
par(mfrow=c(2,1))
boxplot(train$registered ~ train$weather, xlab="weather", ylab="registered users")
boxplot(train$registered ~ train$season, xlab="season", ylab="registered users")
```



```
# semilog plot
par(mfrow=c(2,1))
boxplot(train$logreg ~ train$weather, xlab = "weather")
boxplot(train$logreg ~ train$season, xlab = "season")
```



16.6.3 Predicting for registered and casual users, test dataset

```

tic()
# final model building using random forest
# note that we build different models for predicting for
# registered and casual users
# this was seen as giving best result after a lot of experimentation
set.seed(415)
fit1 <- randomForest(logreg ~ hour + workingday + day + holiday + day_type +
                      temp_reg + humidity + atemp + windspeed + season +
                      weather + dp_reg + weekend + year + year_part,
                      data = train,
                      importance = TRUE,
                      ntree = 250)

pred1 = predict(fit1, test)
test$logreg = pred1
toc()
#> 131.372 sec elapsed

# casual users
set.seed(415)
fit2 <- randomForest(logcas ~ hour + day_type + day + humidity + atemp +
                      temp_cas + windspeed + season + weather + holiday +
                      workingday + dp_cas + weekend + year + year_part,
                      data = train, importance = TRUE, ntree = 250)

pred2 = predict(fit2, test)
test$logcas = pred2

```

16.6.4 Preparing and exporting results

```

# creating the final submission file
# reverse log conversion
test$registered <- exp(test$logreg) - 1
test$casual     <- exp(test$logcas) - 1
test$count       <- test$casual + test$registered

r <- data.frame(datetime = test$datetime,
                 casual = test$casual,
                 registered = test$registered)

print(sum(r$casual))
#> [1] 205804
print(sum(r$registered))
#> [1] 962834

s <- data.frame(datetime = test$datetime, count = test$count)
write.csv(s, file = file.path(data_out_dir, "bike-submit.csv"), row.names = FALSE)

# sum(cas+reg) = 1168638
# month number now is correct

```

After following the steps mentioned above, you can score 0.38675 on Kaggle leaderboard i.e. top 5 percentile of total participants. As you might have seen, we have not applied any extraordinary science in getting to this level. But, the real competition starts here. I would like to see, if I can improve this further by use of more features and some more advanced modeling techniques.

16.7 End Notes

In this article, we have looked at structured approach of problem solving and how this method can help you to improve performance. I would recommend you to generate hypothesis before you deep dive in the data set as this technique will not limit your thought process. You can improve your performance by applying advanced techniques (or ensemble methods) and understand your data trend better.

You can find the complete solution here : GitHub Link

```
# this is the older submission. months were incomplete
old <- read.csv(file = file.path(data_raw_dir, "bike-submit-old.csv"))
sum(old$count)
#> [1] 1164829
```

Chapter 17

Breast Cancer Wisconsin

Source: https://shiring.github.io/machine_learning/2017/01/15/rfe_ga_post

17.1 Read and process the data

```
bc_data <- read.table(file.path(data_raw_dir, "breast-cancer-wisconsin.data"),
                      header = FALSE, sep = ",")  
  
# assign the column names
colnames(bc_data) <- c("sample_code_number", "clump_thickness",
                       "uniformity_of_cell_size", "uniformity_of_cell_shape",
                       "marginal_adhesion", "single_epithelial_cell_size",
                       "bare_nuclei", "bland_chromatin", "normal_nucleoli",
                       "mitosis", "classes")  
  
# change classes from numeric to character
bc_data$classes <- ifelse(bc_data$classes == "2", "benign",
                           ifelse(bc_data$classes == "4", "malignant", NA))  
  
# if query sign make NA
bc_data[bc_data == "?"] <- NA  
  
# how many NAs are in the data
length(which(is.na(bc_data)))
#> [1] 16  
  
names(bc_data)
#> [1] "sample_code_number"          "clump_thickness"
#> [3] "uniformity_of_cell_size"     "uniformity_of_cell_shape"
#> [5] "marginal_adhesion"          "single_epithelial_cell_size"
#> [7] "bare_nuclei"                 "bland_chromatin"
#> [9] "normal_nucleoli"            "mitosis"
#> [11] "classes"
```

17.1.1 Missing data

```
# impute missing data
library(mice)
#> Loading required package: lattice
#>
#> Attaching package: 'mice'
#> The following objects are masked from 'package:base':
#>
#>     cbind, rbind

# skip these columns: sample_code_number and classes
# convert to numeric
bc_data[,2:10] <- apply(bc_data[, 2:10], 2, function(x) as.numeric(as.character(x)))

# impute but mute
dataset_impute <- mice(bc_data[, 2:10], print = FALSE)

# bind "classes" with the rest. skip "sample_code_number"
bc_data <- cbind(bc_data[, 11, drop = FALSE],
                  mice::complete(dataset_impute, action =1))

bc_data$classes <- as.factor(bc_data$classes)

# how many benign and malignant cases are there?
summary(bc_data$classes)
#>   benign malignant
#>       458      241

# confirm NAs have been removed
length(which(is.na(bc_data)))
#> [1] 0

str(bc_data)
#> 'data.frame': 699 obs. of 10 variables:
#> $ classes : Factor w/ 2 levels "benign", "malignant": 1 1 1 1 1 2 1 1 1 ...
#> $ clump_thickness : num 5 5 3 6 4 8 1 2 2 4 ...
#> $ uniformity_of_cell_size : num 1 4 1 8 1 10 1 1 1 2 ...
#> $ uniformity_of_cell_shape : num 1 4 1 8 1 10 1 2 1 1 ...
#> $ marginal_adhesion : num 1 5 1 1 3 8 1 1 1 1 ...
#> $ single_epithelial_cell_size: num 2 7 2 3 2 7 2 2 2 2 ...
#> $ bare_nuclei : num 1 10 2 4 1 10 10 1 1 1 ...
#> $ bland_chromatin : num 3 3 3 3 9 3 3 1 2 ...
#> $ normal_nucleoli : num 1 2 1 7 1 7 1 1 1 1 ...
#> $ mitosis : num 1 1 1 1 1 1 1 5 1 ...
```

17.2 Principal Component Analysis (PCA)

To get an idea about the dimensionality and variance of the datasets, I am first looking at PCA plots for samples and features. The first two principal components (PCs) show the two components that explain the majority of variation in the data.

After defining my custom `ggplot2` theme, I am creating a function that performs the PCA (using the

pcaGoPromoter package), calculates ellipses of the data points (with the ellipse package) and produces the plot with ggplot2. Some of the features in datasets 2 and 3 are not very distinct and overlap in the PCA plots, therefore I am also plotting hierarchical clustering dendograms.

17.2.0.1 theme

```
# plotting theme

library(ggplot2)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures    rlang
#>   c.quosures    rlang
#>   print.quosures rlang

my_theme <- function(base_size = 12, base_family = "sans"){
  theme_minimal(base_size = base_size, base_family = base_family) +
  theme(
    axis.text = element_text(size = 12),
    axis.text.x = element_text(angle = 0, vjust = 0.5, hjust = 0.5),
    axis.title = element_text(size = 14),
    panel.grid.major = element_line(color = "grey"),
    panel.grid.minor = element_blank(),
    panel.background = element_rect(fill = "aliceblue"),
    strip.background = element_rect(fill = "navy", color = "navy", size = 1),
    strip.text = element_text(face = "bold", size = 12, color = "white"),
    legend.position = "right",
    legend.justification = "top",
    legend.background = element_blank(),
    panel.border = element_rect(color = "grey", fill = NA, size = 0.5)
  )
}

theme_set(my_theme())
```

17.2.0.2 PCA function

```
# function for PCA plotting
library(pcaGoPromoter)                      # install from BioConductor
#> Loading required package: ellipse
#>
#> Attaching package: 'ellipse'
#> The following object is masked from 'package:graphics':
#>
#>   pairs
#> Loading required package: Biostrings
#> Loading required package: BiocGenerics
#> Loading required package: parallel
#>
#> Attaching package: 'BiocGenerics'
#> The following objects are masked from 'package:parallel':
```

```

#>
#>     clusterApply, clusterApplyLB, clusterCall, clusterEvalQ,
#>     clusterExport, clusterMap, parApply, parCapply, parLapply,
#>     parLapplyLB, parRapply, parSapply, parSapplyLB
#> The following objects are masked from 'package:mice':
#>
#>     cbind, rbind
#> The following objects are masked from 'package:stats':
#>
#>     IQR, mad, sd, var, xtabs
#> The following objects are masked from 'package:base':
#>
#>     anyDuplicated, append, as.data.frame, basename, cbind,
#>     colnames, dirname, do.call, duplicated, eval, evalq, Filter,
#>     Find, get, grep, grepl, intersect, is.unsorted, lapply, Map,
#>     mapply, match, mget, order, paste, pmax, pmax.int, pmin,
#>     pmin.int, Position, rank, rbind, Reduce, rownames, sapply,
#>     setdiff, sort, table, tapply, union, unique, unsplit, which,
#>     which.max, which.min
#> Loading required package: S4Vectors
#> Loading required package: stats4
#>
#> Attaching package: 'S4Vectors'
#> The following object is masked from 'package:base':
#>
#>     expand.grid
#> Loading required package: IRanges
#> Loading required package: XVector
#>
#> Attaching package: 'Biostrings'
#> The following object is masked from 'package:base':
#>
#>     strsplit
library(ellipse)

pca_func <- function(data, groups, title, print_ellipse = TRUE) {

  # perform pca and extract scores for all principal components: PC1:PC9
  pcaOutput <- pca(data, printDropped = FALSE, scale = TRUE, center = TRUE)
  pcaOutput2 <- as.data.frame(pcaOutput$scores)

  # define groups for plotting. will group the classes
  pcaOutput2$groups <- groups

  # when plotting samples calculate ellipses for plotting
  # (when plotting features, there are no replicates)
  if (print_ellipse) {
    # group and summarize by classes: benign, malignant
    # centroids w/3 columns: groups, PC1, PC2
    centroids <- aggregate(cbind(PC1, PC2) ~ groups, pcaOutput2, mean)
    # bind for the two groups (classes)
    # conf.rgn w/3 columns: groups, PC1, PC2
    conf.rgn <- do.call(rbind, lapply(unique(pcaOutput2$groups), function(t)

```

```

data.frame(groups = as.character(t),
           # ellipse data for PC1 and PC2
           ellipse(cov(pcaOutput2[pcaOutput2$groups == t, 1:2]),
                   centre = as.matrix(centroids[centroids$groups == t, 2:3]),
                   level = 0.95,
                   stringsAsFactors = FALSE)))

plot <- ggplot(data = pcaOutput2, aes(x = PC1,
                                         group = groups,
                                         color = groups)) +
  geom_polygon(data = conf.rgn, aes(fill = groups), alpha = 0.2) + # ellipses
  geom_point(size = 2, alpha = 0.6) +
  scale_color_brewer(palette = "Set1") +
  labs(title = title,
       color = "",
       fill = "",
       x = paste0("PC1: ", round(pcaOutput$pov[1], digits = 2) * 100,
                  "% variance"),
       y = paste0("PC2: ", round(pcaOutput$pov[2], digits = 2) * 100,
                  "% variance"))

} else {

  # if < 10 groups (e.g. the predictor classes) have colors from RColorBrewer
  if (length(unique(pcaOutput2$groups)) <= 10) {

    plot <- ggplot(data = pcaOutput2, aes(x = PC1, y = PC2,
                                             group = groups,
                                             color = groups)) +
      geom_point(size = 2, alpha = 0.6) +
      scale_color_brewer(palette = "Set1") +
      labs(title = title,
           color = "",
           fill = "",
           x = paste0("PC1: ", round(pcaOutput$pov[1], digits = 2) * 100,
                      "% variance"),
           y = paste0("PC2: ", round(pcaOutput$pov[2], digits = 2) * 100,
                      "% variance"))

  } else {
    # otherwise use the default rainbow colors
    plot <- ggplot(data = pcaOutput2, aes(x = PC1, y = PC2,
                                             group = groups, color = groups)) +
      geom_point(size = 2, alpha = 0.6) +
      labs(title = title,
           color = "",
           fill = "",
           x = paste0("PC1: ", round(pcaOutput$pov[1], digits = 2) * 100,
                      "% variance"),
           y = paste0("PC2: ", round(pcaOutput$pov[2], digits = 2) * 100,
                      "% variance"))
  }
}

```

```

    return(plot)

}

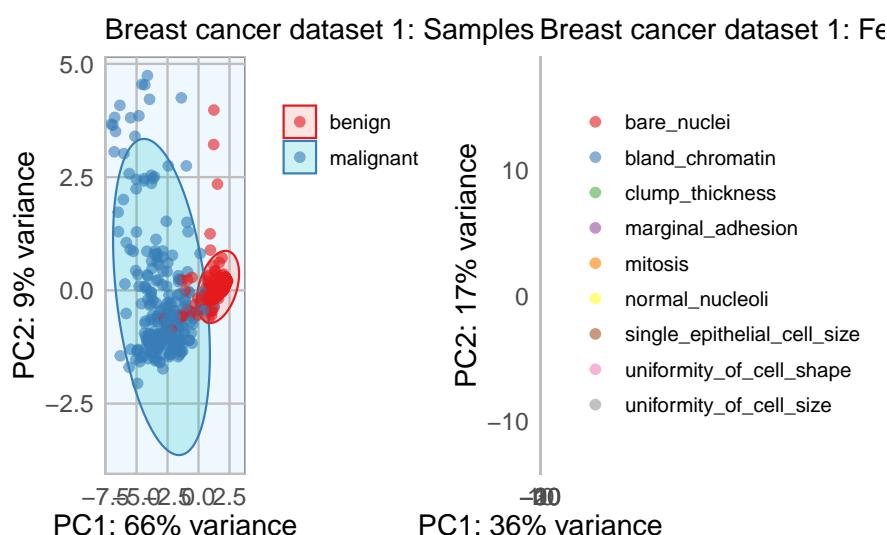
library(gridExtra)
#>
#> Attaching package: 'gridExtra'
#> The following object is masked from 'package:BiocGenerics':
#>
#>     combine
library(grid)

# plot all data. one row is a feature
p1 <- pca_func(data = t(bc_data[, 2:10]),
                 groups = as.character(bc_data$classes),
                 title = "Breast cancer dataset 1: Samples")

# plot features only. features as columns
p2 <- pca_func(data = bc_data[, 2:10],
                 groups = as.character(colnames(bc_data[, 2:10])),
                 title = "Breast cancer dataset 1: Features", print_ellipse = FALSE)

grid.arrange(p1, p2, ncol = 2)

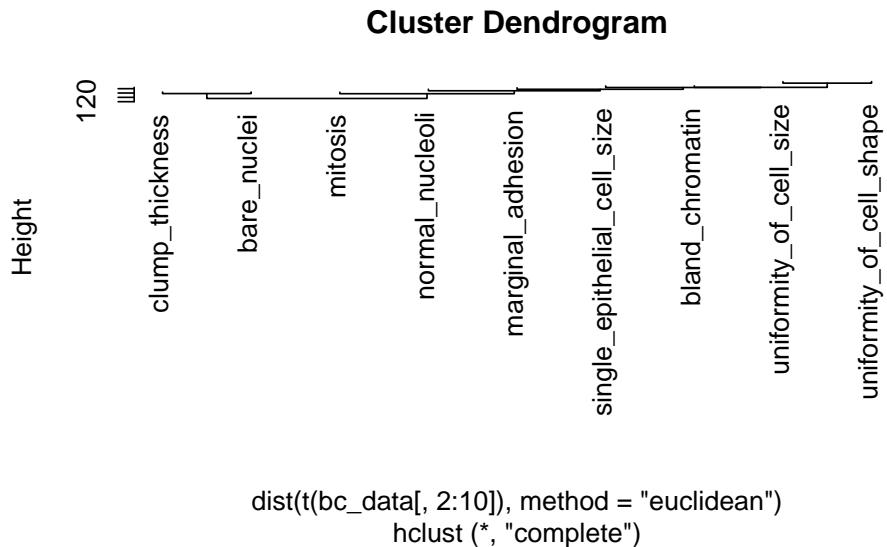
```



```

h_1 <- hclust(dist(t(bc_data[, 2:10])), method = "euclidean", method = "complete")
plot(h_1)

```



17.2.1 density plots vs class

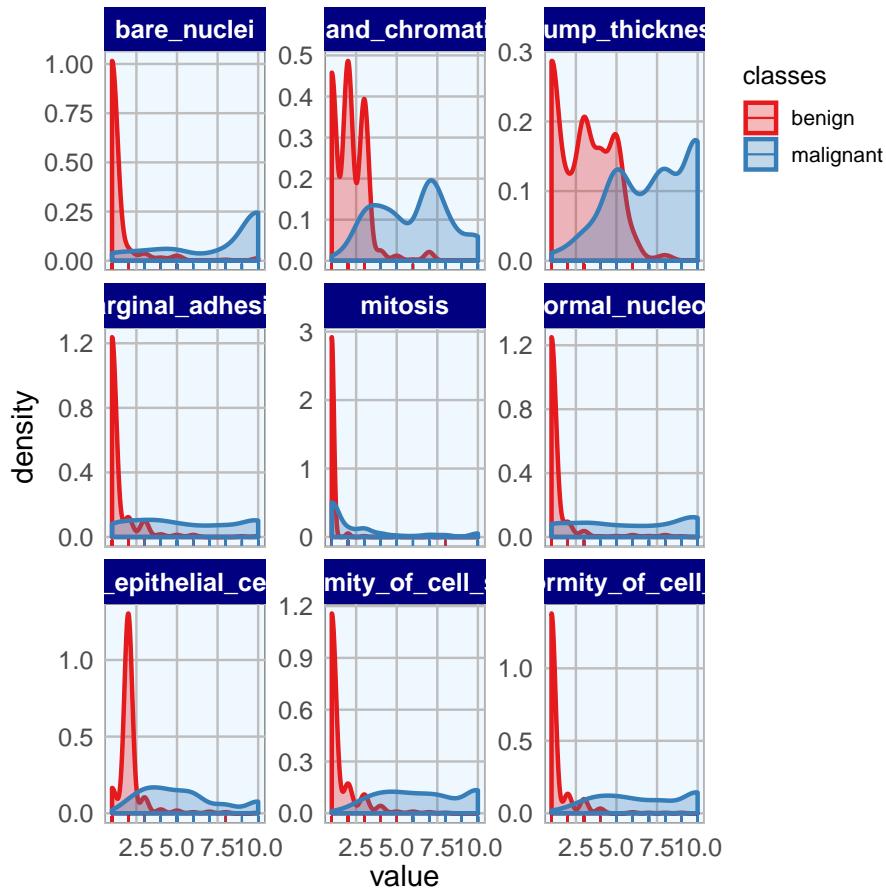
```

# density plot showing the feature vs classes
library(tidyr)
#>
## Attaching package: 'tidyr'
## The following object is masked from 'package:S4Vectors':
#>
#>     expand
## The following object is masked from 'package:mice':
#>
#>     complete

# gather data. from column clump_thickness to mitosis
bc_data_gather <- bc_data %>%
  gather(measure, value, clump_thickness:mitosis)

ggplot(data = bc_data_gather, aes(x = value, fill = classes, color = classes)) +
  geom_density(alpha = 0.3, size = 1) +
  geom_rug() +
  scale_fill_brewer(palette = "Set1") +
  scale_color_brewer(palette = "Set1") +
  facet_wrap(~ measure, scales = "free_y", ncol = 3)

```



17.3 Feature importance

To get an idea about the feature's respective importances, I'm running Random Forest models with 10×10 cross validation using the `caret` package. If I wanted to use feature importance to select features for modeling, I would need to perform it on the training data instead of on the complete dataset. But here, I only want to use it to get acquainted with my data. I am again defining a function that estimates the feature importance and produces a plot.

```
library(caret)
# library(doParallel) # parallel processing
# registerDoParallel()

# prepare training scheme
control <- trainControl(method = "repeatedcv", number = 10, repeats = 10)

feature_imp <- function(model, title) {
  # estimate variable importance
  importance <- varImp(model, scale = TRUE)
  # prepare dataframes for plotting
  importance_df_1 <- importance$importance
  importance_df_1$group <- rownames(importance_df_1)

  importance_df_2 <- importance_df_1
  importance_df_2$Overall <- 0
```

```

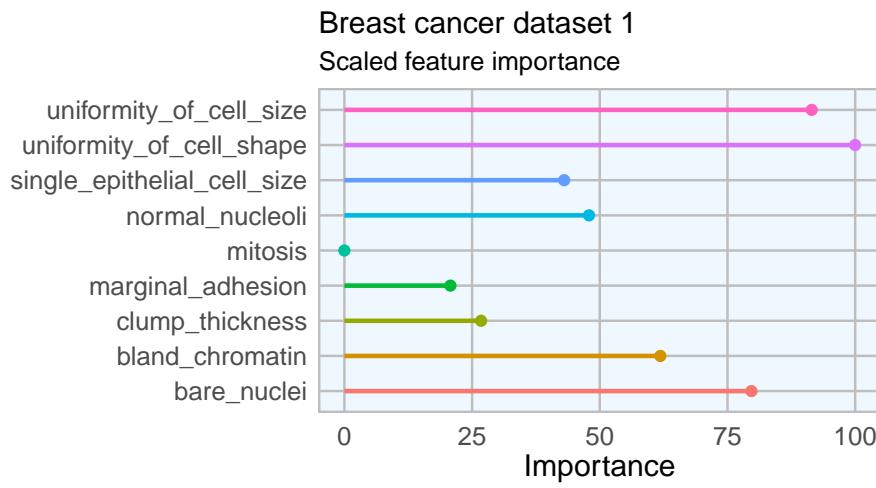
importance_df <- rbind(importance_df_1, importance_df_2)

plot <- ggplot() +
  geom_point(data = importance_df_1, aes(x = Overall,
                                         y = group,
                                         color = group), size = 2) +
  geom_path(data = importance_df, aes(x = Overall,
                                       y = group,
                                       color = group,
                                       group = group), size = 1) +
  theme(legend.position = "none") +
  labs(
    x = "Importance",
    y = "",
    title = title,
    subtitle = "Scaled feature importance",
    caption = "\nDetermined with Random Forest and
    repeated cross validation (10 repeats, 10 times)"
  )
return(plot)
}

# train the model
set.seed(27)
imp_1 <- train(classes ~ ., data = bc_data, method = "rf",
                preProcess = c("scale", "center"),
                trControl = control)

p1 <- feature_imp(imp_1, title = "Breast cancer dataset 1")
p1

```



17.4 Feature Selection

1. By correlation

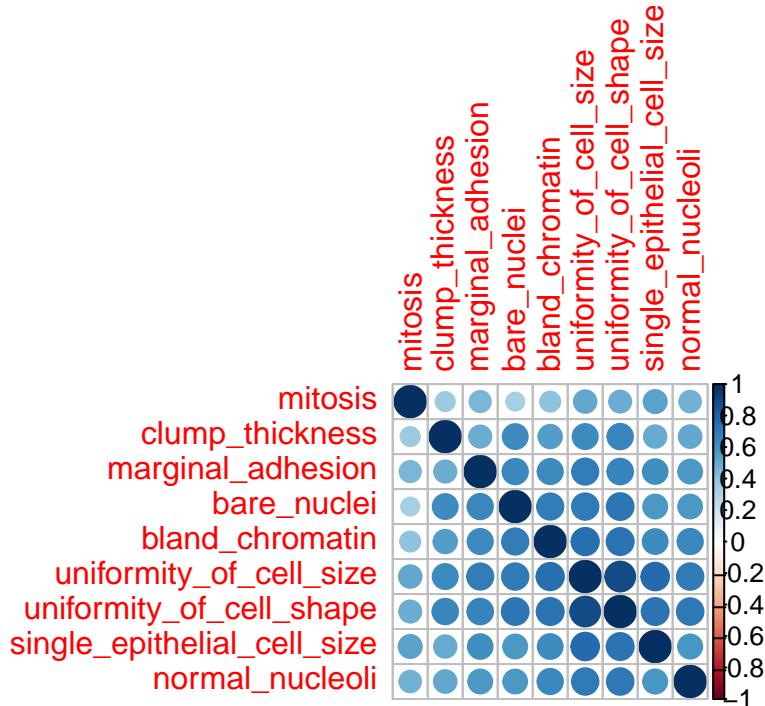
2. By Recursive Feature Elimination
3. By Genetic Algorithm

```
set.seed(27)
bc_data_index <- createDataPartition(bc_data$classes, p = 0.7, list = FALSE)
bc_data_train <- bc_data[bc_data_index, ]
bc_data_test <- bc_data[-bc_data_index, ]
```

17.4.1 Correlation

```
library(corrplot)
#> corrplot 0.84 loaded

# calculate correlation matrix
corMatMy <- cor(bc_data_train[, -1])
corrplot(corMatMy, order = "hclust")
```



```
# Apply correlation filter at 0.70,
highlyCor <- colnames(bc_data_train[, -1])[findCorrelation(corMatMy,
cutoff = 0.7,
verbose = TRUE)]
```

#> Compare row 2 and column 3 with corr 0.9
#> Means: 0.709 vs 0.595 so flagging column 2
#> Compare row 3 and column 7 with corr 0.737
#> Means: 0.674 vs 0.572 so flagging column 3
#> All correlations <= 0.7

```
# which variables are flagged for removal?
highlyCor
#> [1] "uniformity_of_cell_size" "uniformity_of_cell_shape"
```

```
# then we remove these variables
bc_data_cor <- bc_data_train[, which(!colnames(bc_data_train) %in% highlyCor)]
names(bc_data_cor)
#> [1] "classes"                      "clump_thickness"
#> [3] "marginal_adhesion"           "single_epithelial_cell_size"
#> [5] "bare_nuclei"                  "bland_chromatin"
#> [7] "normal_nucleoli"             "mitosis"

# confirm features were removed
outersect <- function(x, y) {
  sort(c(setdiff(x, y),
         setdiff(y, x)))
}

outersect(names(bc_data_cor), names(bc_data_train))
#> [1] "uniformity_of_cell_shape" "uniformity_of_cell_size"
```

Four features removed

17.4.2 Recursive Feature Elimination (RFE)

```
# ensure the results are repeatable
set.seed(7)

# define the control using a random forest selection function with cross validation
control <- rfeControl(functions = rfFuncs, method = "cv", number = 10)

# run the RFE algorithm
results_1 <- rfe(x = bc_data_train[, -1],
                  y = bc_data_train$classes,
                  sizes = c(1:9),
                  rfeControl = control)

# chosen features
predictors(results_1)
#> [1] "bare_nuclei"                  "clump_thickness"
#> [3] "normal_nucleoli"              "uniformity_of_cell_size"
#> [5] "uniformity_of_cell_shape"     "single_epithelial_cell_size"
#> [7] "bland_chromatin"              "marginal_adhesion"

# subset the chosen features
sel_cols <- which(colnames(bc_data_train) %in% predictors(results_1))
bc_data_rfe <- bc_data_train[, c(1, sel_cols)]
names(bc_data_rfe)
#> [1] "classes"                      "clump_thickness"
#> [3] "uniformity_of_cell_size"       "uniformity_of_cell_shape"
#> [5] "marginal_adhesion"            "single_epithelial_cell_size"
#> [7] "bare_nuclei"                  "bland_chromatin"
#> [9] "normal_nucleoli"              "mitosis"

# confirm features removed by RFE
outersect(names(bc_data_rfe), names(bc_data_train))
#> [1] "mitosis"
```

No features removed with RFE

17.4.3 Genetic Algorithm (GA)

```

library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following object is masked from 'package:gridExtra':
#>
#>     combine
#> The following objects are masked from 'package:Biostrings':
#>
#>     collapse, intersect, setdiff, setequal, union
#> The following object is masked from 'package:XVector':
#>
#>     slice
#> The following objects are masked from 'package:IRanges':
#>
#>     collapse, desc, intersect, setdiff, slice, union
#> The following objects are masked from 'package:S4Vectors':
#>
#>     first, intersect, rename, setdiff, setequal, union
#> The following objects are masked from 'package:BiocGenerics':
#>
#>     combine, intersect, setdiff, union
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union

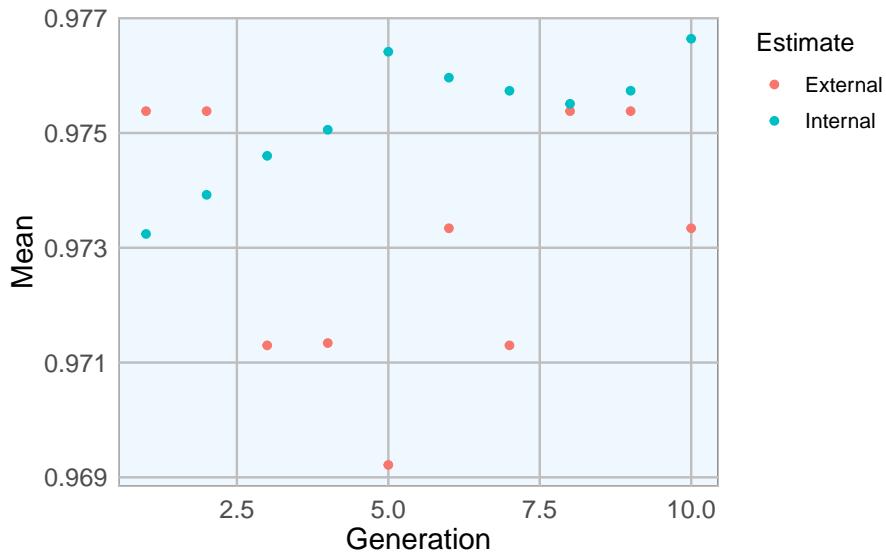
ga_ctrl <- gafsControl(functions = rfGA, # Assess fitness with RF
                        method = "cv",      # 10 fold cross validation
                        genParallel = TRUE, # Use parallel programming
                        allowParallel = TRUE)

lev <- c("malignant", "benign")      # Set the levels

set.seed(27)
model_1 <- gafs(x = bc_data_train[, -1], y = bc_data_train$classes,
                 iters = 10, # generations of algorithm
                 popSize = 5, # population size for each generation
                 levels = lev,
                 gafsControl = ga_ctrl)
#>
#> Attaching package: 'recipes'
#> The following object is masked from 'package:stats':
#>
#>     step

plot(model_1) # Plot mean fitness (AUC) by generation

```



```

# features
model_1$ga$final
#> [1] "clump_thickness"           "uniformity_of_cell_size"
#> [3] "uniformity_of_cell_shape" "marginal_adhesion"
#> [5] "bare_nuclei"              "bland_chromatin"
#> [7] "normal_nucleoli"

# select features
sel_cols_ga <- which(colnames(bc_data_train) %in% model_1$ga$final)
bc_data_ga <- bc_data_train[, c(1, sel_cols_ga)]
names(bc_data_ga)
#> [1] "classes"                  "clump_thickness"
#> [3] "uniformity_of_cell_size" "uniformity_of_cell_shape"
#> [5] "marginal_adhesion"       "bare_nuclei"
#> [7] "bland_chromatin"         "normal_nucleoli"

# features removed GA
outersect(names(bc_data_ga), names(bc_data_train))
#> [1] "mitosis"                  "single_epithelial_cell_size"

```

Two features removed with GA:

17.5 Model comparison

17.5.1 Using all features

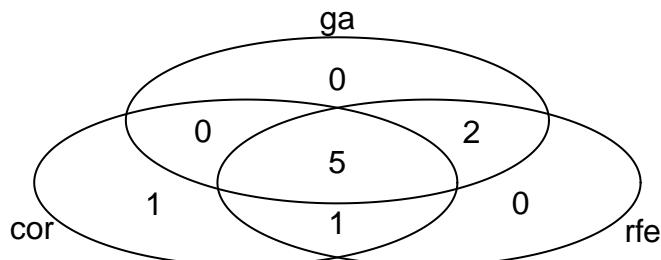
```
# confusion matrix
cm_all_1 <- confusionMatrix(predict(model_bc_data_all, bc_data_test[, -1]), bc_data_test$classes)
cm_all_1
#> Confusion Matrix and Statistics
#>
#>           Reference
#> Prediction benign malignant
#>   benign        131         2
#>   malignant      6        70
#>
#>           Accuracy : 0.962
#>           95% CI  : (0.926, 0.983)
#>   No Information Rate : 0.656
#>   P-Value [Acc > NIR] : <2e-16
#>
#>           Kappa : 0.916
#>
#> Mcnemar's Test P-Value : 0.289
#>
#>           Sensitivity : 0.956
#>           Specificity  : 0.972
#>           Pos Pred Value : 0.985
#>           Neg Pred Value : 0.921
#>           Prevalence    : 0.656
#>           Detection Rate : 0.627
#>           Detection Prevalence : 0.636
#>           Balanced Accuracy : 0.964
#>
#>   'Positive' Class : benign
#>
```

17.5.2 Compare selection methods

```
# compare features selected by the three methods
library(gplots)
#>
#> Attaching package: 'gplots'
#> The following object is masked from 'package:IRanges':
#>
#>   space
#> The following object is masked from 'package:S4Vectors':
#>
#>   space
#> The following object is masked from 'package:stats':
#>
#>   lowess

venn_list <- list(cor = colnames(bc_data_cor)[-1],
                  rfe = colnames(bc_data_rfe)[-1],
                  ga  = colnames(bc_data_ga)[-1])

venn <- venn(venn_list)
```



```

venn
#>      num cor rfe ga
#> 000    0   0   0   0
#> 001    0   0   0   1
#> 010    0   0   1   0
#> 011    2   0   1   1
#> 100    1   1   0   0
#> 101    0   1   0   1
#> 110    1   1   1   0
#> 111    5   1   1   1
#> attr("intersections")
#> attr("intersections")$`cor:rfe:ga`
#> [1] "clump_thickness"   "marginal_adhesion" "bare_nuclei"
#> [4] "bland_chromatin"    "normal_nucleoli"
#>
#> attr("intersections")$`cor`
#> [1] "mitosis"
#>
#> attr("intersections")$`rfe:ga`
#> [1] "uniformity_of_cell_size"  "uniformity_of_cell_shape"
#>
#> attr("intersections")$`cor:rfe`
#> [1] "single_epithelial_cell_size"
#>
#> attr("class")
#> [1] "venn"

```

4 out of 10 features were chosen by all three methods; the biggest overlap is seen between GA and RFE with 7 features. RFE and GA both retained 8 features for modeling, compared to only 5 based on the correlation method.

17.5.3 Correlation

```

# correlation
set.seed(127)
model_bc_data_cor <- train(classes ~ .,
                           data = bc_data_cor,
                           method = "rf",
                           preProcess = c("scale", "center"),
                           trControl = trainControl(method = "repeatedcv", number = 5, repeats = 10, verboseIter = TRUE))

cm_cor_1 <- confusionMatrix(predict(model_bc_data_cor, bc_data_test[, -1]), bc_data_test$classes)
cm_cor_1
#> Confusion Matrix and Statistics
#>

```

```

#>           Reference
#> Prediction benign malignant
#>   benign       130        4
#>   malignant     7       68
#>
#>           Accuracy : 0.947
#>           95% CI : (0.908, 0.973)
#> No Information Rate : 0.656
#> P-Value [Acc > NIR] : <2e-16
#>
#>           Kappa : 0.885
#>
#> Mcnemar's Test P-Value : 0.546
#>
#>           Sensitivity : 0.949
#>           Specificity : 0.944
#> Pos Pred Value : 0.970
#> Neg Pred Value : 0.907
#> Prevalence : 0.656
#> Detection Rate : 0.622
#> Detection Prevalence : 0.641
#> Balanced Accuracy : 0.947
#>
#> 'Positive' Class : benign
#>

```

17.5.4 Recursive Feature Elimination

```

set.seed(127)
model_bc_data_rfe <- train(classes ~ .,
                             data = bc_data_rfe,
                             method = "rf",
                             preProcess = c("scale", "center"),
                             trControl = trainControl(method = "repeatedcv",
                                                       number = 5, repeats = 10,
                                                       verboseIter = FALSE))

cm_rfe_1 <- confusionMatrix(predict(model_bc_data_rfe, bc_data_test[, -1]), bc_data_test$classes)
cm_rfe_1
#> Confusion Matrix and Statistics
#>
#>           Reference
#> Prediction benign malignant
#>   benign       130        3
#>   malignant     7       69
#>
#>           Accuracy : 0.952
#>           95% CI : (0.914, 0.977)
#> No Information Rate : 0.656
#> P-Value [Acc > NIR] : <2e-16
#>
#>           Kappa : 0.895

```

```
#>
#> Mcnemar's Test P-Value : 0.343
#>
#>           Sensitivity : 0.949
#>           Specificity : 0.958
#> Pos Pred Value : 0.977
#> Neg Pred Value : 0.908
#>      Prevalence : 0.656
#> Detection Rate : 0.622
#> Detection Prevalence : 0.636
#>     Balanced Accuracy : 0.954
#>
#> 'Positive' Class : benign
#>
```

17.5.5 GA

```
set.seed(127)
model_bc_data_ga <- train(classes ~ .,
                           data = bc_data_ga,
                           method = "rf",
                           preProcess = c("scale", "center"),
                           trControl = trainControl(method = "repeatedcv",
                                                     number = 5, repeats = 10,
                                                     verboseIter = FALSE))

cm_ga_1 <- confusionMatrix(predict(model_bc_data_ga, bc_data_test[, -1]), bc_data_test$classes)
cm_ga_1
#> Confusion Matrix and Statistics
#>
#>           Reference
#> Prediction benign malignant
#>   benign          131        2
#>   malignant        6       70
#>
#>           Accuracy : 0.962
#>             95% CI : (0.926, 0.983)
#> No Information Rate : 0.656
#> P-Value [Acc > NIR] : <2e-16
#>
#>           Kappa : 0.916
#>
#> Mcnemar's Test P-Value : 0.289
#>
#>           Sensitivity : 0.956
#>           Specificity : 0.972
#> Pos Pred Value : 0.985
#> Neg Pred Value : 0.921
#>      Prevalence : 0.656
#> Detection Rate : 0.627
#> Detection Prevalence : 0.636
#>     Balanced Accuracy : 0.964
```

```
#>
#>      'Positive' Class : benign
#>
```

17.6 Create comparison tables

```
# take "overall" variable only from Confusion Matrix
overall <- data.frame(dataset = 1,
                       model = rep(c("all", "cor", "rfe", "ga"), 1),
                       rbind(cm_all_1$overall,
                             cm_cor_1$overall,
                             cm_rfe_1$overall,
                             cm_ga_1$overall))
)

# convert to tidy data
library(tidyr)
overall_gather <- overall[, 1:4] %>%      # take the first columns:
  gather(measure, value, Accuracy:Kappa) # dataset, model, Accuracy and Kappa

# take "byClass" variable only from Confusion Matrix
byClass <- data.frame(dataset = 1,
                       model = rep(c("all", "cor", "rfe", "ga"), 1),
                       rbind(cm_all_1$byClass,
                             cm_cor_1$byClass,
                             cm_rfe_1$byClass,
                             cm_ga_1$byClass))
)

# convert to tidy data
byClass_gather <- byClass[, c(1:4, 7)] %>%      # select columns: dataset, model
  gather(measure, value, Sensitivity:Precision) # Sensitiv, Specific, Precis

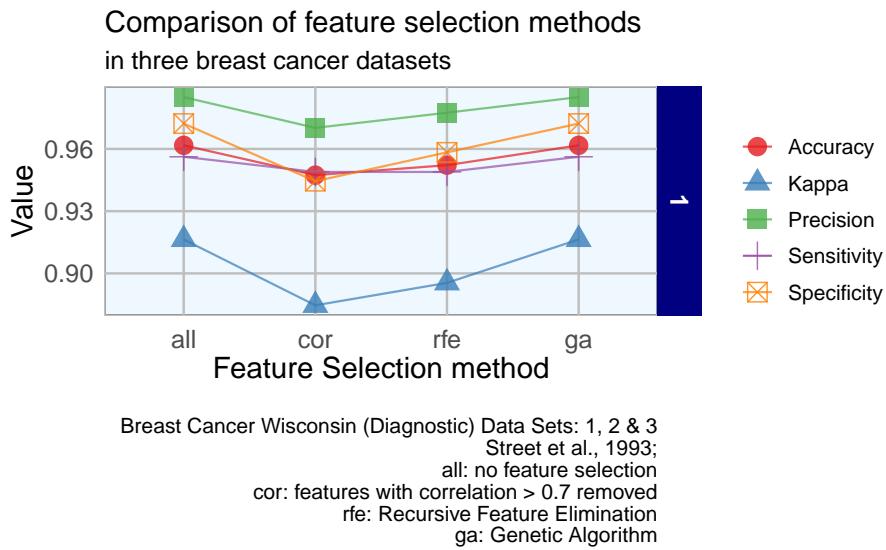
# join the two tables
overall_byClass_gather <- rbind(overall_gather, byClass_gather)
overall_byClass_gather <- within(
  overall_byClass_gather, model <- factor(model,
                                             levels = c("all", "cor", "rfe", "ga")))
  # convert to factor

ggplot(overall_byClass_gather, aes(x = model, y = value, color = measure,
                                     shape = measure, group = measure)) +
  geom_point(size = 4, alpha = 0.8) +
  geom_path(alpha = 0.7) +
  scale_colour_brewer(palette = "Set1") +
  facet_grid(dataset ~ ., scales = "free_y") +
  labs(
    x = "Feature Selection method",
    y = "Value",
    color = "",
    shape = "",
    title = "Comparison of feature selection methods",
```

```

    subtitle = "in three breast cancer datasets",
    caption = "\nBreast Cancer Wisconsin (Diagnostic) Data Sets: 1, 2 & 3
Street et al., 1993;
all: no feature selection
cor: features with correlation > 0.7 removed
rfe: Recursive Feature Elimination
ga: Genetic Algorithm"
)

```



1. Less accurate: selection of features by correlation
2. More accurate: genetic algorithm
3. Including all features is more accurate to removing features by correlation.

17.7 Notes

`pcaGoPromoter` is a BioConductor package. Its dependencies are `BioGenerics`, `AnnotationDbi` and `BioStrings`, which at their turn require `DBI` and `RSQLite` packages from CRAN. Install first those from CRAN, and then move to install `pcaGoPromoter`.

Chapter 18

Titanic with Naive-Bayes Classifier

The Titanic dataset in R is a table for about 2200 passengers summarised according to four factors – economic status ranging from 1st class, 2nd class, 3rd class and crew; gender which is either male or female; Age category which is either Child or Adult and whether the type of passenger survived. For each combination of Age, Gender, Class and Survived status, the table gives the number of passengers who fall into the combination. We will use the Naive Bayes Technique to classify such passengers and check how well it performs.

```
#Getting started with Naive Bayes
#Install the package
#install.packages("e1071")
#Loading the library
library(e1071)

#Next load the Titanic dataset
data("Titanic")
#Save into a data frame and view it
Titanic_df = as.data.frame(Titanic)
```

We see that there are 32 observations which represent all possible combinations of Class, Sex, Age and Survived with their frequency. Since it is summarised, this table is not suitable for modelling purposes. We need to expand the table into individual rows. Let's create a repeating sequence of rows based on the frequencies in the table

```
#Creating data from table
repeating_sequence=rep.int(seq_len(nrow(Titanic_df)), Titanic_df$Freq) #This will repeat each combination

# Create the dataset by row repetition created
Titanic_dataset=Titanic_df[repeating_sequence,]

# We no longer need the frequency, drop the feature
Titanic_dataset$Freq=NULL
```

The data is now ready for Naive Bayes to process. Let's fit the model

```
# Fitting the Naive Bayes model
Naive_Bayes_Model=naiveBayes(Survived ~., data=Titanic_dataset)

# What does the model say? Print the model summary
Naive_Bayes_Model
#>
```

```
#> Naive Bayes Classifier for Discrete Predictors
#>
#> Call:
#> naiveBayes.default(x = X, y = Y, laplace = laplace)
#>
#> A-priori probabilities:
#> Y
#>   No    Yes
#> 0.677 0.323
#>
#> Conditional probabilities:
#>   Class
#> Y      1st    2nd    3rd   Crew
#>   No  0.0819 0.1121 0.3544 0.4517
#>   Yes 0.2855 0.1660 0.2504 0.2982
#>
#>   Sex
#> Y      Male Female
#>   No  0.9154 0.0846
#>   Yes 0.5162 0.4838
#>
#>   Age
#> Y      Child Adult
#>   No  0.0349 0.9651
#>   Yes 0.0802 0.9198
```

The model creates the conditional probability for each feature separately. We also have the a-priori probabilities which indicates the distribution of our data. Let's calculate how we perform on the data.

```
# Prediction on the dataset
NB_Predictions=predict(Naive_Bayes_Model,Titanic_dataset)
# Confusion matrix to check accuracy
table(NB_Predictions,Titanic_dataset$Survived)
#>
#> NB_Predictions  No  Yes
#>           No 1364 362
#>           Yes 126 349
```

We have the results! We are able to classify 1364 out of 1490 “No” cases correctly and 349 out of 711 “Yes” cases correctly. This means the ability of Naive Bayes algorithm to predict “No” cases is about 91.5% but it falls down to only 49% of the “Yes” cases resulting in an overall accuracy of 77.8%

Chapter 19

Can we Do any Better?

Naive Bayes is a parametric algorithm which implies that you cannot perform differently in different runs as long as the data remains the same. We will, however, learn another implementation of Naive Bayes algorithm using the ‘mlr’ package. Assuming the same session is going on for the readers, I will install and load the package and start fitting a model

```
# Getting started with Naive Bayes in mlr
# install.packages("mlr")
# Loading the library
library(mlr)
#> Loading required package: ParamHelpers
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures    rlang
#>   c.quosures    rlang
#>   print.quosures rlang
#>
#> Attaching package: 'mlr'
#> The following object is masked from 'package:e1071':
#>
#>   impute
```

The mlr package consists of a lot of models and works by creating tasks and learners which are then trained. Let’s create a classification task using the titanic dataset and fit a model with the naive bayes algorithm.

```
# Create a classification task for learning on Titanic Dataset and specify the target feature
task = makeClassifTask(data = Titanic_dataset, target = "Survived")

# Initialize the Naive Bayes classifier
selected_model = makeLearner("classif.naiveBayes")

# Train the model
NB_mlr = train(selected_model, task)
```

The summary of the model which was printed in e3071 package is stored in learner model. Let’s print it and compare

```
# Read the model learned
NB_mlr$learner.model
#>
#> Naive Bayes Classifier for Discrete Predictors
```

```

#>
#> Call:
#> naiveBayes.default(x = X, y = Y, laplace = laplace)
#>
#> A-priori probabilities:
#> Y
#>   No Yes
#> 0.677 0.323
#>
#> Conditional probabilities:
#>   Class
#> Y      1st 2nd 3rd Crew
#> No 0.0819 0.1121 0.3544 0.4517
#> Yes 0.2855 0.1660 0.2504 0.2982
#>
#>   Sex
#> Y      Male Female
#> No 0.9154 0.0846
#> Yes 0.5162 0.4838
#>
#>   Age
#> Y      Child Adult
#> No 0.0349 0.9651
#> Yes 0.0802 0.9198

```

The a-priori probabilities and the conditional probabilities for the model are similar to the one calculated by e3071 package as was expected. This means that our predictions will also be the same.

```

# Predict on the dataset without passing the target feature
predictions_mlr = as.data.frame(predict(NB_mlr, newdata = Titanic_dataset[,1:3]))

## Confusion matrix to check accuracy
table(predictions_mlr[,1], Titanic_dataset$Survived)
#>
#>   No Yes
#> No 1364 362
#> Yes 126 349

```

As we see, the predictions are exactly same. The only way to improve is to have more features or more data. Perhaps, if we have more features such as the exact age, size of family, number of parents in the ship and siblings then we may arrive at a better model using Naive Bayes. In essence, Naive Bayes has an advantage of a strong foundation build and is very robust. I know of the ‘caret’ package which also consists of Naive Bayes function but it will also give us the same predictions and probability.

Chapter 20

Building a Naive Bayes Classifier in R

<https://www.machinelearningplus.com/predictive-modeling/how-naive-bayes-algorithm-works-with-example-and-full-code/>

20.1 8. Building a Naive Bayes Classifier in R

Understanding Naive Bayes was the (slightly) tricky part. Implementing it is fairly straightforward.

In R, Naive Bayes classifier is implemented in packages such as `e1071`, `klaR` and `bnlearn`. In Python, it is implemented in `scikit-learn`.

For sake of demonstration, let's use the standard iris dataset to predict the Species of flower using 4 different features: Sepal.Length, Sepal.Width, Petal.Length, Petal.Width

```
# Import Data
training <- read.csv('https://raw.githubusercontent.com/selva86/datasets/master/iris_train.csv')
test <- read.csv('https://raw.githubusercontent.com/selva86/datasets/master/iris_test.csv')
```

The training data is now contained in training and test data in test dataframe. Lets load the `klaR` package and build the naive bayes model.

```
# Using klaR for Naive Bayes
library(klaR)
#> Loading required package: MASS
nb_mod <- NaiveBayes(Species ~ ., data=training)
pred <- predict(nb_mod, test)
```

Lets see the confusion matrix.

```
# Confusion Matrix
tab <- table(pred$class, test$Species)
caret::confusionMatrix(tab)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures    rlang
#>   c.quosures    rlang
#>   print.quosures rlang
#> Confusion Matrix and Statistics
#>
#>
#>           setosa versicolor virginica
```

```

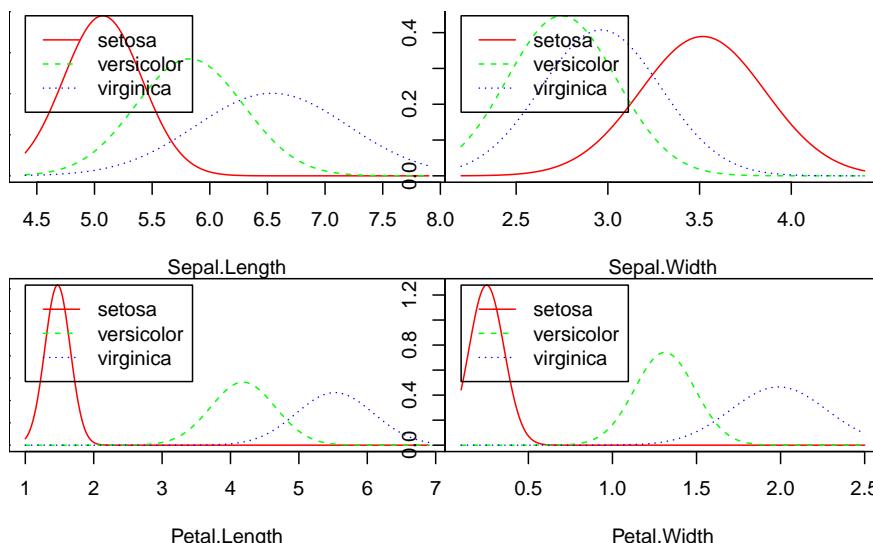
#>   setosa      15      0      0
#>   versicolor    0     11      0
#>   virginica     0      4     15
#>
#> Overall Statistics
#>
#>           Accuracy : 0.911
#>           95% CI : (0.788, 0.975)
#>           No Information Rate : 0.333
#>           P-Value [Acc > NIR] : 8.47e-16
#>
#>           Kappa : 0.867
#>
#> McNemar's Test P-Value : NA
#>
#> Statistics by Class:
#>
#>           Class: setosa Class: versicolor Class: virginica
#> Sensitivity          1.000          0.733          1.000
#> Specificity          1.000          1.000          0.867
#> Pos Pred Value       1.000          1.000          0.789
#> Neg Pred Value       1.000          0.882          1.000
#> Prevalence           0.333          0.333          0.333
#> Detection Rate       0.333          0.244          0.333
#> Detection Prevalence 0.333          0.244          0.422
#> Balanced Accuracy    1.000          0.867          0.933

```

```

# Plot density of each feature using nb_mod
opar = par(mfrow=c(2, 2), mar=c(4,0,0,0))
plot(nb_mod, main="")
par(opar)

```

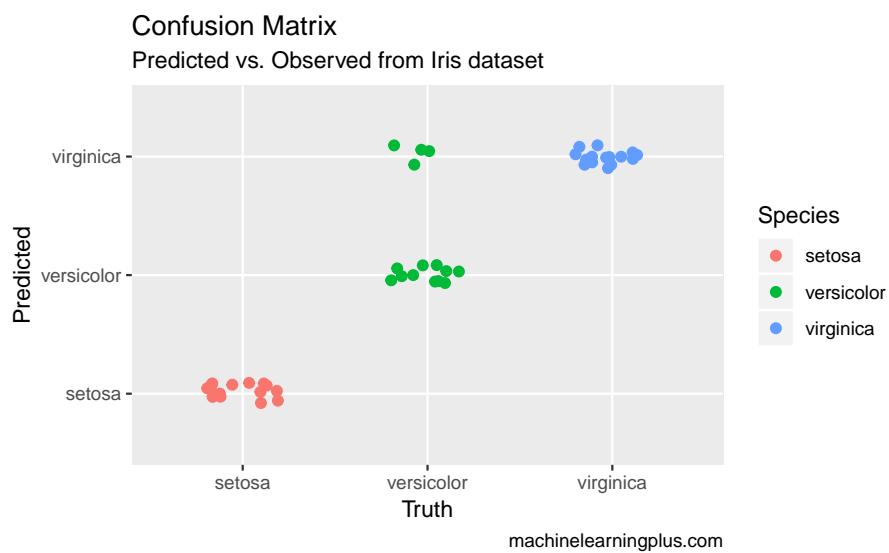


```

# Plot the Confusion Matrix
library(ggplot2)
test$pred <- pred$class
ggplot(test, aes(Species, pred, color = Species)) +

```

```
geom_jitter(width = 0.2, height = 0.1, size=2) +  
  labs(title="Confusion Matrix",  
        subtitle="Predicted vs. Observed from Iris dataset",  
        y="Predicted",  
        x="Truth",  
        caption="machinelearningplus.com")
```



Part III

Feature Engineering

Chapter 21

Employee attrition. Employee-Attrition dataset. *LIME* package

Article: https://www.business-science.io/business/2017/09/18/hr_employee_attrition.html Data: <https://www.ibm.com/communities/analytics/watson-analytics-blog/hr-employee-attrition/>

21.1 Introduction

21.1.1 Employee attrition: a major problem

Bill Gates was once quoted as saying,

“You take away our top 20 employees and we [Microsoft] become a mediocre company”.

His statement cuts to the core of a major problem: employee attrition. An organization is only as good as its employees, and these people are the true source of its competitive advantage.

Organizations face huge costs resulting from employee turnover. Some costs are tangible such as training expenses and the time it takes from when an employee starts to when they become a productive member. However, the most important costs are intangible. Consider what's lost when a productive employee quits: new product ideas, great project management, or customer relationships.

With advances in machine learning and data science, it's possible to not only predict employee attrition but to understand the key variables that influence turnover. We'll take a look at two cutting edge techniques:

1. Machine Learning with `h2o.automl()` from the `h2o` package: This function takes automated machine learning to the next level by testing a number of advanced algorithms such as random forests, ensemble methods, and deep learning along with more traditional algorithms such as logistic regression. The main takeaway is that we can now easily achieve predictive performance that is in the same ball park (and in some cases even better than) commercial algorithms and ML/AI software.
2. Feature Importance with the `lime` package: The problem with advanced machine learning algorithms such as deep learning is that it's near impossible to understand the algorithm because of its complexity. This has all changed with the `lime` package. The major advancement with `lime` is that, by recursively analyzing the models locally, it can extract feature importance that repeats globally. What this means to us is that `lime` has opened the door to understanding the ML models regardless of complexity. Now

the best (and typically very complex) models can also be investigated and potentially understood as to what variables or features make the model tick.

21.1.2 Employee attrition: machine learning analysis

With these new automated ML tools combined with tools to uncover critical variables, we now have capabilities for both extreme predictive accuracy and understandability, which was previously impossible! We'll investigate an HR Analytic example of employee attrition that was evaluated by IBM Watson.

21.1.3 Where we got the data

The example comes from IBM Watson Analytics website. You can download the data and read the analysis here:

Get data used in this post here. Read IBM Watson Analytics article here. To summarize, the article makes a usage case for IBM Watson as an automated ML platform. The article shows that using Watson, the analyst was able to detect features that led to increased probability of attrition.

21.1.4 Automated machine learning (what we did with the data)

In this example we'll show how we can use the combination of H2O for developing a complex model with high predictive accuracy on unseen data and then how we can use LIME to understand important features related to employee attrition.

21.1.5 Load packages

Load the following packages.

```
# Load the following packages
library(tidyquant) # Loads tidyverse and several other pkgs
library(readxl)     # Super simple excel reader
library(h2o)        # Professional grade ML pkg
library(lime)       # Explain complex black-box ML models
```

21.1.6 Load data

Download the data here. You can load the data using `read_excel()`, pointing the path to your local file.

```
# Read excel data
hr_data_raw <- read_excel(path = file.path(data_raw_dir,
                                             "WA_Fn-UseC_-HR-Employee-Attrition.xlsx"))
```

Let's check out the raw data. It's 1470 rows (observations) by 35 columns (features). The "Attrition" column is our target. We'll use all other columns as features to our model.

```
# View first 10 rows
hr_data_raw[1:10,] %>%
  knitr::kable(caption = "First 10 rows")
```

The only pre-processing we'll do in this example is change all character data types to factors. This is needed for H2O. We could make a number of other numeric data that is actually categorical factors, but this tends to increase modeling time and can have little improvement on model performance.

Table 21.1: First 10 rows

Age	Attrition	BusinessTravel	DailyRate	Department	DistanceFromHome	Education	EducationField
41	Yes	Travel_Rarely	1102	Sales	1	2	Life Sciences
49	No	Travel_Frequently	279	Research & Development	8	1	Life Sciences
37	Yes	Travel_Rarely	1373	Research & Development	2	2	Other
33	No	Travel_Frequently	1392	Research & Development	3	4	Life Sciences
27	No	Travel_Rarely	591	Research & Development	2	1	Medical
32	No	Travel_Frequently	1005	Research & Development	2	2	Life Sciences
59	No	Travel_Rarely	1324	Research & Development	3	3	Medical
30	No	Travel_Rarely	1358	Research & Development	24	1	Life Sciences
38	No	Travel_Frequently	216	Research & Development	23	3	Life Sciences
36	No	Travel_Rarely	1299	Research & Development	27	3	Medical

```
hr_data <- hr_data_raw %>%
  mutate_if(is.character, as.factor) %>%
  select(Attrition, everything())
```

Let's take a glimpse at the processed dataset. We can see all of the columns. Note our target ("Attrition") is the first column.

```
glimpse(hr_data)
#> Observations: 1,470
#> Variables: 35
#> $ Attrition
#> $ Age
#> $ BusinessTravel
#> $ DailyRate
#> $ Department
#> $ DistanceFromHome
#> $ Education
#> $ EducationField
#> $ EmployeeCount
#> $ EmployeeNumber
#> $ EnvironmentSatisfaction
#> $ Gender
#> $ HourlyRate
#> $ JobInvolvement
#> $ JobLevel
#> $ JobRole
#> $ JobSatisfaction
#> $ MaritalStatus
#> $ MonthlyIncome
#> $ MonthlyRate
#> $ NumCompaniesWorked
#> $ Over18
#> $ Overtime
#> $ PercentSalaryHike
#> $ PerformanceRating
#> $ RelationshipSatisfaction
#> $ StandardHours
#> $ StockOptionLevel
#> $ TotalWorkingYears
```

<fct> Yes, No, Yes, No, No, No, No, No, No, ...
<dbl> 41, 49, 37, 33, 27, 32, 59, 30, 38, 3...
<fct> Travel_Rarely, Travel_Frequently, Tra...
<dbl> 1102, 279, 1373, 1392, 591, 1005, 132...
<fct> Sales, Research & Development, Resear...
<dbl> 1, 8, 2, 3, 2, 3, 24, 23, 27, 16, ...
<dbl> 2, 1, 2, 4, 1, 2, 3, 1, 3, 3, 3, 2, 1...
<fct> Life Sciences, Life Sciences, Other, ...
<dbl> 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1...
<dbl> 1, 2, 4, 5, 7, 8, 10, 11, 12, 13, 14,...
<dbl> 2, 3, 4, 4, 1, 4, 3, 4, 4, 3, 1, 4, 1...
<fct> Female, Male, Male, Female, Male, Mal...
<dbl> 94, 61, 92, 56, 40, 79, 81, 67, 44, 9...
<dbl> 3, 2, 2, 3, 3, 4, 3, 2, 3, 4, 2, 3...
<dbl> 2, 2, 1, 1, 1, 1, 1, 3, 2, 1, 2, 1...
<fct> Sales Executive, Research Scientist, ...
<dbl> 4, 2, 3, 3, 2, 4, 1, 3, 3, 2, 3, 3...
<fct> Single, Married, Single, Married, Mar...
<dbl> 5993, 5130, 2090, 2909, 3468, 3068, 2...
<dbl> 19479, 24907, 2396, 23159, 16632, 118...
<dbl> 8, 1, 6, 1, 9, 0, 4, 1, 0, 6, 0, 0, 1...
<fct> Y, Y...
<fct> Yes, No, Yes, Yes, No, No, Yes, No, N...
<dbl> 11, 23, 15, 11, 12, 13, 20, 22, 21, 1...
<dbl> 3, 4, 3, 3, 3, 4, 4, 4, 3, 3, 3, 3...
<dbl> 1, 4, 2, 3, 4, 3, 1, 2, 2, 3, 4, 4...
<dbl> 80, 80, 80, 80, 80, 80, 80, 80, 80, 8...
<dbl> 0, 1, 0, 0, 1, 0, 3, 1, 0, 2, 1, 0, 1...
<dbl> 8, 10, 7, 8, 6, 8, 12, 1, 10, 17, 6, ...

```
#> $ TrainingTimesLastYear      <dbl> 0, 3, 3, 3, 3, 2, 3, 2, 2, 3, 5, 3, 1...
#> $ WorkLifeBalance          <dbl> 1, 3, 3, 3, 3, 2, 2, 3, 3, 2, 3, 3, 2...
#> $ YearsAtCompany           <dbl> 6, 10, 0, 8, 2, 7, 1, 1, 9, 7, 5, 9, ...
#> $ YearsInCurrentRole       <dbl> 4, 7, 0, 7, 2, 7, 0, 0, 7, 7, 4, 5, 2...
#> $ YearsSinceLastPromotion  <dbl> 0, 1, 0, 3, 2, 3, 0, 0, 1, 7, 0, 0, 4...
#> $ YearsWithCurrManager     <dbl> 5, 7, 0, 0, 2, 6, 0, 0, 8, 7, 3, 8, 3...
```

21.2 Modeling Employee attrition

We are going to use the `h2o.automl()` function from the H2O platform to model employee attrition.

21.2.1 Machine Learning with h2o

First, we need to initialize the *Java Virtual Machine (JVM)* that H2O uses locally.

```
# Initialize H2O JVM
h2o.init()
#>
#> H2O is not running yet, starting it now...
#>
#> Note: In case of errors look at the following log files:
#>       /tmp/RtmpLPDVN2/h2o_datascience_started_from_r.out
#>       /tmp/RtmpLPDVN2/h2o_datascience_started_from_r.err
#>
#>
#> Starting H2O JVM and connecting: . Connection successful!
#>
#> R is connected to the H2O cluster:
#>   H2O cluster uptime:      1 seconds 130 milliseconds
#>   H2O cluster timezone:    America/Chicago
#>   H2O data parsing timezone: UTC
#>   H2O cluster version:    3.22.1.1
#>   H2O cluster version age: 8 months and 23 days !!!
#>   H2O cluster name:        H2O_started_from_R_datascience_mwl453
#>   H2O cluster total nodes: 1
#>   H2O cluster total memory: 6.96 GB
#>   H2O cluster total cores: 8
#>   H2O cluster allowed cores: 8
#>   H2O cluster healthy:     TRUE
#>   H2O Connection ip:       localhost
#>   H2O Connection port:     54321
#>   H2O Connection proxy:    NA
#>   H2O Internal Security:  FALSE
#>   H2O API Extensions:    XGBoost, Algos, AutoML, Core V3, Core V4
#>   R Version:              R version 3.6.0 (2019-04-26)
h2o.no_progress() # Turn off output of progress bars
```

Next, we change our data to an `h2o` object that the package can interpret. We also split the data into training, validation, and test sets. Our preference is to use 70%, 15%, 15%, respectively.

```
# Split data into Train/Validation/Test Sets
hr_data_h2o <- as.h2o(hr_data)
```

```
split_h2o <- h2o.splitFrame(hr_data_h2o, c(0.7, 0.15), seed = 1234 )
train_h2o <- h2o.assign(split_h2o[[1]], "train" ) # 70%
valid_h2o <- h2o.assign(split_h2o[[2]], "valid" ) # 15%
test_h2o  <- h2o.assign(split_h2o[[3]], "test" ) # 15%
```

21.3 Model

Now we are ready to model. We'll set the target and feature names. The target is what we aim to predict (in our case “Attrition”). The features (every other column) are what we will use to model the prediction.

```
# Set names for h2o
y <- "Attrition"
x <- setdiff(names(train_h2o), y)
```

Now the fun begins. We run the `h2o.automl()` setting the arguments it needs to run models against. For more information, see the `h2o.automl` documentation.

- `x = x`: The names of our feature columns.
- `y = y`: The name of our target column.
- `training_frame = train_h2o`: Our training set consisting of 70% of the data.
- `leaderboard_frame = valid_h2o`: Our validation set consisting of 15% of the data. H2O uses this to ensure the model does not overfit the data.
- `max_runtime_secs = 30`: We supply this to speed up H2O's modeling. The algorithm has a large number of complex models so we want to keep things moving at the expense of some accuracy.

```
# Run the automated machine learning
automl_models_h2o <- h2o.automl(
  x = x,
  y = y,
  training_frame    = train_h2o,
  leaderboard_frame = valid_h2o,
  max_runtime_secs  = 30
)
```

All of the models are stored the `automl_models_h2o` object. However, we are only concerned with the leader, which is the best model in terms of accuracy on the validation set. We'll extract it from the `models` object.

```
# Extract leader model
automl_leader <- automl_models_h2o@leader
```

21.4 Predict

Now we are ready to predict on our test set, which is unseen from during our modeling process. This is the true test of performance. We use the `h2o.predict()` function to make predictions.

```
# Predict on hold-out set, test_h2o
pred_h2o <- h2o.predict(object = automl_leader, newdata = test_h2o)
```

21.5 Performance

Now we can evaluate our leader model. We'll reformat the test set and add the predictions as column so we have the actual and prediction columns side-by-side.

```
# Prep for performance assessment
test_performance <- test_h2o %>%
  tibble::as_tibble() %>%
  select(Attrition) %>%
  add_column(pred = as.vector(pred_h2o$predict)) %>%
  mutate_if(is.character, as.factor)
test_performance
#> # A tibble: 211 x 2
#>   Attrition pred
#>   <fct>     <fct>
#> 1 No        No
#> 2 No        No
#> 3 Yes       Yes
#> 4 No        No
#> 5 No        No
#> 6 No        No
#> # ... with 205 more rows
```

We can use the table() function to quickly get a confusion table of the results. We see that the leader model wasn't perfect, but it did a decent job identifying employees that are likely to quit. For perspective, a logistic regression would not perform nearly this well.

```
# Confusion table counts
confusion_matrix <- test_performance %>%
  table()
confusion_matrix
#>      pred
#> Attrition No Yes
#>       No    163  19
#>       Yes     6  23
```

We'll run through a binary classification analysis to understand the model performance.

```
# Performance analysis
tn <- confusion_matrix[1]
tp <- confusion_matrix[4]
fp <- confusion_matrix[3]
fn <- confusion_matrix[2]

accuracy <- (tp + tn) / (tp + tn + fp + fn)
misclassification_rate <- 1 - accuracy
recall <- tp / (tp + fn)
precision <- tp / (tp + fp)
null_error_rate <- tn / (tp + tn + fp + fn)

tibble(
  accuracy,
  misclassification_rate,
  recall,
```

```

precision,
null_error_rate
) %>%
  transpose()
#> [[1]]
#> [[1]]$accuracy
#> [1] 0.882
#>
#> [[1]]$misclassification_rate
#> [1] 0.118
#>
#> [[1]]$recall
#> [1] 0.793
#>
#> [[1]]$precision
#> [1] 0.548
#>
#> [[1]]$null_error_rate
#> [1] 0.773

```

It is important to understand is that the accuracy can be misleading: 88% sounds pretty good especially for modeling HR data, but if we just pick `Attrition = NO` we would get an accuracy of about 79%. Doesn't sound so great now.

Before we make our final judgement, let's dive a little deeper into precision and recall. Precision is when the model predicts yes, how often is it actually yes. Recall (also true positive rate or specificity) is when the actual value is yes how often is the model correct. Confused yet? Let's explain in terms of what's important to HR.

Most HR groups would probably prefer to incorrectly classify folks not looking to quit as high potential of quitting rather than classify those that are likely to quit as not at risk. Because it's important to not miss at risk employees, HR will really care about recall or when the actual value is `Attrition = YES` how often the model predicts YES.

Recall for our model is 62%. In an HR context, this is 62% more employees that could potentially be targeted prior to quitting. From that standpoint, an organization that loses 100 people per year could possibly target 62 implementing measures to retain.

21.6 The lime package

We have a very good model that is capable of making very accurate predictions on unseen data, but what can it tell us about what causes attrition? Let's find out using LIME.

21.6.1 Set up

The `lime` package implements LIME in R. One thing to note is that it's not setup out-of-the-box to work with `h2o`. The good news is with a few functions we can get everything working properly. We'll need to make two custom functions:

- `model_type`: Used to tell lime what type of model we are dealing with. It could be classification, regression, survival, etc.
- `predict_model`: Used to allow lime to perform predictions that its algorithm can interpret.

The first thing we need to do is identify the class of our model leader object. We do this with the `class()` function.

```
class(automl_leader)
#> [1] "H2OBinomialModel"
#> attr(package)
#> [1] "h2o"
```

Next we create our `model_type` function. It's only input is `x` the h2o model. The function simply returns "classification", which tells LIME we are classifying.

```
# Setup lime::model_type() function for h2o
model_type.H2OBinomialModel <- function(x, ...) {
  # Function tells lime() what model type we are dealing with
  # 'classification', 'regression', 'survival', 'clustering', 'multilabel', etc
  #
  # x is our h2o model

  return("classification")
}
```

Now we can create our `predict_model` function. The trick here is to realize that it's inputs must be `x` a model, `newdata` a dataframe object (this is important), and `type` which is not used but can be used to switch the output type. The output is also a little tricky because it must be in the format of probabilities by classification (this is important; shown next). Internally we just call the `h2o.predict()` function.

```
# Setup lime::predict_model() function for h2o
predict_model.H2OBinomialModel <- function(x, newdata, type, ...) {
  # Function performs prediction and returns data frame with Response
  #
  # x is h2o model
  # newdata is data frame
  # type is only setup for data frame

  pred <- h2o.predict(x, as.h2o(newdata))

  # return probs
  return(as.data.frame(pred[, -1]))
}

}
```

Run this next script to show you what the output looks like and to test our `predict_model` function. See how it's the probabilities by classification. It must be in this form for `model_type = "classification"`.

```
# Test our predict_model() function
predict_model(x = automl_leader, newdata = as.data.frame(test_h2o[, -1]), type = 'raw') %>%
  tibble::as_tibble()
#> # A tibble: 211 x 2
#>       No     Yes
#>   <dbl>   <dbl>
#> 1 0.807  0.193
#> 2 0.958  0.0423
#> 3 0.0448 0.955
#> 4 0.959  0.0411
#> 5 0.860  0.140
#> 6 0.965  0.0350
```

```
#> # ... with 205 more rows
```

Now the fun part, we create an explainer using the `lime()` function. Just pass the training data set without the “Attribution column”. The form must be a data frame, which is OK since our `predict_model` function will switch it to an `h2o` object. Set `model = automl_leader` our leader model, and `bin_continuous = FALSE`. We could tell the algorithm to bin continuous variables, but this may not make sense for categorical numeric data that we didn’t change to factors.

```
# Run lime() on training set
explainer <- lime::lime(
  as.data.frame(train_h2o[,-1]),
  model      = automl_leader,
  bin_continuous = FALSE)
#> Warning: Data contains numeric columns with zero variance
```

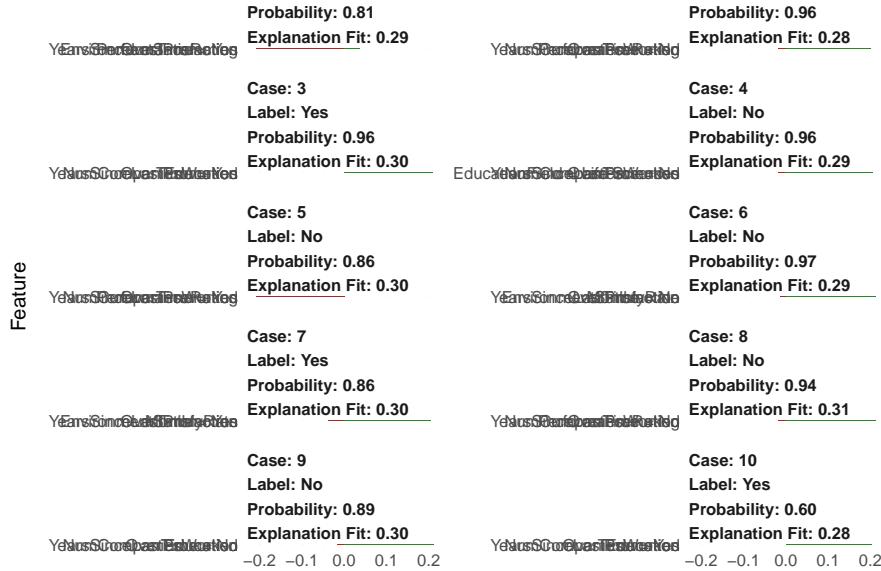
Now we run the `explain()` function, which returns our explanation. This can take a minute to run so we limit it to just the first ten rows of the test data set. We set `n_labels = 1` because we care about explaining a single class. Setting `n_features = 4` returns the top four features that are critical to each case. Finally, setting `kernel_width = 0.5` allows us to increase the “`model_r2`” value by shrinking the localized evaluation.

```
# Run explain() on explainer
explanation <- lime::explain(
  as.data.frame(test_h2o[1:10,-1]),
  explainer     = explainer,
  n_labels      = 1,
  n_features    = 4,
  kernel_width  = 0.5)
```

21.7 Feature Importance Visualization

The payoff for the work we put in using LIME is this feature importance plot. This allows us to visualize each of the ten cases (observations) from the test data. The top four features for each case are shown. Note that they are not the same for each case. The green bars mean that the feature supports the model conclusion, and the red bars contradict. We’ll focus in on Cases with `Label = Yes`, which are predicted to have attrition. We can see a common theme with Case 3 and Case 7: Training Time, Job Role, and Over Time are among the top factors influencing attrition. These are only two cases, but they can be used to potentially generalize to the larger population as we will see next.

```
plot_features(explanation) +
  labs(title = "HR Predictive Analytics: LIME Feature Importance Visualization",
       subtitle = "Hold Out (Test) Set, First 10 Cases Shown")
```



21.7.1 What features are linked to employee attrition

Now we turn to our three critical features from the LIME Feature Importance Plot:

- Training Time
- Job Role
- Over Time

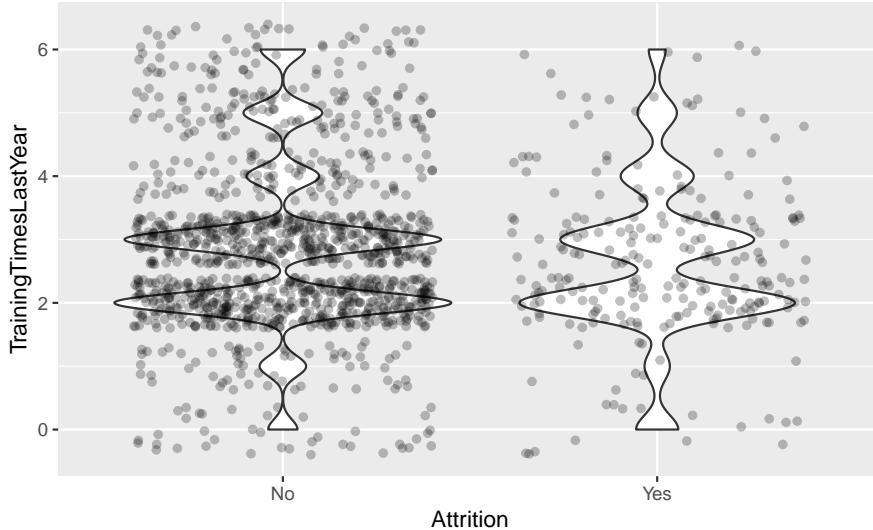
We'll subset this data and visualize to detect trends.

```
# Focus on critical features of attrition
attrition_critical_features <- hr_data %>%
  tibble::as_tibble() %>%
  select(Attrition, TrainingTimesLastYear, JobRole, OverTime) %>%
  rowid_to_column(var = "Case")
attrition_critical_features
#> # A tibble: 1,470 x 5
#>   Case Attrition TrainingTimesLastYear JobRole           OverTime
#>   <int> <fct>      <dbl> <fct>          <fct>
#> 1     1 Yes            0 Sales Executive Yes
#> 2     2 No             3 Research Scientist No
#> 3     3 Yes            3 Laboratory Technician Yes
#> 4     4 No             3 Research Scientist Yes
#> 5     5 No             3 Laboratory Technician No
#> 6     6 No             2 Laboratory Technician No
#> # ... with 1,464 more rows
```

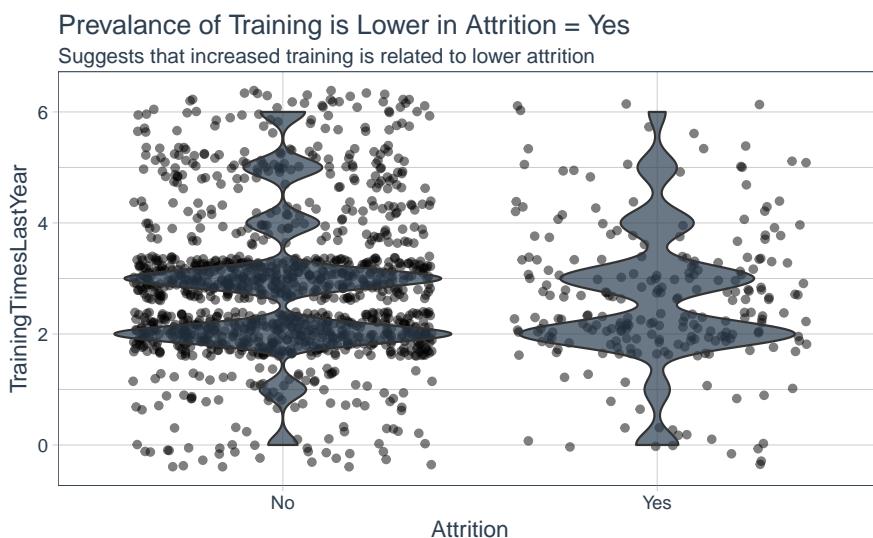
21.7.2 Training

From the violin plot, the employees that stay tend to have a large peaks at two and three trainings per year whereas the employees that leave tend to have a large peak at two trainings per year. This suggests that employees with more trainings may be less likely to leave.

```
ggplot(attrition_critical_features, aes(x = Attrition,
                                         y = TrainingTimesLastYear)) +
  geom_violin() +
  geom_jitter(alpha = 0.25)
```



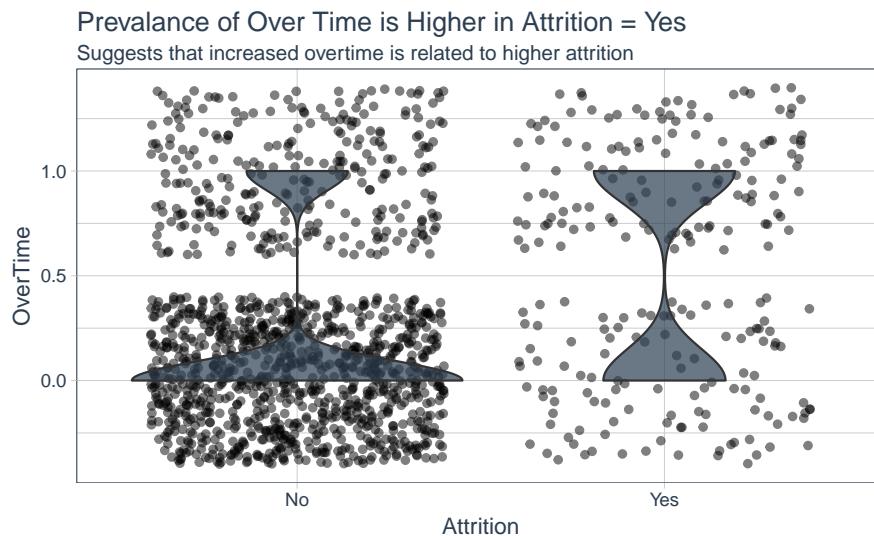
```
attrition_critical_features %>%
  ggplot(aes(Attrition, TrainingTimesLastYear)) +
  geom_jitter(alpha = 0.5, fill = palette_light()[[1]]) +
  geom_violin(alpha = 0.7, fill = palette_light()[[1]]) +
  theme_tq() +
  labs(
    title = "Prevalance of Training is Lower in Attrition = Yes",
    subtitle = "Suggests that increased training is related to lower attrition"
  )
```



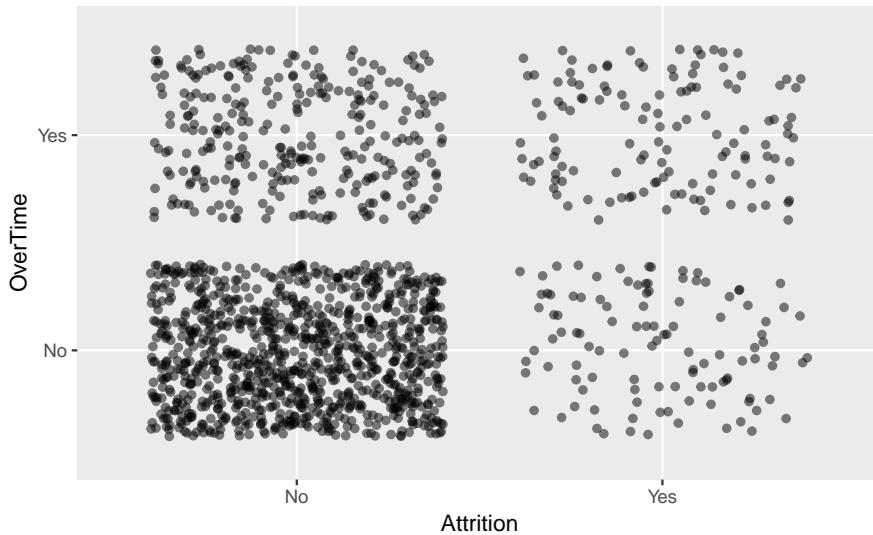
21.7.3 Overtime

The plot below shows a very interesting relationship: a very high proportion of employees that turnover are working over time. The opposite is true for employees that stay.

```
attrition_critical_features %>%
  mutate(OverTime = case_when(
    OverTime == "Yes" ~ 1,
    OverTime == "No" ~ 0 )) %>%
  ggplot(aes(Attrition, OverTime)) +
  geom_jitter(alpha = 0.5, fill = palette_light()[[1]]) +
  geom_violin(alpha = 0.7, fill = palette_light()[[1]]) +
  theme_tq() +
  labs(
    title = "Prevalance of Over Time is Higher in Attrition = Yes",
    subtitle = "Suggests that increased overtime is related to higher attrition")
```



```
ggplot(attrition_critical_features, aes(x = Attrition,
                                         y = OverTime,
                                         )) +
  # geom_violin(aes(y = ..prop.., group = 1)) +
  geom_jitter(alpha = 0.5)
```

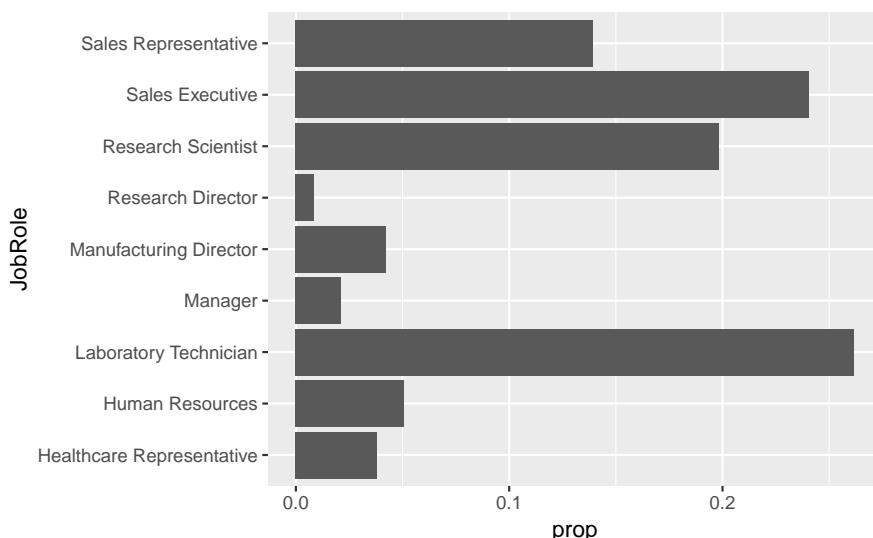


21.7.4 Job Role

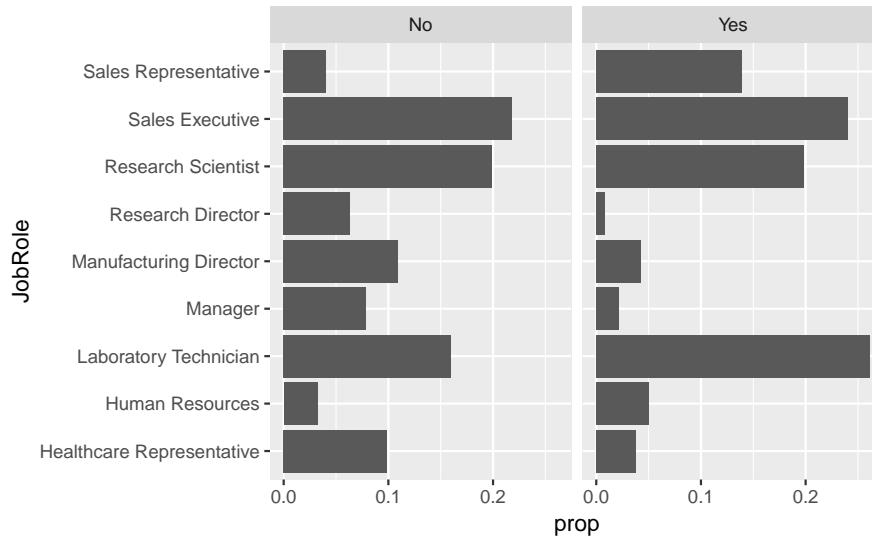
Several job roles are experiencing more turnover. Sales reps have the highest turnover at about 40% followed by Lab Technician, Human Resources, Sales Executive, and Research Scientist. It may be worthwhile to investigate what localized issues could be creating the high turnover among these groups within the organization.

```
p <- ggplot(data = subset(attrition_critical_features, Attrition == "Yes"),
             mapping = aes(x = JobRole))
p + geom_bar(mapping = aes(y = ..prop.., group = 1)) +
  coord_flip()

# geom_bar(mapping = aes(y = ..prop.., group = 1))
```



```
p <- ggplot(data = attrition_critical_features,
             mapping = aes(x = JobRole))
p + geom_bar(mapping = aes(y = ..prop.., group = 1)) +
  coord_flip() +
  facet_wrap(Attrition ~ .)
```



```
attrition_critical_features %>%
  group_by(JobRole, Attrition) %>%
  summarize(total = n())

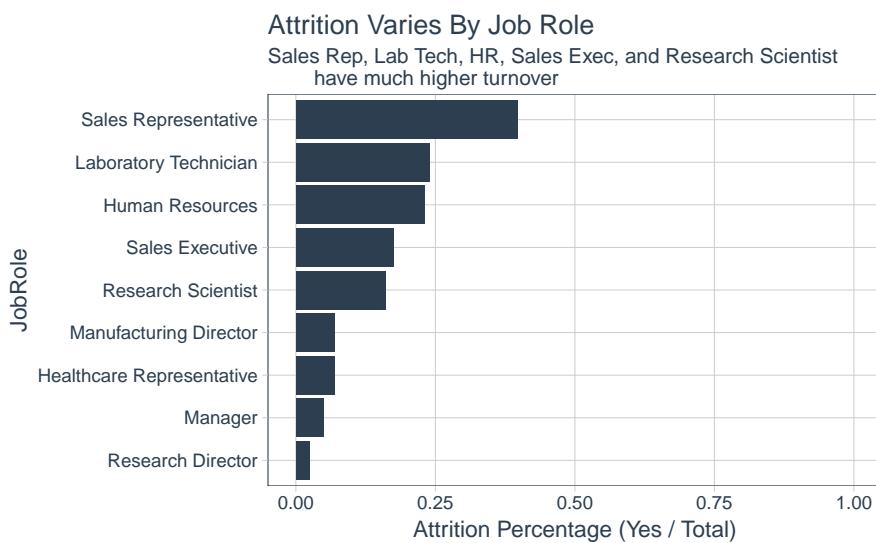
#> # A tibble: 18 x 3
#> # Groups:   JobRole [9]
#>   JobRole          Attrition total
#>   <fct>            <fct>     <int>
#> 1 Healthcare Representative No      122
#> 2 Healthcare Representative Yes     9
#> 3 Human Resources           No      40
#> 4 Human Resources           Yes     12
#> 5 Laboratory Technician    No      197
#> 6 Laboratory Technician    Yes     62
#> # ... with 12 more rows
```

```
attrition_critical_features %>%
  group_by(JobRole, Attrition) %>%
  summarize(total = n()) %>%
  spread(key = Attrition, value = total) %>%
  mutate(pct_attrition = Yes / (Yes + No))

#> # A tibble: 9 x 4
#> # Groups:   JobRole [9]
#>   JobRole          No     Yes pct_attrition
#>   <fct>        <int>  <int>      <dbl>
#> 1 Healthcare Representative 122      9      0.0687
#> 2 Human Resources          40      12      0.231
#> 3 Laboratory Technician   197      62      0.239
#> 4 Manager                  97      5      0.0490
#> 5 Manufacturing Director  135     10      0.0690
#> 6 Research Director        78      2      0.025
#> # ... with 3 more rows
```

```
attrition_critical_features %>%
  group_by(JobRole, Attrition) %>%
  summarize(total = n()) %>%
  spread(key = Attrition, value = total) %>%
  mutate(pct_attrition = Yes / (Yes + No)) %>%
```

```
ggplot(aes(x =forcats::fct_reorder(JobRole, pct_attrition), y = pct_attrition)) +
  geom_bar(stat = "identity", alpha = 1, fill = palette_light()[[1]]) +
  expand_limits(y = c(0, 1)) +
  coord_flip() +
  theme_tq() +
  labs(
    title = "Attrition Varies By Job Role",
    subtitle = "Sales Rep, Lab Tech, HR, Sales Exec, and Research Scientist have much higher turnover",
    y = "Attrition Percentage (Yes / Total)",
    x = "JobRole"
)
```



21.8 Conclusions

There's a lot to take away from this article. We showed how you can use predictive analytics to develop sophisticated models that very accurately detect employees that are at risk of turnover. The autoML algorithm from H2O.ai worked well for classifying attrition with an accuracy around 87% on unseen / unmodeled data. We then used LIME to breakdown the complex ensemble model returned from H2O into critical features that are related to attrition. Overall, this is a really useful example where we can see how machine learning and data science can be used in business applications.

Chapter 22

Dealing with unbalanced data

22.1 Breast cancer dataset

22.2 Introduction

Source: https://shiring.github.io/machine_learning/2017/04/02/unbalanced

```
library(caret)
#> Loading required package: lattice
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures    rlang
#>   c.quosures    rlang
#>   print.quosures rlang
library(mice)
#>
#> Attaching package: 'mice'
#> The following objects are masked from 'package:base':
#>
#>   cbind, rbind
library(ggplot2)
```

In my last post, where I shared the code that I used to produce an example analysis to go along with my webinar on building meaningful models for disease prediction, I mentioned that it is advised to consider over- or under-sampling when you have unbalanced data sets. Because my focus in this webinar was on evaluating model performance, I did not want to add an additional layer of complexity and therefore did not further discuss how to specifically deal with unbalanced data.

But because I had gotten a few questions regarding this, I thought it would be worthwhile to explain over- and under-sampling techniques in more detail and show how you can very easily implement them with `caret`.

22.3 Read and process the data

```
bc_data <- read.table(file.path(data_raw_dir, "breast-cancer-wisconsin.data"),
                      header = FALSE, sep = ",")
```

```

colnames(bc_data) <- c("sample_code_number", "clump_thickness",
                      "uniformity_of_cell_size", "uniformity_of_cell_shape",
                      "marginal_adhesion", "single_epithelial_cell_size",
                      "bare_nuclei", "bland_chromatin", "normal_nucleoli",
                      "mitosis", "classes")

bc_data$classes <- ifelse(bc_data$classes == "2", "benign",
                           ifelse(bc_data$classes == "4", "malignant", NA))

bc_data[bc_data == "?"] <- NA

# how many NAs are in the data
length(which(is.na(bc_data)))
#> [1] 16

# impute missing data

# skip columns: sample_code_number and classes
bc_data[, 2:10] <- apply(bc_data[, 2:10], 2, function(x) as.numeric(as.character(x)))

# impute but stay mute
dataset_impute <- mice(bc_data[, 2:10], print = FALSE)

# bind "classes" with the rest. skip "sample_code_number"
bc_data <- cbind(bc_data[, 11, drop = FALSE],
                  mice::complete(dataset_impute, action = 1))

bc_data$classes <- as.factor(bc_data$classes)

```

22.3.1 Unbalanced data

In this context, unbalanced data refers to classification problems where we have unequal instances for different classes. Having unbalanced data is actually very common in general, but it is especially prevalent when working with disease data where we usually have more healthy control samples than disease cases. Even more extreme unbalance is seen with fraud detection, where e.g. most credit card uses are okay and only very few will be fraudulent. In the example I used for my webinar, a *breast cancer* dataset, we had about twice as many benign than malignant samples.

```

# how many benign and malignant cases are there?
summary(bc_data$classes)
#>   benign malignant
#>     458       241

```

22.3.1.1 Why is unbalanced data a problem in machine learning?

Most machine learning classification algorithms are sensitive to unbalance in the predictor classes. Let's consider an even more extreme example than our breast cancer dataset: assume we had 10 malignant vs 90 benign samples. A machine learning model that has been trained and tested on such a dataset could now predict "benign" for all samples and still gain a very high accuracy. An unbalanced dataset will bias the prediction model towards the more common class!

22.3.1.2 How to balance data for modeling

The basic theoretical concepts behind over- and under-sampling are very simple:

With under-sampling, we randomly select a subset of samples from the class with more instances to match the number of samples coming from each class. In our example, we would randomly pick 241 out of the 458 benign cases. The main disadvantage of under-sampling is that we lose potentially relevant information from the left-out samples.

With oversampling, we randomly duplicate samples from the class with fewer instances or we generate additional instances based on the data that we have, so as to match the number of samples in each class. While we avoid losing information with this approach, we also run the risk of overfitting our model as we are more likely to get the same samples in the training and in the test data, i.e. the test data is no longer independent from training data. This would lead to an overestimation of our model's performance and generalizability.

In reality though, we should not simply perform over- or under-sampling on our training data and then run the model. We need to account for cross-validation and perform over- or under-sampling on each fold independently to get an honest estimate of model performance!

22.3.1.3 Modeling the original unbalanced data

Here is the same model I used in my webinar example: I randomly divide the data into training and test sets (stratified by class) and perform Random Forest modeling with 10 x 10 repeated cross-validation. Final model performance is then measured on the test set.

```
set.seed(42)
index <- createDataPartition(bc_data$classes, p = 0.7, list = FALSE)
train_data <- bc_data[index, ]
test_data <- bc_data[-index, ]

set.seed(42)
model_rf <- caret::train(classes ~ .,
                           data = train_data,
                           method = "rf",
                           preProcess = c("scale", "center"),
                           trControl = trainControl(method = "repeatedcv",
                                                     number = 10,
                                                     repeats = 10,
                                                     verboseIter = FALSE))

final <- data.frame(actual = test_data$classes,
                     predict(model_rf,
                             newdata = test_data,
                             type = "prob"))

final$predict <- ifelse(final$benign > 0.5, "benign", "malignant")

final_predict <- as.factor(final$predict)
test_data_classes <- as.factor(test_data$classes)

cm_original <- confusionMatrix(final_predict, test_data_classes)
cm_original$byClass['Sensitivity']
#> Sensitivity
#>      0.978
```

22.4 Under-sampling

Luckily, `caret` makes it very easy to incorporate over- and under-sampling techniques with cross-validation resampling. We can simply add the sampling option to our `trainControl` and choose down for under- (also called down-) sampling. The rest stays the same as with our original model.

```
set.seed(42)
ctrl <- trainControl(method = "repeatedcv",
                      number = 10,
                      repeats = 10,
                      verboseIter = FALSE,
                      sampling = "down")

model_rf_under <- caret::train(classes ~.,
                                 data = train_data,
                                 method = "rf",
                                 preProcess = c("scale", "center"),
                                 trControl = ctrl)

final_under <- data.frame(actual = test_data$classes,
                           predict(model_rf_under,
                                   newdata = test_data,
                                   type = "prob"))

final_under$predict <- ifelse(final_under$benign > 0.5, "benign", "malignant")

final_under_predict <- as.factor(final_under$predict)
test_data_classes <- test_data$classes

cm_under <- confusionMatrix(final_under_predict, test_data_classes)
cm_under$byClass['Sensitivity']
#> Sensitivity
#>      0.978
```

22.5 Oversampling

For over- (also called up-) sampling we simply specify `sampling = "up"`.

```
set.seed(42)
ctrl <- trainControl(method = "repeatedcv",
                      number = 10,
                      repeats = 10,
                      verboseIter = FALSE,
                      sampling = "up")

model_rf_over <- caret::train(classes ~.,
                               data = train_data,
                               method = "rf",
                               preProcess = c("scale", "center"),
                               trControl = ctrl)
```

```

final_over <- data.frame(actual = test_data$classes,
                         predict(model_rf_over,
                                 newdata = test_data,
                                 type = "prob"))

final_over$predict <- ifelse(final_over$benign > 0.5, "benign", "malignant")

final_over_predict <- as.factor(final_over$predict)
test_data_classes <- test_data$classes

cm_over <- confusionMatrix(final_over_predict, test_data_classes)
cm_over$byClass['Sensitivity']
#> Sensitivity
#>      0.978

```

22.5.1 ROSE

Besides over- and under-sampling, there are hybrid methods that combine under-sampling with the generation of additional data. Two of the most popular are ROSE and SMOTE.

From Nicola Lunardon, Giovanna Menardi and Nicola Torelli's "ROSE: A Package for Binary Imbalanced Learning" (R Journal, 2014, Vol. 6 Issue 1, p. 79): "The ROSE package provides functions to deal with binary classification problems in the presence of imbalanced classes. Artificial balanced samples are generated according to a smoothed bootstrap approach and allow for aiding both the phases of estimation and accuracy evaluation of a binary classifier in the presence of a rare class. Functions that implement more traditional remedies for the class imbalance and different metrics to evaluate accuracy are also provided. These are estimated by holdout, bootstrap, or cross-validation methods."

You implement them the same way as before, this time choosing sampling = "rose"...

```

set.seed(42)
ctrl <- trainControl(method = "repeatedcv",
                      number = 10,
                      repeats = 10,
                      verboseIter = FALSE,
                      sampling = "rose")

model_rf_rose <- caret::train(classes ~ .,
                                data = train_data,
                                method = "rf",
                                preProcess = c("scale", "center"),
                                trControl = ctrl)

#> Loaded ROSE 0.0-3

final_rose <- data.frame(actual = test_data$classes,
                           predict(model_rf_rose,
                                   newdata = test_data,
                                   type = "prob"))

final_rose$predict <- ifelse(final_rose$benign > 0.5, "benign", "malignant")

cm_rose <- confusionMatrix(as.factor(final_rose$predict),
                           as.factor(test_data$classes))
cm_rose$byClass['Sensitivity']

```

```
#> Sensitivity
#>      0.985
```

22.5.2 SMOTE

... or by choosing sampling = “smote” in the `trainControl` settings.

From Nitesh V. Chawla, Kevin W. Bowyer, Lawrence O. Hall and W. Philip Kegelmeyer’s “SMOTE: Synthetic Minority Over-sampling Technique” (Journal of Artificial Intelligence Research, 2002, Vol. 16, pp. 321–357): “This paper shows that a combination of our method of over-sampling the minority (abnormal) class and under-sampling the majority (normal) class can achieve better classifier performance (in ROC space) than only under-sampling the majority class. This paper also shows that a combination of our method of over-sampling the minority class and under-sampling the majority class can achieve better classifier performance (in ROC space) than varying the loss ratios in Ripper or class priors in Naive Bayes. Our method of over-sampling the minority class involves creating synthetic minority class examples.”

```
set.seed(42)
ctrl <- trainControl(method = "repeatedcv",
                      number = 10,
                      repeats = 10,
                      verboseIter = FALSE,
                      sampling = "smote")

model_rf_smote <- caret::train(classes ~ .,
                                 data = train_data,
                                 method = "rf",
                                 preProcess = c("scale", "center"),
                                 trControl = ctrl)

#> Loading required package: grid
#> Registered S3 method overwritten by 'xts':
#>   method      from
#>   as.zoo.xts zoo
#> Registered S3 method overwritten by 'quantmod':
#>   method      from
#>   as.zoo.data.frame zoo

final_smote <- data.frame(actual = test_data$classes,
                           predict(model_rf_smote,
                                   newdata = test_data,
                                   type = "prob"))

final_smote$predict <- ifelse(final_smote$benign > 0.5, "benign", "malignant")

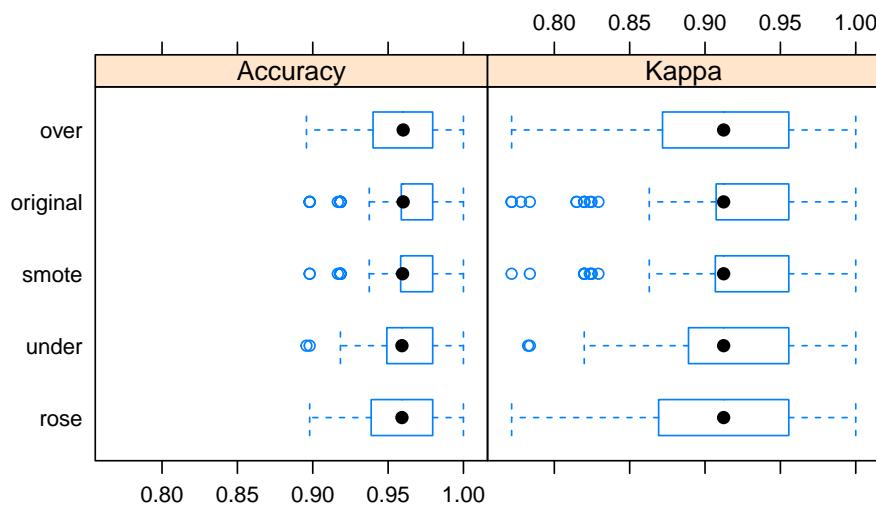
cm_smote <- confusionMatrix(as.factor(final_smote$predict),
                            as.factor(test_data$classes))
cm_smote$byClass['Sensitivity']
#> Sensitivity
#>      0.978
```

22.6 Predictions

Now let's compare the predictions of all these models:

```
models <- list(
  original = model_rf,
  under = model_rf_under,
  over = model_rf_over,
  smote = model_rf_smote,
  rose = model_rf_rose)

resampling <- resamples(models)
bwplot(resampling)
```



```
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
comparison <- data.frame(model = names(models),
  Sensitivity = rep(NA, length(models)),
  Specificity = rep(NA, length(models)),
  Precision = rep(NA, length(models)),
  Recall = rep(NA, length(models)),
  F1 = rep(NA, length(models)))

for (name in names(models)) {
  cm_model <- get(paste0("cm_", name))
  comparison[comparison$model==name, ] <- filter(comparison, model==name) %>%
    mutate(Sensitivity = cm_model$byClass["Sensitivity"],
           Specificity = cm_model$byClass["Specificity"],
           Precision = cm_model$byClass["Precision"],
           Recall = cm_model$byClass["Recall"],
           F1 = cm_model$byClass["F1"])
```

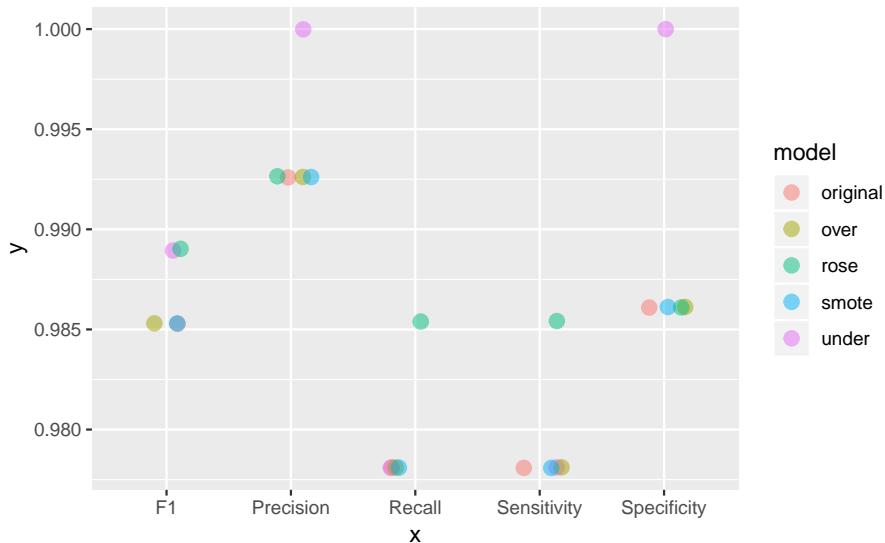
```

    )
}

print(comparison)
#>      model Sensitivity Specificity Precision Recall     F1
#> 1 original      0.978      0.986      0.993  0.978 0.985
#> 2 under         0.978      1.000      1.000  0.978 0.989
#> 3 over          0.978      0.986      0.993  0.978 0.985
#> 4 smote         0.978      0.986      0.993  0.978 0.985
#> 5 rose          0.985      0.986      0.993  0.985 0.989

library(tidyr)
#>
#> Attaching package: 'tidyr'
#> The following object is masked from 'package:mice':
#>
#>     complete
comparison %>%
  gather(x, y, Sensitivity:F1) %>%
  ggplot(aes(x = x, y = y, color = model)) +
  geom_jitter(width = 0.2, alpha = 0.5, size = 3)

```



With this small dataset, we can already see how the different techniques can influence model performance. **Sensitivity** (or **recall**) describes the proportion of benign cases that have been predicted correctly, while **specificity** describes the proportion of malignant cases that have been predicted correctly. **Precision** describes the true positives, i.e. the proportion of benign predictions that were actual from benign samples. **F1** is the weighted average of precision and sensitivity/ recall.

22.7 Final notes

Here, all four methods improved specificity and precision compared to the original model. Under-sampling, over-sampling and ROSE additionally improved precision and the F1 score.

This post shows a simple example of how to correct for unbalance in datasets for machine learning. For more advanced instructions and potential caveats with these techniques, check out the excellent **caret** documentation.

If you are interested in more machine learning posts, check out the category listing for machine_learning on my blog.

Chapter 23

Ten different methods to assess Variable Importance

23.1 Glaucoma dataset

Source: <https://www.machinelearningplus.com/machine-learning/feature-selection/>

23.2 Introduction

In real-world datasets, it is fairly common to have columns that are nothing but noise.

You are better off getting rid of such variables because of the memory space they occupy, the time and the computational resources it is going to cost, especially in large datasets.

Sometimes, you have a variable that makes business sense, but you are not sure if it actually helps in predicting the Y. You also need to consider the fact that, a feature that could be useful in one ML algorithm (say a decision tree) may go underrepresented or unused by another (like a regression model).

Having said that, it is still possible that a variable that shows poor signs of helping to explain the response variable (Y), can turn out to be significantly useful in the presence of (or combination with) other predictors. What I mean by that is, a variable might have a low correlation value of (~0.2) with Y. But in the presence of other variables, it can help to explain certain patterns/phenomenon that other variables can't explain.

In such cases, it can be hard to make a call whether to include or exclude such variables.

The strategies we are about to discuss can help fix such problems. Not only that, it will also help understand if a particular variable is important or not and how much it is contributing to the model

An important caveat. It is always best to have variables that have sound business logic backing the inclusion of a variable and rely solely on variable importance metrics.

Alright. Let's load up the 'Glaucoma' dataset where the goal is to predict if a patient has Glaucoma or not based on 63 different physiological measurements. You can directly run the codes or download the dataset here.

A lot of interesting examples ahead. Let's get started.

```
# Load Packages and prepare dataset
library(TH.data)
#> Loading required package: survival
```

```

#> Loading required package: MASS
#>
#> Attaching package: 'TH.data'
#> The following object is masked from 'package:MASS':
#>
#>     geyser
library(caret)
#> Loading required package: lattice
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures    rlang
#>   c.quosures    rlang
#>   print.quosures rlang
#>
#> Attaching package: 'caret'
#> The following object is masked from 'package:survival':
#>
#>     cluster
library(tictoc)

data("GlaucomaM", package = "TH.data")
trainData <- GlaucomaM
head(trainData)
#>    ag   at   as   an   ai   eag   eat   eas   ean   eai   abrg   abrt
#> 2  2.22 0.354 0.580 0.686 0.601 1.267 0.336 0.346 0.255 0.331 0.479 0.260
#> 43 2.68 0.475 0.672 0.868 0.667 2.053 0.440 0.520 0.639 0.454 1.090 0.377
#> 25 1.98 0.343 0.508 0.624 0.504 1.200 0.299 0.396 0.259 0.246 0.465 0.209
#> 65 1.75 0.269 0.476 0.525 0.476 0.612 0.147 0.017 0.044 0.405 0.170 0.062
#> 70 2.99 0.599 0.686 1.039 0.667 2.513 0.543 0.607 0.871 0.492 1.800 0.431
#> 16 2.92 0.483 0.763 0.901 0.770 2.200 0.462 0.637 0.504 0.597 1.311 0.394
#>    abrs   abrn   abri   hic   mhcg   mhct   mhcs   mhcn   mhci   phcg
#> 2  0.107 0.014 0.098 0.214 0.111 0.412 0.036 0.105 -0.022 -0.139
#> 43 0.257 0.212 0.245 0.382 0.140 0.338 0.104 0.080 0.109 -0.015
#> 25 0.112 0.041 0.103 0.195 0.062 0.356 0.045 -0.009 -0.048 -0.149
#> 65 0.000 0.000 0.108 -0.030 -0.015 0.074 -0.084 -0.050 0.035 -0.182
#> 70 0.494 0.601 0.274 0.383 0.089 0.233 0.145 0.023 0.007 -0.131
#> 16 0.365 0.251 0.301 0.442 0.128 0.375 0.049 0.111 0.052 -0.088
#>    phct   phcs   phcn   phci   hvc   vbsg   vbst   vbss   vbsn   vbsi   vasg
#> 2  0.242 -0.053 0.010 -0.139 0.613 0.303 0.103 0.088 0.022 0.090 0.062
#> 43 0.296 -0.015 -0.015 0.036 0.382 0.676 0.181 0.186 0.141 0.169 0.029
#> 25 0.206 -0.092 -0.081 -0.149 0.557 0.300 0.084 0.088 0.046 0.082 0.036
#> 65 -0.097 -0.125 -0.138 -0.182 0.373 0.048 0.011 0.000 0.000 0.036 0.070
#> 70 0.163 0.055 -0.131 -0.115 0.405 0.889 0.151 0.253 0.330 0.155 0.020
#> 16 0.281 -0.067 -0.062 -0.088 0.507 0.972 0.213 0.316 0.197 0.246 0.043
#>    vast   vass   vasn   vasi   vbrg   vbrt   vbrs   vbrn   vbri   varg   vart   vars
#> 2  0.000 0.011 0.032 0.018 0.075 0.039 0.021 0.002 0.014 0.756 0.009 0.209
#> 43 0.001 0.007 0.011 0.010 0.370 0.127 0.099 0.050 0.093 0.410 0.006 0.105
#> 25 0.002 0.004 0.016 0.013 0.081 0.034 0.019 0.007 0.021 0.565 0.014 0.132
#> 65 0.005 0.030 0.033 0.002 0.005 0.001 0.000 0.000 0.004 0.380 0.032 0.147
#> 70 0.001 0.004 0.008 0.007 0.532 0.103 0.173 0.181 0.075 0.228 0.011 0.026
#> 16 0.001 0.005 0.028 0.009 0.467 0.136 0.148 0.078 0.104 0.540 0.008 0.133
#>    varn   vari   mdg   mdt   mds   mdn   mdi   tmg   tmt   tms   tmn

```

```
#> 2 0.298 0.240 0.705 0.637 0.738 0.596 0.691 -0.236 -0.018 -0.230 -0.510
#> 43 0.181 0.117 0.898 0.850 0.907 0.771 0.940 -0.211 -0.014 -0.165 -0.317
#> 25 0.243 0.177 0.687 0.643 0.689 0.684 0.700 -0.185 -0.097 -0.235 -0.337
#> 65 0.151 0.050 0.207 0.171 0.022 0.046 0.221 -0.148 -0.035 -0.449 -0.217
#> 70 0.105 0.087 0.721 0.638 0.730 0.730 0.640 -0.052 -0.105 0.084 -0.012
#> 16 0.232 0.167 0.927 0.842 0.953 0.906 0.898 -0.040 0.087 0.018 -0.094
#> tmi mr rnf mdic emd mu Class
#> 2 -0.158 0.841 0.410 0.137 0.239 0.035 normal
#> 43 -0.192 0.924 0.256 0.252 0.329 0.022 normal
#> 25 -0.020 0.795 0.378 0.152 0.250 0.029 normal
#> 65 -0.091 0.746 0.200 0.027 0.078 0.023 normal
#> 70 -0.054 0.977 0.193 0.297 0.354 0.034 normal
#> 16 -0.051 0.965 0.339 0.333 0.442 0.028 normal
```

23.3 1. Boruta

Boruta is a feature ranking and selection algorithm based on random forests algorithm.

The advantage with Boruta is that it clearly decides if a variable is important or not and helps to select variables that are statistically significant. Besides, you can adjust the strictness of the algorithm by adjusting the *p* values that defaults to 0.01 and the `maxRuns`.

`maxRuns` is the number of times the algorithm is run. The higher the `maxRuns` the more selective you get in picking the variables. The default value is 100.

In the process of deciding if a feature is important or not, some features may be marked by Boruta as ‘Tentative’. Sometimes increasing the `maxRuns` can help resolve the ‘Tentativeness’ of the feature.

Lets see an example based on the Glaucoma dataset from `TH.data` package that I created earlier.

```
# install.packages('Boruta')
library(Boruta)
#> Loading required package: ranger
```

The `boruta` function uses a formula interface just like most predictive modeling functions. So the first argument to `boruta()` is the formula with the response variable on the left and all the predictors on the right.

By placing a dot, all the variables in `trainData` other than `Class` will be included in the model.

The `doTrace` argument controls the amount of output printed to the console. Higher the value, more the log details you get. So save space I have set it to 0, but try setting it to 1 and 2 if you are running the code.

Finally the output is stored in `boruta_output`.

```
# Perform Boruta search
boruta_output <- Boruta(Class ~ ., data=na.omit(trainData), doTrace=0)
```

Let's see what the `boruta_output` contains.

```
names(boruta_output)
#> [1] "finalDecision"   "ImpHistory"      "pValue"          "maxRuns"
#> [5] "light"           "mcAdj"          "timeTaken"       "roughfixed"
#> [9] "call"            "impSource"

# Get significant variables including tentatives
boruta_signif <- getSelectedAttributes(boruta_output, withTentative = TRUE)
```

```
print(boruta_signif)
#> [1] "as"    "ai"    "eas"   "ean"   "abrg"  "abrs"  "abrн"  "abri"  "hic"   "mhcg"
#> [11] "mhcs" "mhcn" "mhci"  "phcg"  "phcn"  "phci"  "hvc"   "vbsg"  "vbss"  "vbsn"
#> [21] "vbsi" "vasg"  "vass"  "vasi"  "vbrg"  "vbrs"  "vbrн"  "vbri"  "varg"  "vart"
#> [31] "vars"  "varn"  "vari"  "mdn"   "tmг"   "tmt"   "tms"   "tmi"   "mr"   "rnf"
#> [41] "mdic"  "emd"
```

If you are not sure about the tentative variables being selected for granted, you can choose a TentativeRoughFix on boruta_output.

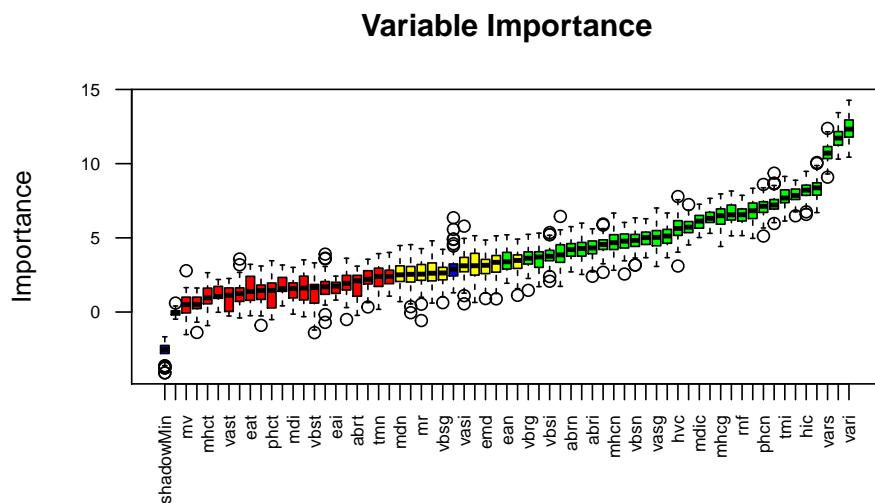
```
# Do a tentative rough fix
roughFixMod <- TentativeRoughFix(boruta_output)
boruta_signif <- getSelectedAttributes(roughFixMod)
print(boruta_signif)
#> [1] "as"    "ai"    "ean"   "abrg"  "abrs"  "abrн"  "abri"  "hic"   "mhcg"  "mhcn"
#> [11] "mhci" "phcg"  "phcn"  "phci"  "hvc"   "vbsn"  "vbsi"  "vasg"  "vass"  "vasi"
#> [21] "vbrg"  "vbrs"  "vbrн"  "vbri"  "varg"  "vart"  "vars"  "varn"  "vari"  "mdn"
#> [31] "tmг"   "tms"   "tmi"   "mr"   "rnf"   "mdic"
```

There you go. Boruta has decided on the ‘Tentative’ variables on our behalf. Let’s find out the importance scores of these variables.

```
# Variable Importance Scores
imps <- attStats(roughFixMod)
imps2 = imps[imps$decision != 'Rejected', c('meanImp', 'decision')]
head(imps2[order(-imps2$meanImp), ]) # descending sort
#>      meanImp decision
#> vari    12.37 Confirmed
#> varg    11.74 Confirmed
#> vars    10.74 Confirmed
#> phci     8.34 Confirmed
#> hic      8.21 Confirmed
#> varn     7.88 Confirmed
```

Let’s plot it to see the importances of these variables.

```
# Plot variable importance
plot(boruta_output, cex.axis=.7, las=2, xlab="", main="Variable Importance")
```



This plot reveals the importance of each of the features.

The columns in green are ‘confirmed’ and the ones in red are not. There are couple of blue bars representing `ShadowMax` and `ShadowMin`. They are not actual features, but are used by the boruta algorithm to decide if a variable is important or not.

23.4 Variable Importance from Machine Learning Algorithms

Another way to look at feature selection is to consider variables most used by various ML algorithms the most to be important.

Depending on how the machine learning algorithm learns the relationship between X’s and Y, different machine learning algorithms may possibly end up using different variables (but mostly common vars) to various degrees.

What I mean by that is, the variables that proved useful in a tree-based algorithm like `rpart`, can turn out to be less useful in a regression-based model. So all variables need not be equally useful to all algorithms.

So how do we find the variable importance for a given ML algo?

`train()` the desired model using the caret package. Then, use `varImp()` to determine the feature importances.

You may want to try out multiple algorithms, to get a feel of the usefulness of the features across algos.

23.4.1 rpart

```
# Train an rpart model and compute variable importance.
library(caret)
set.seed(100)
rPartMod <- train(Class ~ .,
                   data=trainData,
                   method="rpart")

rpartImp <- varImp(rPartMod)
print(rpartImp)
#> rpart variable importance
#>
#>    only 20 most important variables shown (out of 62)
#>
#>      Overall
#> varg     100.0
#> vari     93.2
#> vars     85.2
#> varn     76.9
#> tmi      72.3
#> mhcn     0.0
#> as       0.0
#> phcs     0.0
#> vbst     0.0
#> abrt     0.0
#> vbsg     0.0
#> eai      0.0
#> ubrs     0.0
#> vbsi     0.0
```

```
#> eag      0.0
#> tmt      0.0
#> phcn     0.0
#> vart     0.0
#> mds      0.0
#> an       0.0
```

Only 5 of the 63 features was used by rpart and if you look closely, the 5 variables used here are in the top 6 that boruta selected.

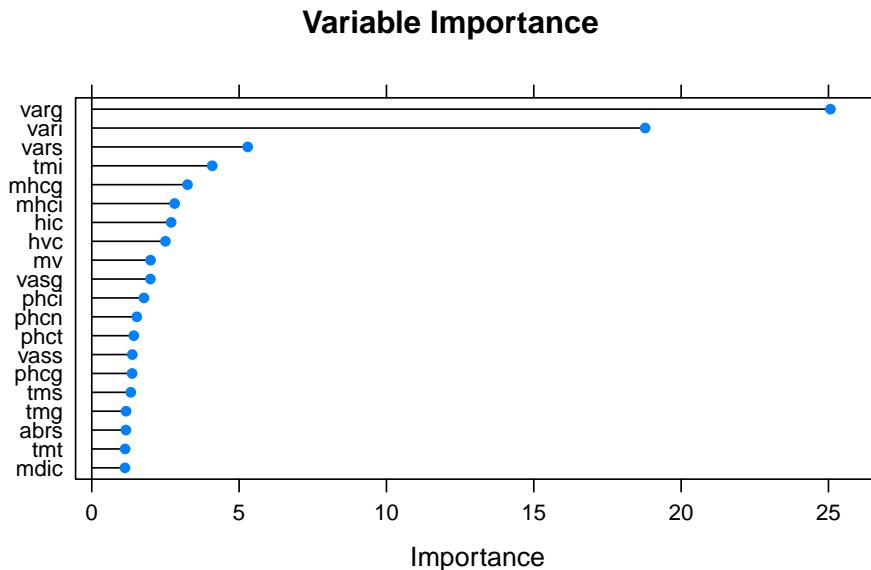
Let's do one more: the variable importances from Regularized Random Forest (RRF) algorithm.

23.4.2 Regularized Random Forest (RRF)

```
tic()
# Train an RRF model and compute variable importance.
set.seed(100)
rrfMod <- train(Class ~ .,
                  data = trainData,
                  method = "RRF")
#> Registered S3 method overwritten by 'RRF':
#>   method      from
#>   plot.margin randomForest

rrfImp <- varImp(rrfMod, scale=F)
toc()
#> 341.618 sec elapsed
rrfImp
#> RRF variable importance
#>
#> only 20 most important variables shown (out of 62)
#>
#>      Overall
#> varg    25.07
#> vari    18.78
#> vars    5.29
#> tmi     4.09
#> mhcg    3.25
#> mhci    2.81
#> hic     2.69
#> hvc     2.50
#> mv      2.00
#> vasg    1.99
#> phci    1.77
#> phcn    1.53
#> phct    1.43
#> vass    1.37
#> phcg    1.37
#> tms     1.32
#> tmg     1.16
#> abrs    1.16
#> tmt     1.13
#> mdic    1.13
```

```
plot(rrfImp, top = 20, main='Variable Importance')
```



The topmost important variables are pretty much from the top tier of Boruta's selections.

Some of the other algorithms available in `train()` that you can use to compute `varImp` are the following:

```
ada, AdaBag, AdaBoost.M1, adaboost, bagEarth, bagEarthGCV, bagFDA, bagFDAGCV, bartMachine, blasso, BstL...
```

23.5 Lasso Regression

Least Absolute Shrinkage and Selection Operator (LASSO) regression is a type of regularization method that penalizes with L1-norm.

It basically imposes a cost to having large weights (value of coefficients). And its called L1 regularization, because the cost added, is proportional to the absolute value of weight coefficients.

As a result, in the process of shrinking the coefficients, it eventually reduces the coefficients of certain unwanted features all the to zero. That is, it removes the unneeded variables altogether.

So effectively, LASSO regression can be considered as a variable selection technique as well.

```
library(glmnet)
#> Loading required package: Matrix
#> Loading required package: foreach
#> Loaded glmnet 2.0-16

# online data
# trainData <- read.csv('https://raw.githubusercontent.com/selva86/datasets/master/GlaucomaM.csv')

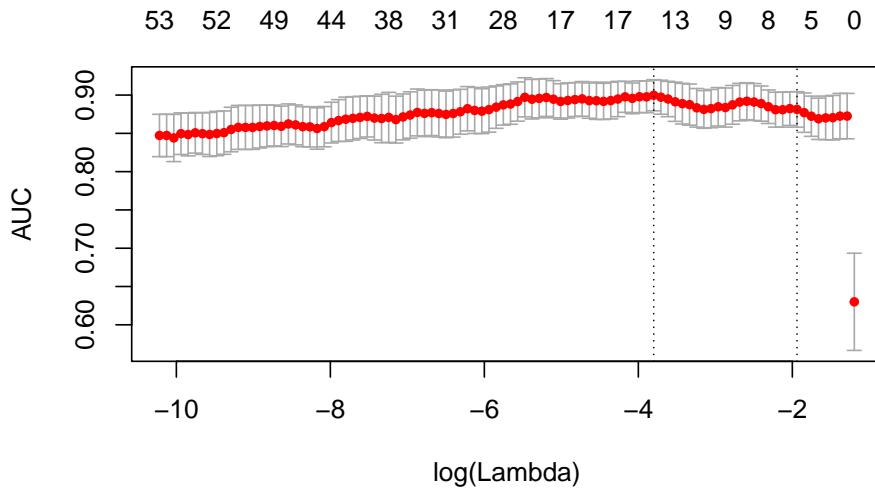
trainData <- read.csv(file.path(data_raw_dir, "glaucoma.csv"))

x <- as.matrix(trainData[,-63]) # all X vars
y <- as.double(as.matrix(ifelse(trainData[, 63]=='normal', 0, 1))) # Only Class

# Fit the LASSO model (Lasso: Alpha = 1)
set.seed(100)
cv.lasso <- cv.glmnet(x, y, family='binomial', alpha=1, parallel=TRUE, standardize=TRUE, type.measure='...
```

```
#> Warning: executing %dopar% sequentially: no parallel backend registered
```

```
# Results
plot(cv.lasso)
```



Let's see how to interpret this plot.

The X axis of the plot is the log of `lambda`. That means when it is 2 here, the lambda value is actually 100.

The numbers at the top of the plot show how many predictors were included in the model. The position of red dots along the Y-axis tells what AUC we got when you include as many variables shown on the top x-axis.

You can also see two dashed vertical lines.

The first one on the left points to the lambda with the lowest mean squared error. The one on the right point to the number of variables with the highest deviance within 1 standard deviation.

The best lambda value is stored inside 'cv.lasso\$lambda.min'.

```
# plot(cv.lasso$glmnet.fit, xvar="lambda", label=TRUE)
cat('Min Lambda: ', cv.lasso$lambda.min, '\n 1Sd Lambda: ', cv.lasso$lambda.1se)
#> Min Lambda:  0.0224
#>  1Sd Lambda:  0.144
df_coef <- round(as.matrix(coef(cv.lasso, s=cv.lasso$lambda.min)), 2)

# See all contributing variables
df_coef[df_coef[, 1] != 0, ]
#> (Intercept)      as      mhc1      phci      hvc      vast
#>     2.68     -1.59     3.85     5.60    -2.41   -13.90
#> vars      vari      mdn      mdi      tmg      tms
#>     -20.18    -1.58     0.50     0.99     0.06     2.56
#> tmi
#>     2.23
```

The above output shows what variables LASSO considered important. A high positive or low negative implies more important is that variable.

23.6 Step wise Forward and Backward Selection

Stepwise regression can be used to select features if the Y variable is a numeric variable. It is particularly used in selecting best linear regression models.

It searches for the best possible regression model by iteratively selecting and dropping variables to arrive at a model with the lowest possible AIC.

It can be implemented using the `step()` function and you need to provide it with a lower model, which is the base model from which it won't remove any features and an upper model, which is a full model that has all possible features you want to have.

Our case is not so complicated (< 20 vars), so lets just do a simple stepwise in 'both' directions.

I will use the `ozone` dataset for this where the objective is to predict the `ozone_reading` based on other weather related observations.

```
# Load data
# online
# trainData <- read.csv("http://rstatistics.net/wp-content/uploads/2015/09/ozone1.csv",
#                         stringsAsFactors=F)
trainData <- read.csv(file.path(data_raw_dir, "ozone1.csv"))
print(head(trainData))
#>   Month Day_of_month Day_of_week ozone_reading pressure_height Wind_speed
#> 1     1            1          4            3        5480             8
#> 2     1            2          5            3        5660             6
#> 3     1            3          6            3        5710             4
#> 4     1            4          7            5        5700             3
#> 5     1            5          1            5        5760             3
#> 6     1            6          2            6        5720             4
#>   Humidity Temperature_Sandburg Temperature_ElMonte Inversion_base_height
#> 1      20           40.5          39.8           5000
#> 2      41           38.0          46.7           4109
#> 3      28           40.0          49.5           2693
#> 4      37           45.0          52.3           590
#> 5      51           54.0          45.3           1450
#> 6      69           35.0          49.6           1568
#>   Pressure_gradient Inversion_temperature Visibility
#> 1          -15            30.6           200
#> 2          -14            48.0           300
#> 3          -25            47.7           250
#> 4          -24            55.0           100
#> 5          25            57.0            60
#> 6          15            53.8            60
```

The data is ready. Let's perform the stepwise.

```
# Step 1: Define base intercept only model
base.mod <- lm(ozone_reading ~ 1 , data=trainData)

# Step 2: Full model with all predictors
all.mod <- lm(ozone_reading ~ . , data= trainData)

# Step 3: Perform step-wise algorithm. direction='both' implies both forward and backward stepwise
stepMod <- step(base.mod, scope = list(lower = base.mod, upper = all.mod), direction = "both", trace = 0)

# Step 4: Get the shortlisted variable.
```

```

shortlistedVars <- names(unlist(stepMod[[1]]))
shortlistedVars <- shortlistedVars[!shortlistedVars %in% "(Intercept)"] # remove intercept

# Show
print(shortlistedVars)
#> [1] "Temperature_Sandburg"   "Humidity"           "Temperature_ElMonte"
#> [4] "Month"                 "pressure_height"    "Inversion_base_height"

```

The selected model has the above 6 features in it.

But if you have too many features (> 100) in training data, then it might be a good idea to split the dataset into chunks of 10 variables each with Y as mandatory in each dataset. Loop through all the chunks and collect the best features.

We are doing it this way because some variables that came as important in a training data with fewer features may not show up in a linear reg model built on lots of features.

Finally, from a pool of shortlisted features (from small chunk models), run a full stepwise model to get the final set of selected features.

You can take this as a learning assignment to be solved within 20 minutes.

23.7 Relative Importance from Linear Regression

This technique is specific to linear regression models.

Relative importance can be used to assess which variables contributed how much in explaining the linear model's R-squared value. So, if you sum up the produced importances, it will add up to the model's R-sq value.

In essence, it is not directly a feature selection method, because you have already provided the features that go in the model. But after building the model, the `relaimpo` can provide a sense of how important each feature is in contributing to the R-sq, or in other words, in 'explaining the Y variable'.

So, how to calculate relative importance?

It is implemented in the `relaimpo` package. Basically, you build a linear regression model and pass that as the main argument to `calc.relimp()`. `relaimpo` has multiple options to compute the relative importance, but the recommended method is to use `type='lmg'`, as I have done below.

```

# install.packages('relaimpo')
library(relaimpo)
#> Loading required package: boot
#>
#> Attaching package: 'boot'
#> The following object is masked from 'package:lattice':
#>
#>     melanoma
#> The following object is masked from 'package:survival':
#>
#>     aml
#> Loading required package: survey
#> Loading required package: grid
#>
#> Attaching package: 'survey'
#> The following object is masked from 'package:graphics':

```

```
#>
#>     dotchart
#> Loading required package: mitools
#> This is the global version of package relaimpo.
#> If you are a non-US user, a version with the interesting additional metric pmvd is available
#> from Ulrike Groempings web site at prof.beuth-hochschule.de/groemping.

# Build linear regression model
model_formula = ozone_reading ~ Temperature_Sandburg + Humidity + Temperature_ElMonte + Month + pressure_height
lmMod <- lm(model_formula, data=trainData)

# calculate relative importance
relImportance <- calc.relimp(lmMod, type = "lmg", rela = F)

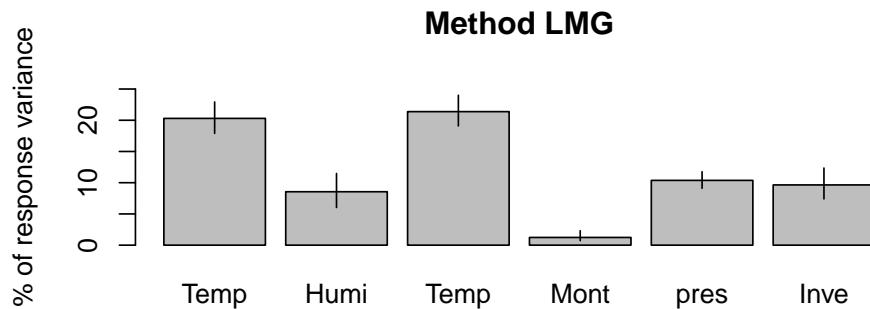
# Sort
cat('Relative Importances: \n')
#> Relative Importances:
sort(round(relImportance$lmg, 3), decreasing=TRUE)
#>   Temperature_ElMonte   Temperature_Sandburg      pressure_height
#>           0.214              0.203                 0.104
#>   Inversion_base_height       Humidity            Month
#>           0.096              0.086                 0.012
```

Additionally, you can use bootstrapping (using `boot.relimp`) to compute the confidence intervals of the produced relative importances.

```
bootsub <- boot.relimp(ozone_reading ~ Temperature_Sandburg + Humidity + Temperature_ElMonte + Month + pressure_height, b = 1000, type = 'lmg', rank = TRUE, diff = TRUE)

plot(booteval.relimp(bootsub, level=.95))
```

Relative importances for ozone_reading with 95% bootstrap confidence intervals



$R^2 = 71.5\%$, metrics are not normalized.

23.8 Recursive Feature Elimination (RFE)

Recursive feature elimination (rfe) offers a rigorous way to determine the important variables before you even feed them into a ML algo.

It can be implemented using the `rfe()` from caret package.

The `rfe()` also takes two important parameters.

- `sizes`
- `rfeControl`

So, what does `sizes` and `rfeControl` represent?

The `sizes` determines the number of most important features the `rfe` should iterate. Below, I have set the size as 1 to 5, 10, 15 and 18.

Secondly, the `rfeControl` parameter receives the output of the `rfeControl()`. You can set what type of variable evaluation algorithm must be used. Here, I have used random forests based `rfFuncs`. The `method='repeatedCV'` means it will do a repeated k-Fold cross validation with `repeats=5`.

Once complete, you get the accuracy and kappa for each model size you provided. The final selected model subset size is marked with a * in the rightmost selected column.

```
str(trainData)
#> 'data.frame': 366 obs. of 13 variables:
#> $ Month : int 1 1 1 1 1 1 1 1 1 ...
#> $ Day_of_month : int 1 2 3 4 5 6 7 8 9 10 ...
#> $ Day_of_week : int 4 5 6 7 1 2 3 4 5 6 ...
#> $ ozone_reading : num 3 3 3 5 5 6 4 4 6 7 ...
#> $ pressure_height : num 5480 5660 5710 5700 5760 5720 5790 5790 5700 5700 ...
#> $ Wind_speed : int 8 6 4 3 3 4 6 3 3 3 ...
#> $ Humidity : num 20 41 28 37 51 ...
#> $ Temperature_Sandburg : num 40.5 38 40 45 54 ...
#> $ Temperature_ElMonte : num 39.8 46.7 49.5 52.3 45.3 ...
#> $ Inversion_base_height: num 5000 4109 2693 590 1450 ...
#> $ Pressure_gradient : num -15 -14 -25 -24 25 15 -33 -28 23 -2 ...
#> $ Inversion_temperature: num 30.6 48 47.7 55 57 ...
#> $ Visibility : int 200 300 250 100 60 60 100 250 120 120 ...
```

```
tic()
set.seed(100)
options(warn=-1)

subsets <- c(1:5, 10, 15, 18)

ctrl <- rfeControl(functions = rfFuncs,
                    method = "repeatedcv",
                    repeats = 5,
                    verbose = FALSE)

lmProfile <- rfe(x=trainData[, c(1:3, 5:13)], y=trainData$ozone_reading,
                  sizes = subsets,
                  rfeControl = ctrl)
toc()
#> 91.881 sec elapsed
lmProfile
#>
#> Recursive feature selection
#>
#> Outer resampling method: Cross-Validated (10 fold, repeated 5 times)
#>
#> Resampling performance over subset size:
```

```

#>
#>   Variables RMSE Rsquared MAE RMSESD RsquaredSD MAESD Selected
#>       1 5.13    0.595 3.92  0.826      0.1275 0.586
#>       2 4.03    0.746 3.11  0.542      0.0743 0.416
#>       3 3.95    0.756 3.06  0.472      0.0670 0.380
#>       4 3.93    0.759 3.01  0.468      0.0683 0.361
#>       5 3.90    0.763 2.98  0.467      0.0659 0.350
#>      10 3.77    0.782 2.85  0.496      0.0734 0.393      *
#>      12 3.77    0.781 2.86  0.508      0.0756 0.401
#>
#>   The top 5 variables (out of 10):
#>      Temperature_ElMonte, Pressure_gradient, Temperature_Sandburg, Inversion_temperature, Humidity

```

So, it says, Temperature_ElMonte, Pressure_gradient, Temperature_Sandburg, Inversion_temperature, Humidity are the top 5 variables in that order.

And the best model size out of the provided models sizes (in subsets) is 10.

You can see all of the top 10 variables from ‘lmProfile\$optVariables’ that was created using `rfe` function above.

23.9 Genetic Algorithm

You can perform a supervised feature selection with genetic algorithms using the `gafs()`. This is **quite resource expensive** so consider that before choosing the number of iterations (iters) and the number of repeats in `gafsControl()`.

```

tic()
# Define control function
ga_ctrl <- gafsControl(functions = rfGA, # another option is `caretGA`.
                        method = "cv",
                        repeats = 3)

# Genetic Algorithm feature selection
set.seed(100)
ga_obj <- gafs(x=trainData[, c(1:3, 5:13)],
                y=trainData[, 4],
                iters = 3, # normally much higher (100+)
                gafsControl = ga_ctrl)
toc()
#> 636.08 sec elapsed
ga_obj
#>
#>   Genetic Algorithm Feature Selection
#>
#>   366 samples
#>   12 predictors
#>
#>   Maximum generations: 3
#>   Population per generation: 50
#>   Crossover probability: 0.8
#>   Mutation probability: 0.1
#>   Elitism: 0
#>

```

```

#> Internal performance values: RMSE, Rsquared
#> Subset selection driven to minimize internal RMSE
#>
#> External performance values: RMSE, Rsquared, MAE
#> Best iteration chose by minimizing external RMSE
#> External resampling method: Cross-Validated (10 fold)
#>
#> During resampling:
#>   * the top 5 selected variables (out of a possible 12):
#>     Month (100%), Pressure_gradient (100%), Temperature_ElMonte (100%), Humidity (80%), Visibility (80%)
#>   * on average, 6.8 variables were selected (min = 5, max = 9)
#>
#> In the final search using the entire training set:
#>   * 9 features selected at iteration 2 including:
#>     Month, Day_of_month, pressure_height, Wind_speed, Humidity ...
#>   * external performance at this iteration is
#>
#>     RMSE      Rsquared        MAE
#>     3.721      0.788       2.800

# Optimal variables
ga_obj$optVariables
#> [1] "Month"                  "Day_of_month"           "pressure_height"
#> [4] "Wind_speed"              "Humidity"                "Temperature_ElMonte"
#> [7] "Inversion_base_height"   "Pressure_gradient"      "Inversion_temperature"

```

'Month' 'Day_of_month' 'Wind_speed' 'Temperature_ElMonte' 'Pressure_gradient' 'Visibility'

So the optimal variables according to the genetic algorithms are listed above. But, I wouldn't use it just yet because, the above variant was tuned for only 3 iterations, which is quite low. I had to set it so low to save computing time.

23.10 Simulated Annealing

Simulated annealing is a global search algorithm that allows a suboptimal solution to be accepted in hope that a better solution will show up eventually.

It works by making small random changes to an initial solution and sees if the performance improved. The change is accepted if it improves, else it can still be accepted if the difference of performances meet an acceptance criteria.

In caret it has been implemented in the `safs()` which accepts a control parameter that can be set using the `safsControl()` function.

`safsControl` is similar to other control functions in caret (like you saw in rfe and ga), and additionally it accepts an `improve` parameter which is the number of iterations it should wait without improvement until the values are reset to previous iteration.

```

tic()
# Define control function
sa_ctrl <- safsControl(functions = rfSA,
                        method = "repeatedcv",
                        repeats = 3,
                        improve = 5) # n iterations without improvement before a reset

```

```

# Genetic Algorithm feature selection
set.seed(100)
sa_obj <- safs(x=trainData[, c(1:3, 5:13)],
                 y=trainData[, 4],
                 safsControl = sa_ctrl)
toc()
#> 109.61 sec elapsed
sa_obj
#>
#> Simulated Annealing Feature Selection
#>
#> 366 samples
#> 12 predictors
#>
#> Maximum search iterations: 10
#> Restart after 5 iterations without improvement (0.3 restarts on average)
#>
#> Internal performance values: RMSE, Rsquared
#> Subset selection driven to minimize internal RMSE
#>
#> External performance values: RMSE, Rsquared, MAE
#> Best iteration chose by minimizing external RMSE
#> External resampling method: Cross-Validated (10 fold, repeated 3 times)
#>
#> During resampling:
#>   * the top 5 selected variables (out of a possible 12):
#>     Temperature_Sandburg (80%), Month (66.7%), Pressure_gradient (66.7%), Temperature_ElMonte (63.3%)
#>   * on average, 6.5 variables were selected (min = 3, max = 11)
#>
#> In the final search using the entire training set:
#>   * 6 features selected at iteration 9 including:
#>     Day_of_week, pressure_height, Wind_speed, Humidity, Inversion_base_height ...
#>   * external performance at this iteration is
#>
#>     RMSE      Rsquared        MAE
#>     4.108      0.743       3.111

# Optimal variables
print(sa_obj$optVariables)
#> [1] "Day_of_week"           "pressure_height"        "Wind_speed"
#> [4] "Humidity"              "Inversion_base_height" "Pressure_gradient"

```

23.11 Information Value and Weights of Evidence

The Information Value can be used to judge how important a given categorical variable is in explaining the binary Y variable. It goes well with logistic regression and other classification models that can model binary variables.

Let's try to find out how important the categorical variables are in predicting if an individual will earn > 50k from the `adult.csv` dataset. Just run the code below to import the dataset.

```

library(InformationValue)
#>
```

```

#> Attaching package: 'InformationValue'
#> The following objects are masked from 'package:caret':
#>
#>     confusionMatrix, precision, sensitivity, specificity

# online data
# inputData <- read.csv("http://rstatistics.net/wp-content/uploads/2015/09/adult.csv")

inputData <- read.csv(file.path(data_raw_dir, "adult.csv"))
print(head(inputData))
#>   AGE      WORKCLASS FNLWGT EDUCATION EDUCATIONNUM      MARITALSTATUS
#> 1 39      State-gov  77516 Bachelor  13      Never-married
#> 2 50 Self-emp-not-inc 83311 Bachelor  13 Married-civ-spouse
#> 3 38      Private 215646 HS-grad    9      Divorced
#> 4 53      Private 234721 11th      7 Married-civ-spouse
#> 5 28      Private 338409 Bachelor  13 Married-civ-spouse
#> 6 37      Private 284582 Masters   14 Married-civ-spouse
#>   OCCUPATION RELATIONSHIP RACE      SEX CAPITALGAIN CAPITALLOSS
#> 1 Adm-clerical Not-in-family White  Male  2174          0
#> 2 Exec-managerial Husband White  Male  0          0
#> 3 Handlers-cleaners Not-in-family White  Male  0          0
#> 4 Handlers-cleaners Husband Black  Male  0          0
#> 5 Prof-specialty      Wife Black Female  0          0
#> 6 Exec-managerial      Wife White Female  0          0
#>   HOURSPERWEEK NATIVECOUNTRY ABOVE50K
#> 1      40 United-States  0
#> 2      13 United-States  0
#> 3      40 United-States  0
#> 4      40 United-States  0
#> 5      40 Cuba          0
#> 6      40 United-States  0

# Choose Categorical Variables to compute Info Value.
cat_vars <- c("WORKCLASS", "EDUCATION", "MARITALSTATUS", "OCCUPATION", "RELATIONSHIP", "RACE", "SEX", ...)

factor_vars <- cat_vars

# Init Output
df_iv <- data.frame(VARS=cat_vars, IV=numeric(length(cat_vars)), STRENGTH=character(length(cat_vars)), ...)

# Get Information Value for each variable
for (factor_var in factor_vars){
  df_iv[df_iv$VARS == factor_var, "IV"] <- InformationValue::IV(X=inputData[, factor_var], Y=inputData$)
  df_iv[df_iv$VARS == factor_var, "STRENGTH"] <- attr(InformationValue::IV(X=inputData[, factor_var], Y=...),
}

# Sort
df_iv <- df_iv[order(-df_iv$IV), ]

df_iv
#>           VARS      IV           STRENGTH
#> 5 RELATIONSHIP 1.5356 Highly Predictive
#> 3 MARITALSTATUS 1.3388 Highly Predictive

```

```
#> 4    OCCUPATION 0.7762  Highly Predictive
#> 2    EDUCATION 0.7411  Highly Predictive
#> 7    SEX 0.3033  Highly Predictive
#> 1    WORKCLASS 0.1634  Highly Predictive
#> 8    NATIVECOUNTRY 0.0794 Somewhat Predictive
#> 6    RACE 0.0693 Somewhat Predictive
```

Here is what the quantum of Information Value means:

Less than 0.02, then the predictor is not useful for modeling (separating the Goods from the Bads)

0.02 to 0.1, then the predictor has only a weak relationship. 0.1 to 0.3, then the predictor has a medium strength relationship. 0.3 or higher, then the predictor has a strong relationship. That was about IV. Then what is Weight of Evidence?

Weights of evidence can be useful to find out how important a given categorical variable is in explaining the ‘events’ (called ‘Goods’ in below table.)

The ‘Information Value’ of the categorical variable can then be derived from the respective WOE values.

IV=(perc good of all goods–perc bad of all bards) *WOE

The ‘WOETable’ below given the computation in more detail.

```
WOETable(X=inputData[, 'WORKCLASS'], Y=inputData$ABOVE50K)
#>          CAT GOODS BADS TOTAL PCT_G PCT_B      WOE      IV
#> 1          ?   191  1645  1836 0.02429 0.066545 -1.008 0.042574
#> 2    Federal-gov  371   589   960 0.04719 0.023827  0.683 0.015964
#> 3    Local-gov  617  1476  2093 0.07848 0.059709  0.273 0.005131
#> 4  Never-worked    7     7     7 0.00089 0.000283  1.146 0.000696
#> 5        Private 4963 17733 22696 0.63126 0.717354 -0.128 0.011006
#> 6  Self-emp-inc  622   494  1116 0.07911 0.019984  1.376 0.081363
#> 7 Self-emp-not-inc  724  1817  2541 0.09209 0.073503  0.225 0.004190
#> 8    State-gov  353   945  1298 0.04490 0.038228  0.161 0.001073
#> 9  Without-pay   14    14    14 0.00178 0.000566  1.146 0.001391
```

The total IV of a variable is the sum of IV’s of its categories.

23.12 DALEX Package

The DALEX is a powerful package that explains various things about the variables used in an ML model.

For example, using the `variable_dropout()` function you can find out how important a variable is based on a dropout loss, that is how much loss is incurred by removing a variable from the model.

Apart from this, it also has the `single_variable()` function that gives you an idea of how the model’s output will change by changing the values of one of the X’s in the model.

It also has the `single_prediction()` that can decompose a single model prediction so as to understand which variable caused what effect in predicting the value of Y.

```
library(randomForest)
#> randomForest 4.6-14
#> Type rfNews() to see new features/changes/bug fixes.
#>
#> Attaching package: 'randomForest'
#> The following object is masked from 'package:dplyr':
#>
```

```

#>      combine
#> The following object is masked from 'package:ranger':
#>
#>      importance
#> The following object is masked from 'package:ggplot2':
#>
#>      margin
library(DALEX)
#> Welcome to DALEX (version: 0.3.0).
#> This is a plain DALEX. Use 'install_dependencies()' to get all required packages.
#>
#> Attaching package: 'DALEX'
#> The following object is masked from 'package:dplyr':
#>
#>      explain

# Load data
# inputData <- read.csv("http://rstatistics.net/wp-content/uploads/2015/09/adult.csv")

inputData <- read.csv(file.path(data_raw_dir, "adult.csv"))

# Train random forest model
rf_mod <- randomForest(factor(ABOVE50K) ~ ., data=inputData, ntree=100)
rf_mod
#>
#> Call:
#> randomForest(formula = factor(ABOVE50K) ~ ., data = inputData,      ntree = 100)
#>           Type of random forest: classification
#>           Number of trees: 100
#> No. of variables tried at each split: 3
#>
#>           OOB estimate of error rate: 13.6%
#> Confusion matrix:
#>          0    1 class.error
#> 0 23051 1669    0.0675
#> 1 2754 5087    0.3512

# Variable importance with DALEX
explained_rf <- explain(rf_mod, data=inputData, y=inputData$ABOVE50K)

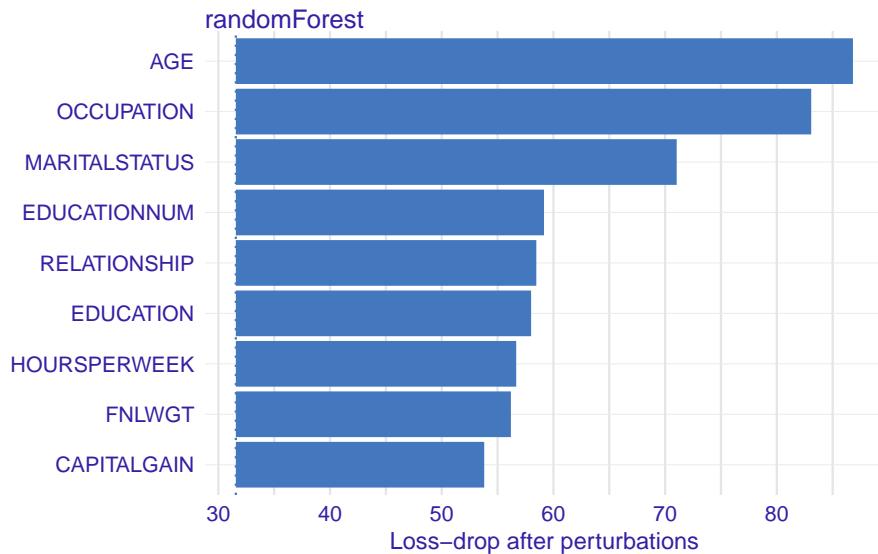
# Get the variable importances
varimps = variable_dropout(explained_rf, type='raw')

print(varimps)
#>      variable dropout_loss      label
#> 1 _full_model_        31.6 randomForest
#> 2 ABOVE50K        31.6 randomForest
#> 3 RACE            36.6 randomForest
#> 4 SEX              39.4 randomForest
#> 5 CAPITALLOSS     39.9 randomForest
#> 6 NATIVECOUNTRY   40.3 randomForest
#> 7 WORKCLASS       51.0 randomForest
#> 8 CAPITALGAIN     53.8 randomForest

```

```
#> 9      FNLWGT      56.2 randomForest
#> 10     HOURSPERWEEK 56.7 randomForest
#> 11     EDUCATION    58.0 randomForest
#> 12     RELATIONSHIP 58.5 randomForest
#> 13     EDUCATIONNUM 59.2 randomForest
#> 14     MARITALSTATUS 71.0 randomForest
#> 15     OCCUPATION   83.1 randomForest
#> 16     AGE          86.8 randomForest
#> 17     _baseline_   304.4 randomForest
```

```
plot(varimps)
```



23.13 Conclusion

Hope you find these methods useful. As it turns out different methods showed different variables as important, or at least the degree of importance changed. This need not be a conflict, because each method gives a different perspective of how the variable can be useful depending on how the algorithms learn $\mathbf{Y} \sim \mathbf{x}$. So its cool.

If you find any code breaks or bugs, report the issue here or just write it below.

Chapter 24

Imputting missing values with Random Forest

24.1 Flu Prediction. fluH7N9_china_2013 dataset

Source: https://shirinsplayground.netlify.com/2018/04/flu_prediction/

```
library(outbreaks)
library(tidyverse)
library(plyr)
library(mice)
library(caret)
library(purrr)
library("tibble")
library("dplyr")
library("tidyr")
```

Since I migrated my blog from Github Pages to blogdown and Netlify, I wanted to start migrating (most of) my old posts too - and use that opportunity to update them and make sure the code still works.

Here I am updating my very first machine learning post from 27 Nov 2016: Can we predict flu deaths with Machine Learning and R?. Changes are marked as bold comments.

The main changes I made are:

- using the tidyverse more consistently throughout the analysis
- focusing on comparing multiple imputations from the mice package, rather than comparing different algorithms
- using purrr, map(), nest() and unnest() to model and predict the machine learning algorithm over the different imputed datasets

Among the many nice R packages containing data collections is the **outbreaks** package. It contains a dataset on epidemics and among them is data from the 2013 outbreak of influenza A H7N9 in China as analysed by Kucharski et al. (2014):

A. Kucharski, H. Mills, A. Pinsent, C. Fraser, M. Van Kerkhove, C. A. Donnelly, and S. Riley.
2014. Distinguishing between reservoir exposure and human-to-human transmission for emerging

pathogens using case onset data. PLOS Currents Outbreaks. Mar 7, edition 1. doi: 10.1371/currents.outbreaks.e1473d9bfc99d080ca242139a06c455f.

A. Kucharski, H. Mills, A. Pinsent, C. Fraser, M. Van Kerkhove, C. A. Donnelly, and S. Riley. 2014. Data from: Distinguishing between reservoir exposure and human-to-human transmission for emerging pathogens using case onset data. Dryad Digital Repository. <http://dx.doi.org/10.5061/dryad.2g43n>.

I will be using their data as an example to show how to use Machine Learning algorithms for predicting disease outcome.

24.2 The data

The dataset contains case ID, date of onset, date of hospitalization, date of outcome, gender, age, province and of course outcome: Death or Recovery.

24.2.1 Pre-processing

Change: variable names (i.e. column names) have been renamed, dots have been replaced with underscores, letters are all lower case now.

Change: I am using the tidyverse notation more consistently.

First, I'm doing some preprocessing, including:

- renaming missing data as NA
- adding an ID column
- setting column types
- gathering date columns
- changing factor names of dates (to make them look nicer in plots) and of province (to combine provinces with few cases)

```
from1 <- c("date_of_onset", "date_of_hospitalisation", "date_of_outcome")
to1   <- c("date of onset", "date of hospitalisation", "date of outcome")
from2 <- c("Anhui", "Beijing", "Fujian", "Guangdong", "Hebei", "Henan",
          "Hunan", "Jiangxi", "Shandong", "Taiwan")
to2   <- rep("Other", 10)

fluH7N9_china_2013$age[which(fluH7N9_china_2013$age == "?")] <- NA
fluH7N9_china_2013_gather <- fluH7N9_china_2013 %>%
  mutate(case_id = paste("case", case_id, sep = "_"),
        age = as.numeric(age)) %>%
  gather(Group, Date, date_of_onset:date_of_outcome) %>%
  mutate(Province = as.factor(mapvalues(Group, from = from1, to = to1)),
        Province = mapvalues(Province, from = from2, to = to2))

fluH7N9_china_2013 <- as.tibble(fluH7N9_china_2013)
#> Warning: `as.tibble()` is deprecated, use `as_tibble()` (but mind the new semantics).
#> This warning is displayed once per session.
fluH7N9_china_2013_gather <- as.tibble(fluH7N9_china_2013_gather)
print(fluH7N9_china_2013)
#> # A tibble: 136 x 8
#>   case_id date_of_onset date_of_hospitalisation date_of_outcome outcome gender
#>   <fct>    <date>           <date>            <date>       <fct>    <fct>
```

```
#> 1 1      2013-02-19    NA          2013-03-04    Death   m
#> 2 2      2013-02-27    2013-03-03  2013-03-10    Death   m
#> 3 3      2013-03-09    2013-03-19  2013-04-09    Death   f
#> 4 4      2013-03-19    2013-03-27  NA          <NA>    f
#> 5 5      2013-03-19    2013-03-30  2013-05-15    Recover f
#> 6 6      2013-03-21    2013-03-28  2013-04-26    Death   f
#> # ... with 130 more rows, and 2 more variables: age <fct>, province <fct>
```

I'm also adding a third gender level for unknown gender

```
levels(fluH7N9_china_2013_gather$gender) <- 
  c(levels(fluH7N9_china_2013_gather$gender), "unknown")
fluH7N9_china_2013_gather$gender[is.na(fluH7N9_china_2013_gather$gender)] <- "unknown"
print(fluH7N9_china_2013_gather)
#> # A tibble: 408 x 7
#>   case_id outcome gender  age province Group      Date
#>   <chr>    <fct>  <fct> <dbl> <fct>   <fct>    <date>
#> 1 case_1   Death   m     58 Shanghai date of onset 2013-02-19
#> 2 case_2   Death   m     7  Shanghai date of onset 2013-02-27
#> 3 case_3   Death   f     11 Other    date of onset 2013-03-09
#> 4 case_4   <NA>    f     18 Jiangsu   date of onset 2013-03-19
#> 5 case_5   Recover f     20 Jiangsu   date of onset 2013-03-19
#> 6 case_6   Death   f     9  Jiangsu   date of onset 2013-03-21
#> # ... with 402 more rows
```

For plotting, I am defining a custom ggplot2 theme:

```
my_theme <- function(base_size = 12, base_family = "sans"){
  theme_minimal(base_size = base_size, base_family = base_family) +
  theme(
    axis.text = element_text(size = 12),
    axis.text.x = element_text(angle = 45, vjust = 0.5, hjust = 0.5),
    axis.title = element_text(size = 14),
    panel.grid.major = element_line(color = "grey"),
    panel.grid.minor = element_blank(),
    panel.background = element_rect(fill = "aliceblue"),
    strip.background = element_rect(fill = "lightgrey", color = "grey", size = 1),
    strip.text = element_text(face = "bold", size = 12, color = "black"),
    legend.position = "bottom",
    legend.justification = "top",
    legend.box = "horizontal",
    legend.box.background = element_rect(colour = "grey50"),
    legend.background = element_blank(),
    panel.border = element_rect(color = "grey", fill = NA, size = 0.5)
  )
}
```

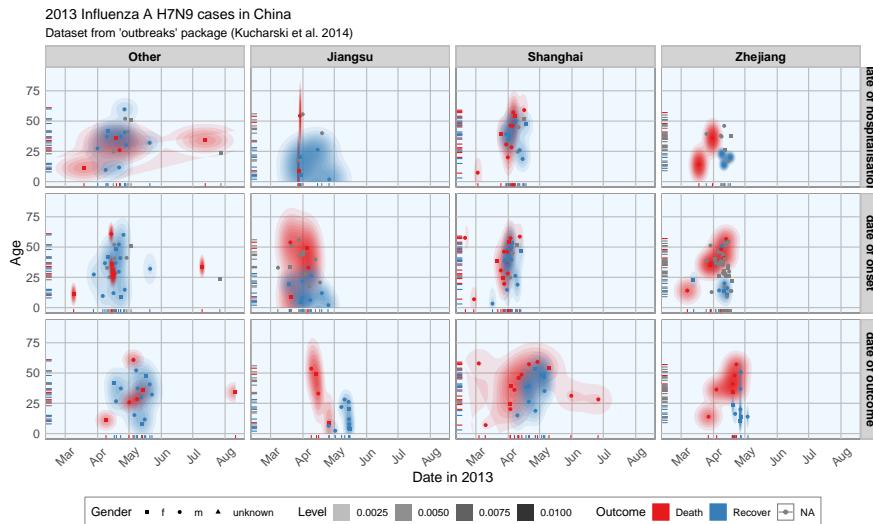
And use that theme to visualize the data:

```
ggplot(data = fluH7N9_china_2013_gather, aes(x = Date, y = age, fill = outcome)) +
  stat_density2d(aes(alpha = ..level..), geom = "polygon") +
  geom_jitter(aes(color = outcome, shape = gender), size = 1.5) +
  geom_rug(aes(color = outcome)) +
  scale_y_continuous(limits = c(0, 90)) +
  labs(
    fill = "Outcome",
```

```

color = "Outcome",
alpha = "Level",
shape = "Gender",
x = "Date in 2013",
y = "Age",
title = "2013 Influenza A H7N9 cases in China",
subtitle = "Dataset from 'outbreaks' package (Kucharski et al. 2014)",
caption = ""
) +
facet_grid(Group ~ province) +
my_theme() +
scale_shape_manual(values = c(15, 16, 17)) +
scale_color_brewer(palette="Set1", na.value = "grey50") +
scale_fill_brewer(palette="Set1")
#> Warning: Removed 149 rows containing non-finite values (stat_density2d).
#> Warning: Removed 149 rows containing missing values (geom_point).

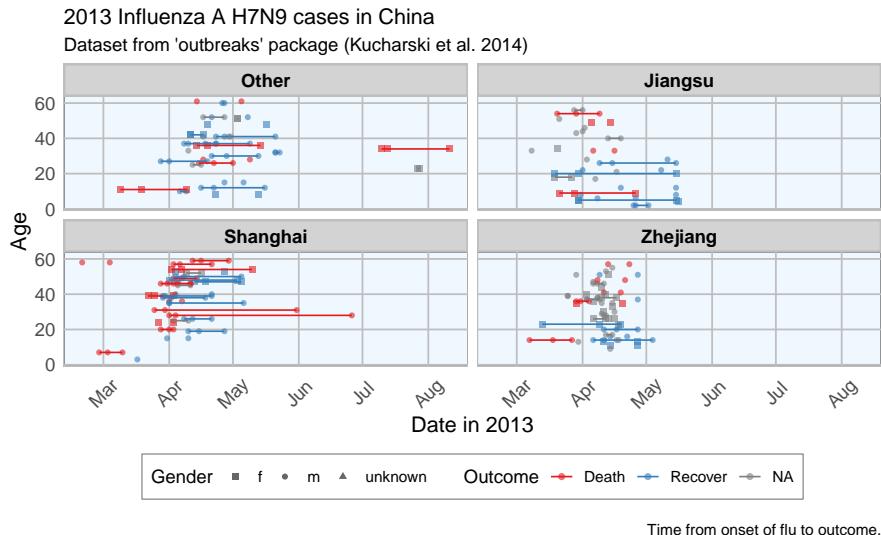
```



```

ggplot(data = fluH7N9_china_2013_gather, aes(x = Date, y = age, color = outcome)) +
  geom_point(aes(color = outcome, shape = gender), size = 1.5, alpha = 0.6) +
  geom_path(aes(group = case_id)) +
  facet_wrap(~ province, ncol = 2) +
  my_theme() +
  scale_shape_manual(values = c(15, 16, 17)) +
  scale_color_brewer(palette="Set1", na.value = "grey50") +
  scale_fill_brewer(palette="Set1") +
  labs(
    color = "Outcome",
    shape = "Gender",
    x = "Date in 2013",
    y = "Age",
    title = "2013 Influenza A H7N9 cases in China",
    subtitle = "Dataset from 'outbreaks' package (Kucharski et al. 2014)",
    caption = "\nTime from onset of flu to outcome."
  )
#> Warning: Removed 149 rows containing missing values (geom_point).
#> Warning: Removed 122 rows containing missing values (geom_path).

```



24.3 Features

In machine learning-speak features are what we call the variables used for model training. Using the right features dramatically influences the accuracy and success of your model.

For this example, I am keeping age, but I am also generating new features from the date information and converting gender and province into numerical values.

```
delta_dates <- function(onset, ref) {
  d2 = as.Date(as.character(onset), format = "%Y-%m-%d")
  d1 = as.Date(as.character(ref), format = "%Y-%m-%d")
  as.numeric(as.character(gsub(" days", "", d1 - d2)))
}

dataset <- fluH7N9_china_2013 %>%
  mutate(
    hospital = as.factor(ifelse(is.na(date_of_hospitalisation), 0, 1)),
    gender_f = as.factor(ifelse(gender == "f", 1, 0)),
    province_Jiangsu = as.factor(ifelse(province == "Jiangsu", 1, 0)),
    province_Shanghai = as.factor(ifelse(province == "Shanghai", 1, 0)),
    province_Zhejiang = as.factor(ifelse(province == "Zhejiang", 1, 0)),
    province_other = as.factor(ifelse(province == "Zhejiang"
                                         | province == "Jiangsu"
                                         | province == "Shanghai", 0, 1)),

    days_onset_to_outcome = delta_dates(date_of_onset, date_of_outcome),
    days_onset_to_hospital = delta_dates(date_of_onset, date_of_hospitalisation),
    age = age,
    early_onset = as.factor(ifelse(date_of_onset <
                                         summary(date_of_onset)[[3]], 1, 0)),
    early_outcome = as.factor(ifelse(date_of_outcome <
                                         summary(date_of_outcome)[[3]], 1, 0))
  ) %>%
  subset(select = -c(2:4, 6, 8))

rownames(dataset) <- dataset$case_id
```

```
#> Warning: Setting row names on a tibble is deprecated.
dataset[, -2] <- as.numeric(as.matrix(dataset[, -2]))
print(dataset)
#> # A tibble: 136 x 13
#>   case_id outcome  age hospital gender_f province_Jiangsu province_Shangh~
#> * <dbl> <fct>    <dbl>    <dbl>    <dbl>           <dbl>           <dbl>
#> 1     1 Death      87        0        0            0            1
#> 2     2 Death      27        1        0            0            1
#> 3     3 Death      35        1        1            0            0
#> 4     4 <NA>       45        1        1            1            0
#> 5     5 Recover     48        1        1            1            0
#> 6     6 Death      32        1        1            1            0
#> # ... with 130 more rows, and 6 more variables: province_Zhejiang <dbl>,
#> #   province_other <dbl>, days_onset_to_outcome <dbl>,
#> #   days_onset_to_hospital <dbl>, early_onset <dbl>, early_outcome <dbl>

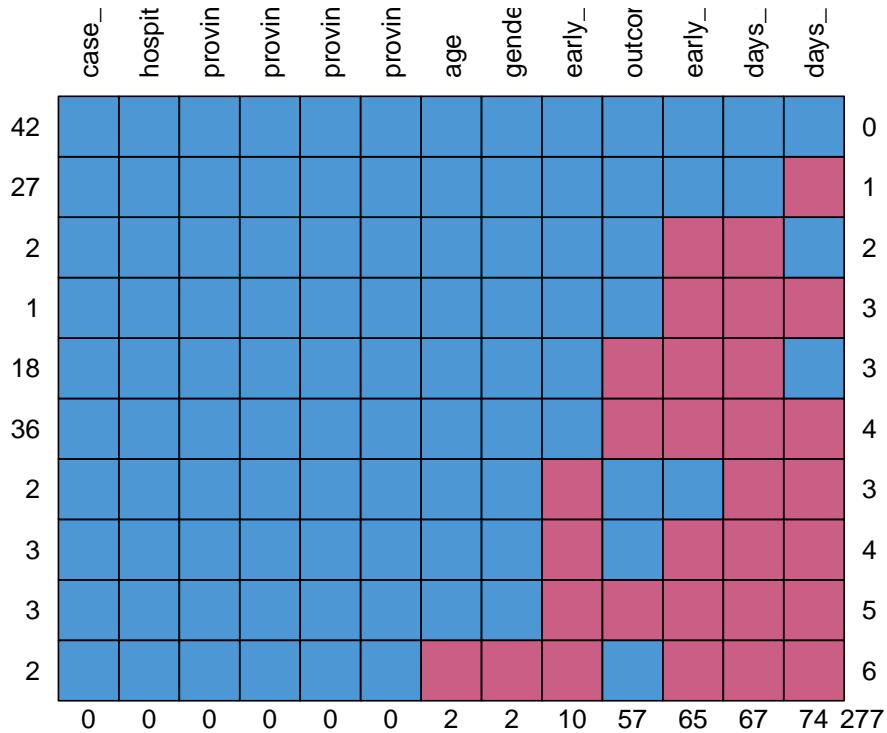
summary(dataset$outcome)
#>   Death Recover   NA's
#>     32      47      57
```

24.4 Imputing missing values

I am using the `mice` package for imputing missing values

Note: Since publishing this blogpost I learned that the idea behind using mice is to compare different imputations to see how stable they are, instead of picking one imputed set as fixed for the remainder of the analysis. Therefore, I changed the focus of this post a little bit: in the old post I compared many different algorithms and their outcome; in this updated version I am only showing the **Random Forest** algorithm and focus on **comparing the different imputed datasets**. I am ignoring feature importance and feature plots because nothing changed compared to the old post.

```
# plot the missing data in a matrix by variables
md_pattern <- md.pattern(dataset, rotate.names = TRUE)
```



```
dataset_impute <- mice(data = dataset[, -2], print = FALSE)
#> Warning: Number of logged events: 150
```

24.4.1 Generate a dataframe of five imputing strategies

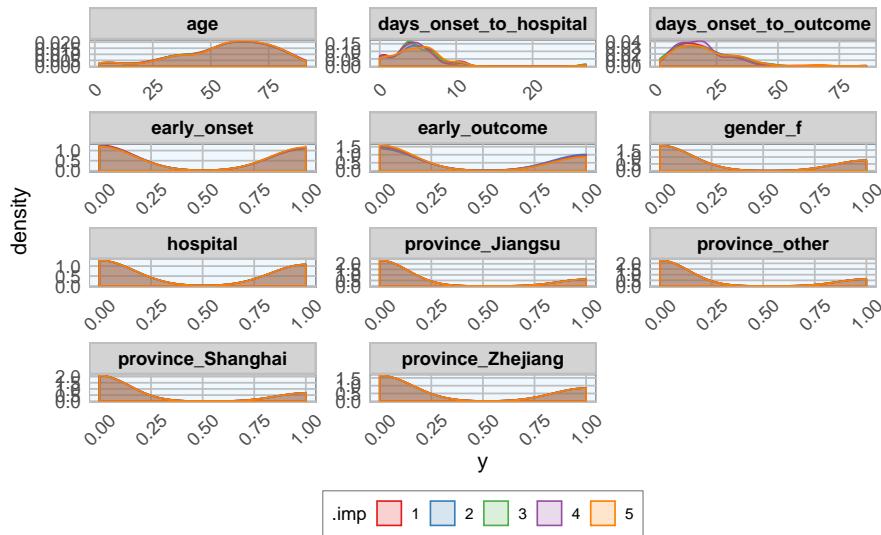
- by default, mice() calculates five ($m = 5$) imputed data sets
- we can combine them all in one output with the complete("long") function
- I did not want to impute missing values in the outcome column, so I have to merge it back in with the imputed data

```
# c(1,2): case_id, outcome
datasets_complete <- right_join(dataset[, c(1, 2)],
                                 complete(dataset_impute, "long"),
                                 by = "case_id") %>%
  mutate(.imp = as.factor(.imp)) %>%
  select(-.id) %>%
  print()
#> # A tibble: 680 x 14
#>   case_id outcome .imp     age hospital gender_f province_Jiangsu
#>   <dbl> <fct>   <fct> <dbl>    <dbl>    <dbl>    <dbl>
#> 1      1 Death    1      87      0      0      0
#> 2      2 Death    1      27      1      0      0
#> 3      3 Death    1      35      1      1      0
#> 4      4 <NA>     1      45      1      1      1
#> 5      5 Recover  1      48      1      1      1
#> 6      6 Death    1      32      1      1      1
#> # ... with 674 more rows, and 7 more variables: province_Shanghai <dbl>,
#> #   province_Zhejiang <dbl>, province_other <dbl>,
#> #   days_onset_to_outcome <dbl>, days_onset_to_hospital <dbl>,
#> #   early_onset <dbl>, early_outcome <dbl>
```

Let's compare the distributions of the five different imputed datasets:

24.4.2 plot effect of imputing on features

```
datasets_complete %>%
  gather(x, y, age:early_outcome) %>%
  ggplot(aes(x = y, fill = .imp, color = .imp)) +
  geom_density(alpha = 0.20) +
  facet_wrap(~ x, ncol = 3, scales = "free") +
  scale_fill_brewer(palette="Set1", na.value = "grey50") +
  scale_color_brewer(palette="Set1", na.value = "grey50") +
  my_theme()
```



24.5 Test, train and validation data sets

Now, we can go ahead with machine learning!

The dataset contains a few missing values in the outcome column; those will be the test set used for final predictions (see the old blog post for this).

```
length(which(is.na(datasets_complete$outcome)))
length(which(!is.na(datasets_complete$outcome)))
#> [1] 285
#> [1] 395

train_index <- which(is.na(datasets_complete$outcome))
train_data <- datasets_complete[-train_index, ]
test_data <- datasets_complete[train_index, -2] # remove variable outcome

print(train_data)
#> # A tibble: 395 x 14
#>   case_id outcome .imp    age hospital gender_f province_Jiangsu
#>   <dbl> <fct>   <dbl> <dbl>     <dbl>      <dbl>           <dbl>
#> 1       1 Death     1     87      0        0             0
#> 2       2 Death     1     27      1        0             0
```

```

#> 3      3 Death    1      35      1      1          0
#> 4      5 Recover   1      48      1      1          1
#> 5      6 Death    1      32      1      1          1
#> 6      7 Death    1      83      1      0          1
#> # ... with 389 more rows, and 7 more variables: province_Shanghai <dbl>,
#> #   province_Zhejiang <dbl>, province_other <dbl>,
#> #   days_onset_to_outcome <dbl>, days_onset_to_hospital <dbl>,
#> #   early_onset <dbl>, early_outcome <dbl>

# outcome variable removed
print(test_data)
#> # A tibble: 285 x 13
#>   case_id .imp     age hospital gender_f province_Jiangsu province_Shangh~
#>   <dbl> <fct> <dbl>     <dbl>     <dbl>           <dbl>           <dbl>
#> 1     4 1     45       1       1           1           0
#> 2     9 1     67       1       0           0           0
#> 3    15 1     61       0       1           1           0
#> 4    16 1     79       0       0           1           0
#> 5    22 1     85       1       0           1           0
#> 6    28 1     79       0       0           0           0
#> # ... with 279 more rows, and 6 more variables: province_Zhejiang <dbl>,
#> #   province_other <dbl>, days_onset_to_outcome <dbl>,
#> #   days_onset_to_hospital <dbl>, early_onset <dbl>, early_outcome <dbl>

```

The remainder of the data will be used for modeling. Here, I am splitting the data into 70% training and 30% test data.

Because I want to model each imputed dataset separately, I am using the `nest()` and `map()` functions.

```

train_data_nest <- train_data %>%
  group_by(.imp) %>%
  nest() %>%
  print()
#> # A tibble: 5 x 2
#>   .imp   data
#>   <fct> <list>
#> 1 1    <tibble [79 x 13]>
#> 2 2    <tibble [79 x 13]>
#> 3 3    <tibble [79 x 13]>
#> 4 4    <tibble [79 x 13]>
#> 5 5    <tibble [79 x 13]>

# split the training data in validation training and validation test
set.seed(42)
val_data <- train_data_nest %>%
  mutate(val_index = map(data, ~ createDataPartition(.\$outcome,
                                                    p = 0.7,
                                                    list = FALSE)),
        val_train_data = map2(data, val_index, ~ .x[.y, ]),
        val_test_data = map2(data, val_index, ~ .x[-.y, ])) %>%
  print()
#> # A tibble: 5 x 5
#>   .imp   data      val_index      val_train_data      val_test_data
#>   <fct> <list>      <list>      <list>      <list>
#> 1 1    <tibble [79 x ~ <int[,1]> [56 x ~ <tibble [56 x 13~ <tibble [23 x 1~
```

```
#> 2 2      <tibble [79 x ~ <int[,1] [56 x ~ <tibble [56 x 13~ <tibble [23 x 1~  
#> 3 3      <tibble [79 x ~ <int[,1] [56 x ~ <tibble [56 x 13~ <tibble [23 x 1~  
#> 4 4      <tibble [79 x ~ <int[,1] [56 x ~ <tibble [56 x 13~ <tibble [23 x 1~  
#> 5 5      <tibble [79 x ~ <int[,1] [56 x ~ <tibble [56 x 13~ <tibble [23 x 1~
```

24.6 Machine Learning algorithms

24.6.1 Random Forest

To make the code tidier, I am first defining the modeling function with the parameters I want.

```
model_function <- function(df) {  
  caret::train(outcome ~ .,  
               data = df,  
               method = "rf",  
               preProcess = c("scale", "center"),  
               trControl = trainControl(method = "repeatedcv",  
                                         number = 5,  
                                         repeats = 3,  
                                         verboseIter = FALSE))  
}
```

24.6.2 Add model and prediction to nested dataframe and calculate

Next, I am using the nested tibble from before to map() the model function, predict the outcome and calculate confusion matrices.

24.6.2.1 add model list-column

```
val_data_model <- val_data %>%  
  mutate(model = map(val_train_data, ~ model_function(.x))) %>%  
  select(-val_index) %>%  
  print()  
#> # A tibble: 5 x 5  
#>   .imp  data          val_train_data    val_test_data     model  
#>   <fct> <list>        <list>           <list>           <list>  
#> 1 1   <tibble [79 x 13]> <tibble [56 x 13]> <tibble [23 x 13]> <train>  
#> 2 2   <tibble [79 x 13]> <tibble [56 x 13]> <tibble [23 x 13]> <train>  
#> 3 3   <tibble [79 x 13]> <tibble [56 x 13]> <tibble [23 x 13]> <train>  
#> 4 4   <tibble [79 x 13]> <tibble [56 x 13]> <tibble [23 x 13]> <train>  
#> 5 5   <tibble [79 x 13]> <tibble [56 x 13]> <tibble [23 x 13]> <train>
```

24.6.2.2 add prediction and confusion matrix list-columns

```
set.seed(42)  
val_data_model <- val_data_model %>%  
  mutate(  
    predict = map2(model, val_test_data, ~
```

```

        data.frame(prediction = predict(.x, .y[, -2]))),
predict_prob = map2(model, val_test_data, ~
  data.frame(outcome = .y[, 2],
             prediction = predict(.x, .y[, -2], type = "prob"))),
confusion_matrix = map2(val_test_data, predict, ~
  confusionMatrix(.x$outcome, .y$prediction)),
confusion_matrix_tbl = map(confusion_matrix, ~ as.tibble(.x$table)) %>%
print()
#> # A tibble: 5 x 9
#>   .imp  data  val_train_data  val_test_data  model  predict  predict_prob
#>   <fct> <lis> <list>       <list>       <list> <list>
#> 1 1    <tib> <tibble [56 x~ <tibble [23 ~ <tra~ <df[,1~ <df[,3] [23~
#> 2 2    <tib> <tibble [56 x~ <tibble [23 ~ <tra~ <df[,1~ <df[,3] [23~
#> 3 3    <tib> <tibble [56 x~ <tibble [23 ~ <tra~ <df[,1~ <df[,3] [23~
#> 4 4    <tib> <tibble [56 x~ <tibble [23 ~ <tra~ <df[,1~ <df[,3] [23~
#> 5 5    <tib> <tibble [56 x~ <tibble [23 ~ <tra~ <df[,1~ <df[,3] [23~
#> # ... with 2 more variables: confusion_matrix <list>,
#> #   confusion_matrix_tbl <list>

```

Finally, we have a nested dataframe of 5 rows or cases, one per imputing strategy with its corresponding models and prediction results.

24.7 Comparing accuracy of models

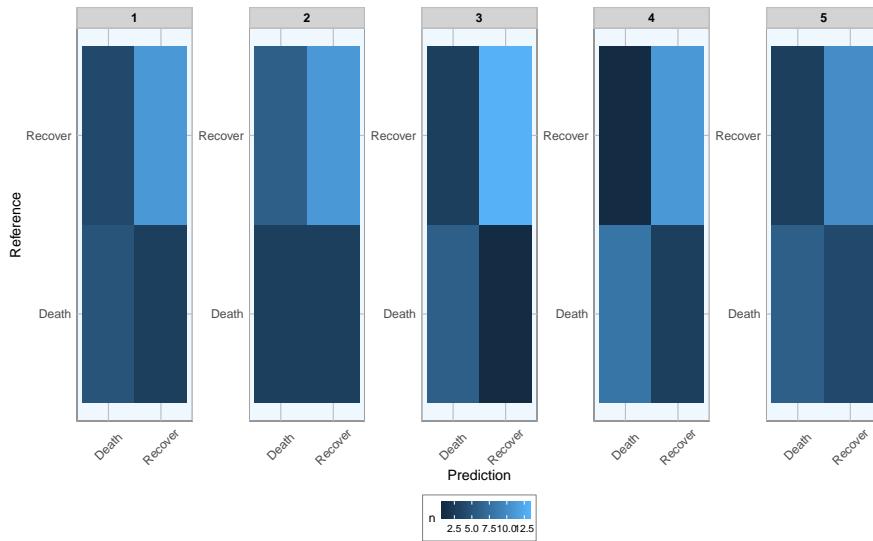
To compare how the different imputations did, I am plotting the confusion matrices:

```

val_data_model_unnest <- val_data_model %>%
  unnest(confusion_matrix_tbl) %>%
  print()
#> # A tibble: 20 x 4
#>   .imp  Prediction Reference     n
#>   <fct> <chr>      <chr>     <int>
#> 1 1    Death       Death      5
#> 2 1    Recover     Death      3
#> 3 1    Death       Recover    4
#> 4 1    Recover     Recover   11
#> 5 2    Death       Death      3
#> 6 2    Recover     Death      3
#> # ... with 14 more rows

val_data_model_unnest %>%
  ggplot(aes(x = Prediction, y = Reference, fill = n)) +
  facet_wrap(~ .imp, ncol = 5, scales = "free") +
  geom_tile() +
  my_theme()

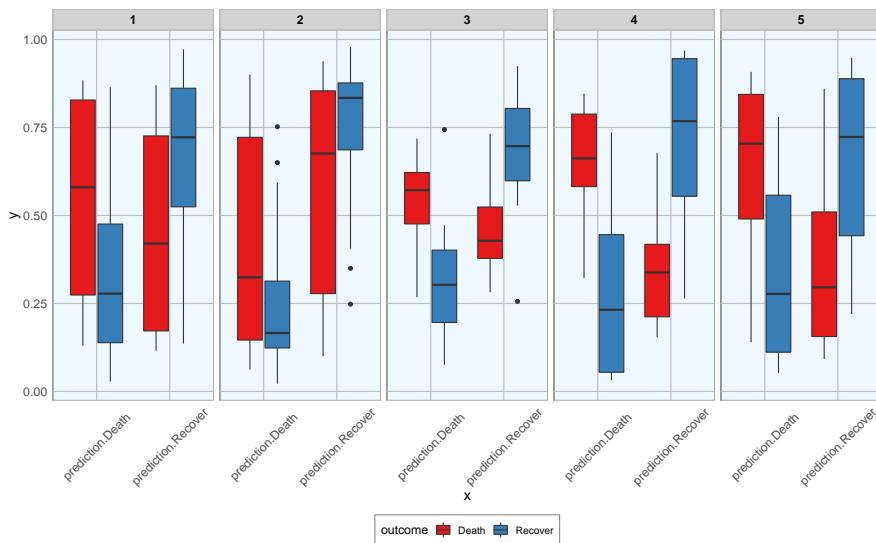
```



and the prediction probabilities for correct and wrong predictions:

```
val_data_model_gather <- val_data_model %>%
  unnest(predict_prob) %>%
  gather(x, y, prediction.Death:prediction.Recover) %>%
  print()
#> # A tibble: 230 x 4
#>   .imp outcome x           y
#>   <fct> <fct>  <chr>     <dbl>
#> 1 1     Death   prediction.Death 0.758
#> 2 1     Recover prediction.Death 0.864
#> 3 1     Death   prediction.Death 0.828
#> 4 1     Death   prediction.Death 0.828
#> 5 1     Recover prediction.Death 0.342
#> 6 1     Recover prediction.Death 0.552
#> # ... with 224 more rows
```

```
val_data_model_gather %>%
  ggplot(aes(x = x, y = y, fill = outcome)) +
  facet_wrap(~ .imp, ncol = 5) +
  geom_boxplot() +
  scale_fill_brewer(palette="Set1", na.value = "grey50") +
  my_theme()
```



Hope, you found that example interesting and helpful!

```
sessionInfo()
#> R version 3.6.0 (2019-04-26)
#> Platform: x86_64-pc-linux-gnu (64-bit)
#> Running under: Ubuntu 18.04.3 LTS
#>
#> Matrix products: default
#> BLAS/LAPACK: /usr/lib/x86_64-linux-gnu/libopenblas-r0.2.20.so
#>
#> locale:
#> [1] LC_CTYPE=en_US.UTF-8          LC_NUMERIC=C
#> [3] LC_TIME=en_US.UTF-8          LC_COLLATE=en_US.UTF-8
#> [5] LC_MONETARY=en_US.UTF-8       LC_MESSAGES=en_US.UTF-8
#> [7] LC_PAPER=en_US.UTF-8         LC_NAME=C
#> [9] LC_ADDRESS=C                 LC_TELEPHONE=C
#> [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
#>
#> attached base packages:
#> [1] stats      graphics   grDevices  utils      datasets   methods    base
#>
#> other attached packages:
#> [1] caret_6.0-84     mice_3.4.0     lattice_0.20-38  plyr_1.8.4
#> [5] forcats_0.4.0    stringr_1.4.0   dplyr_0.8.0.1   purrr_0.3.2
#> [9] readr_1.3.1      tidyr_0.8.3    tibble_2.1.1    ggplot2_3.1.1
#> [13] tidyverse_1.2.1   outbreaks_1.5.0  logging_0.9-107
#>
#> loaded via a namespace (and not attached):
#> [1] nlme_3.1-139      lubridate_1.7.4   RColorBrewer_1.1-2
#> [4] httr_1.4.0        rprojroot_1.3-2   tools_3.6.0
#> [7] backports_1.1.4   utf8_1.1.4       R6_2.4.0
#> [10] rpart_4.1-15     lazyeval_0.2.2   colorspace_1.4-1
#> [13] jomo_2.6-7       nnet_7.3-12     withr_2.1.2
#> [16] tidyselect_0.2.5  compiler_3.6.0   cli_1.1.0
#> [19] rvest_0.3.3      xml2_1.2.0      labeling_0.3
#> [22] bookdown_0.10    scales_1.0.0    randomForest_4.6-14
#> [25] digest_0.6.18    minqa_1.2.4    rmarkdown_1.12
```

```
#> [28] pkgconfig_2.0.2      htmltools_0.3.6      lme4_1.1-21
#> [31] rlang_0.3.4        readxl_1.3.1       rstudioapi_0.10
#> [34] generics_0.0.2     jsonlite_1.6       ModelMetrics_1.2.2
#> [37] magrittr_1.5       Matrix_1.2-17      Rcpp_1.0.1
#> [40] munsell_0.5.0      fansi_0.4.0        stringi_1.4.3
#> [43] yaml_2.2.0         MASS_7.3-51.4      recipes_0.1.5
#> [46] grid_3.6.0         parallel_3.6.0    mitml_0.3-7
#> [49] crayon_1.3.4      haven_2.1.0        splines_3.6.0
#> [52] hms_0.4.2          zeallot_0.1.0      knitr_1.22
#> [55] pillar_1.4.0      boot_1.3-22        reshape2_1.4.3
#> [58] codetools_0.2-16    stats4_3.6.0       pan_1.6
#> [61] glue_1.3.1         evaluate_0.13     data.table_1.12.2
#> [64] modelr_0.1.4       vctrs_0.1.0        nloptr_1.2.1
#> [67] foreach_1.4.4      cellranger_1.1.0   gtable_0.3.0
#> [70] assertthat_0.2.1   xfun_0.6           gower_0.2.0
#> [73] prodlim_2018.04.18 broom_0.5.2        e1071_1.7-1
#> [76] class_7.3-15      survival_2.44-1.1 timeDate_3043.102
#> [79] iterators_1.0.10   lava_1.6.5         ipred_0.9-9
```

Part IV

Meta ML

Chapter 25

PCA: prcomp vs princomp

<http://www.sthda.com/english/articles/31-principal-component-methods-in-r-practical-guide/118-principal-component-analysis-pca/>

25.1 General methods for principal component analysis

There are two general methods to perform PCA in R :

- Spectral decomposition which examines the covariances / correlations between variables
- Singular value decomposition which examines the covariances / correlations between individuals

The function `princomp()` uses the spectral decomposition approach. The functions `prcomp()` and `PCA()`[FactoMineR] use the singular value decomposition (SVD).

25.2 prcomp() and princomp() functions

The simplified format of these 2 functions are :

```
prcomp(x, scale = FALSE)
princomp(x, cor = FALSE, scores = TRUE)
```

1. Arguments for `prcomp()`:
`x`: a numeric matrix or data frame
`scale`: a logical value indicating whether the variables should be scaled to have unit variance before the analysis takes place
2. Arguments for `princomp()`:
`x`: a numeric matrix or data frame `cor`: a logical value. If TRUE, the data will be centered and scaled before the analysis `scores`: a logical value. If TRUE, the coordinates on each principal component are calculated

25.3 factoextra

```
# install.packages("factoextra")
```

```
library(factoextra)
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures    rlang
#>   c.quosures    rlang
#>   print.quosures rlang
#> Welcome! Related Books: `Practical Guide To Cluster Analysis in R` at https://goo.gl/13EFCZ
```

25.4 demo dataset

We'll use the data sets `decathlon2` [in `factoextra`], which has been already described at: PCA - Data format.

Briefly, it contains:

- Active individuals (rows 1 to 23) and active variables (columns 1 to 10), which are used to perform the principal component analysis
- Supplementary individuals (rows 24 to 27) and supplementary variables (columns 11 to 13), which coordinates will be predicted using the PCA information and parameters obtained with active individuals/variables.

```
library("factoextra")
data(decathlon2)
decathlon2.active <- decathlon2[1:23, 1:10]
head(decathlon2.active[, 1:6])
#>          X100m Long.jump Shot.put High.jump X400m X110m.hurdle
#> SEBRLE    11.0    7.58    14.8    2.07  49.8     14.7
#> CLAY      10.8    7.40    14.3    1.86  49.4     14.1
#> BERNARD   11.0    7.23    14.2    1.92  48.9     15.0
#> YURKOV    11.3    7.09    15.2    2.10  50.4     15.3
#> ZSIVOCZKY 11.1    7.30    13.5    2.01  48.6     14.2
#> McMULLEN  10.8    7.31    13.8    2.13  49.9     14.4

decathlon2.supplementary <- decathlon2[24:27, 1:10]
head(decathlon2.supplementary[, 1:6])
#>          X100m Long.jump Shot.put High.jump X400m X110m.hurdle
#> KARPOV    11.0    7.30    14.8    2.04  48.4     14.1
#> WARNERS   11.1    7.60    14.3    1.98  48.7     14.2
#> Nool      10.8    7.53    14.3    1.88  48.8     14.8
#> Drews     10.9    7.38    13.1    1.88  48.5     14.0
```

25.5 Compute PCA in R using `prcomp()`

In this section we'll provide an easy-to-use R code to compute and visualize PCA in R using the `prcomp()` function and the `factoextra` package.

1. Load factoextra for visualization

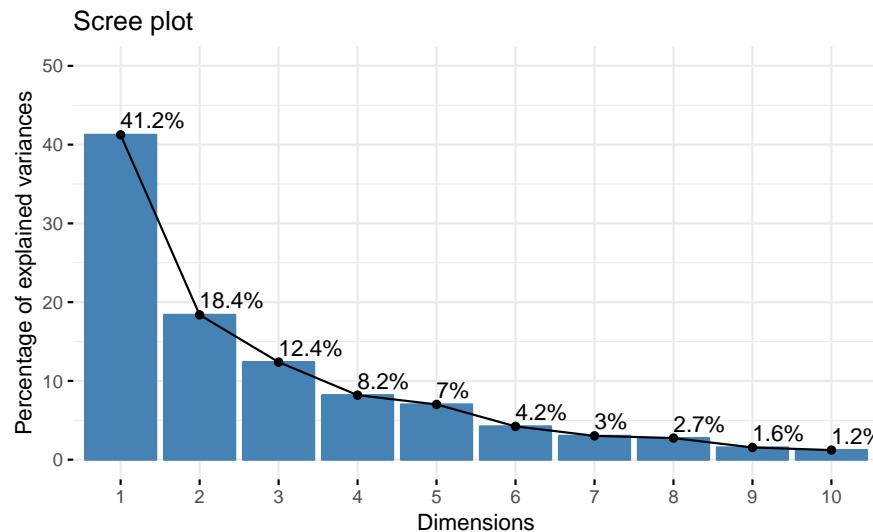
```
library(factoextra)
```

2. compute PCA

```
# compute PCA
res.pca <- prcomp(decathlon2.active, scale = TRUE)
```

3. Visualize eigenvalues (scree plot). Show the percentage of variances explained by each principal component.

```
# Visualize eigenvalues (scree plot).
fviz_eig(res.pca, addlabels = TRUE, ylim = c(0, 50))
```

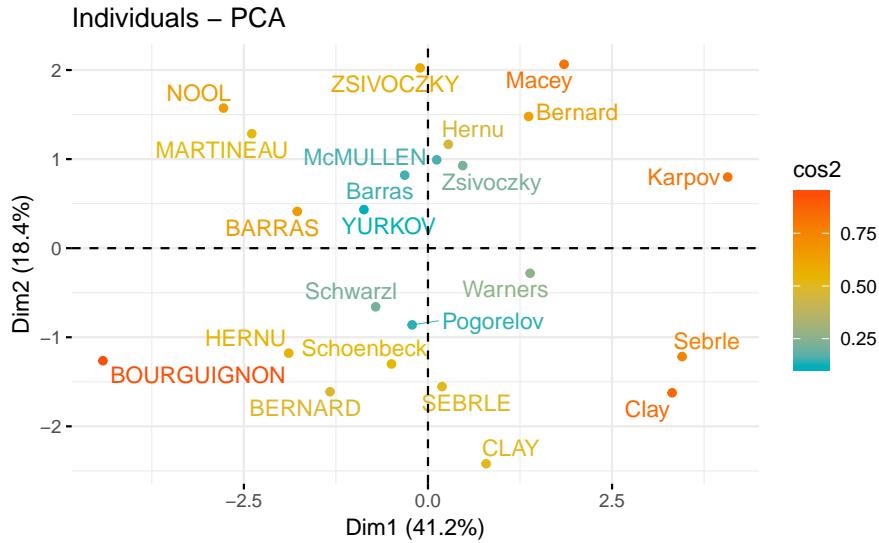


From the plot above, we might want to stop at the fifth principal component. 87% of the information (variances) contained in the data are retained by the first five principal components.

25.6 Plots: quality and contribution

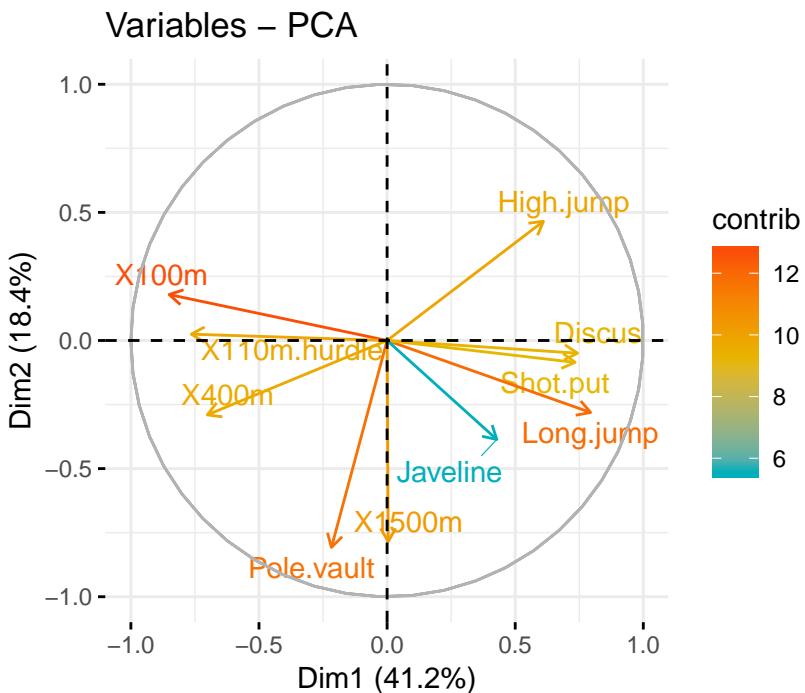
4. Graph of individuals. Individuals with a similar profile are grouped together.

```
# Graph of individuals.
fviz_pca_ind(res.pca,
             col.ind = "cos2", # Color by the quality of representation
             gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07"),
             repel = TRUE      # Avoid text overlapping
             )
```



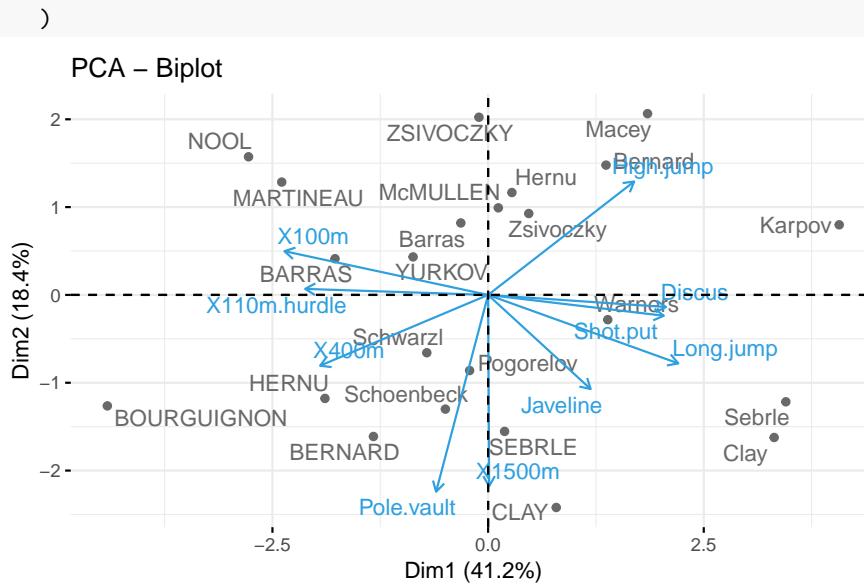
5. Graph of variables. Positive correlated variables point to the same side of the plot. Negative correlated variables point to opposite sides of the graph.

```
# Graph of variables.
fviz_pca_var(res.pca,
  col.var = "contrib", # Color by contributions to the PC
  gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07"),
  repel = TRUE      # Avoid text overlapping
)
```



6. Biplot of individuals and variables

```
# Biplot of individuals and variables
fviz_pca_biplot(res.pca, repel = TRUE,
  col.var = "#2E9FDF", # Variables color
  col.ind = "#696969" # Individuals color
```



25.7 Access to the PCA results

```

library(factoextra)
# Eigenvalues
eig.val <- get_eigenvalue(res.pca)
eig.val
#>           eigenvalue variance.percent cumulative.variance.percent
#> Dim.1        4.124          41.24                  41.2
#> Dim.2        1.839          18.39                  59.6
#> Dim.3        1.239          12.39                  72.0
#> Dim.4        0.819           8.19                  80.2
#> Dim.5        0.702           7.02                  87.2
#> Dim.6        0.423           4.23                  91.5
#> Dim.7        0.303           3.03                  94.5
#> Dim.8        0.274           2.74                  97.2
#> Dim.9        0.155           1.55                  98.8
#> Dim.10       0.122           1.22                  100.0

# Results for Variables
res.var <- get_pca_var(res.pca)
res.var$coord      # Coordinates
#>           Dim.1   Dim.2   Dim.3   Dim.4   Dim.5   Dim.6   Dim.7
#> X100m     -0.85063  0.1794 -0.3016  0.0336 -0.194  0.03537 -0.09134
#> Long.jump  0.79418 -0.2809  0.1905 -0.1154  0.233 -0.03373 -0.15433
#> Shot.put   0.73391 -0.0854 -0.5176  0.1285 -0.249 -0.23979 -0.00989
#> High.jump  0.61008  0.4652 -0.3301  0.1446  0.403 -0.28464  0.02816
#> X400m     -0.70160 -0.2902 -0.2835  0.4308  0.104 -0.04929  0.28611
#> X110m.hurdle -0.76413  0.0247 -0.4489 -0.0169  0.224  0.00263 -0.37007
#> Discus    0.74321 -0.0497 -0.1765  0.3950 -0.408  0.19854 -0.14273
#> Pole.vault -0.21727 -0.8075 -0.0941 -0.3390 -0.222 -0.32746 -0.01039
#> Javeline   0.42823 -0.3861 -0.6041 -0.3317  0.198  0.36210  0.13356
#> X1500m    0.00428 -0.7845  0.2195  0.4480  0.263  0.04205 -0.11137

```

```

#>           Dim.8   Dim.9   Dim.10
#> X100m      -0.10472 -0.3031  0.04442
#> Long.jump   -0.39738 -0.0516  0.02972
#> Shot.put     0.02436  0.0478  0.21745
#> High.jump    0.08441 -0.1121 -0.13357
#> X400m      -0.23355  0.0822 -0.03417
#> X110m.hurdle -0.00834  0.1618 -0.01563
#> Discus      -0.03956  0.0134 -0.17259
#> Pole.vault   0.03291 -0.0258 -0.13721
#> Javeline     0.05284 -0.0405 -0.00385
#> X1500m      0.19447 -0.1022  0.06283
res.var$contrib      # Contributions to the PCs
#>           Dim.1   Dim.2   Dim.3   Dim.4 Dim.5   Dim.6   Dim.7
#> X100m      1.75e+01 1.7505  7.339  0.1376  5.39  0.29592 2.7571
#> Long.jump   1.53e+01 4.2904  2.930  1.6249  7.75  0.26900 7.8716
#> Shot.put     1.31e+01 0.3967  21.620  2.0141  8.82 13.59686 0.0323
#> High.jump    9.02e+00 11.7716  8.793  2.5499 23.12 19.15961 0.2620
#> X400m      1.19e+01 4.5799  6.488  22.6509  1.54  0.57451 27.0527
#> X110m.hurdle 1.42e+01 0.0333  16.261  0.0348  7.17  0.00164 45.2616
#> Discus      1.34e+01 0.1341  2.515  19.0413 23.76  9.32175 6.7323
#> Pole.vault   1.14e+00 35.4619  0.714  14.0231  7.01 25.35762 0.0357
#> Javeline     4.45e+00 8.1087  29.453  13.4296  5.58 31.00496 5.8957
#> X1500m     4.44e-04 33.4729  3.887  24.4939  9.88  0.41813 4.0989
#>           Dim.8   Dim.9   Dim.10
#> X100m      3.9952 59.174  1.6176
#> Long.jump   57.5332 1.715  0.7241
#> Shot.put     0.2162 1.471 38.7677
#> High.jump    2.5957 8.102 14.6265
#> X400m      19.8734 4.349  0.9573
#> X110m.hurdle 0.0254 16.858  0.2003
#> Discus      0.5702 0.115 24.4217
#> Pole.vault   0.3947 0.428 15.4356
#> Javeline     1.0173 1.054  0.0122
#> X1500m     13.7787 6.734  3.2370
res.var$cos2      # Quality of representation
#>           Dim.1   Dim.2   Dim.3   Dim.4 Dim.5   Dim.6   Dim.7
#> X100m      7.24e-01 0.032184 0.09094 0.001127 0.0378 1.25e-03 8.34e-03
#> Long.jump   6.31e-01 0.078881 0.03631 0.013315 0.0544 1.14e-03 2.38e-02
#> Shot.put     5.39e-01 0.007294 0.26791 0.016504 0.0619 5.75e-02 9.77e-05
#> High.jump    3.72e-01 0.216424 0.10896 0.020895 0.1622 8.10e-02 7.93e-04
#> X400m      4.92e-01 0.084203 0.08039 0.185611 0.0108 2.43e-03 8.19e-02
#> X110m.hurdle 5.84e-01 0.000612 0.20150 0.000285 0.0503 6.93e-06 1.37e-01
#> Discus      5.52e-01 0.002466 0.03116 0.156032 0.1667 3.94e-02 2.04e-02
#> Pole.vault   4.72e-02 0.651977 0.00885 0.114911 0.0491 1.07e-01 1.08e-04
#> Javeline     1.83e-01 0.149080 0.36497 0.110048 0.0391 1.31e-01 1.78e-02
#> X1500m     1.83e-05 0.615409 0.04817 0.200713 0.0693 1.77e-03 1.24e-02
#>           Dim.8   Dim.9   Dim.10
#> X100m      1.10e-02 0.091848 1.97e-03
#> Long.jump   1.58e-01 0.002661 8.83e-04
#> Shot.put     5.93e-04 0.002284 4.73e-02
#> High.jump    7.12e-03 0.012575 1.78e-02
#> X400m      5.45e-02 0.006750 1.17e-03
#> X110m.hurdle 6.96e-05 0.026166 2.44e-04

```

```

#> Discus      1.56e-03 0.000179 2.98e-02
#> Pole.vault 1.08e-03 0.000664 1.88e-02
#> Javeline    2.79e-03 0.001637 1.49e-05
#> X1500m     3.78e-02 0.010453 3.95e-03
# Results for individuals
res.ind <- get_pca_ind(res.pca)
res.ind$coord          # Coordinates
#>                 Dim.1   Dim.2   Dim.3   Dim.4   Dim.5   Dim.6   Dim.7
#> SEBRLE        0.191 -1.554 -0.628  0.0821  1.142614 -0.4639 -0.2080
#> CLAY          0.790 -2.420  1.357  1.2698 -0.806848  1.3042 -0.2129
#> BERNARD       -1.329 -1.612 -0.196 -1.9209  0.082343 -0.4006 -0.4064
#> YURKOV        -0.869  0.433 -2.474  0.6972  0.398858  0.1029 -0.3249
#> ZSIVOCZKY    -0.106  2.023  1.305 -0.0993 -0.197024  0.8955  0.0883
#> McMULLEN      0.119  0.992  0.844  1.3122  1.585871  0.1866  0.4783
#> MARTINEAU     -2.392  1.285 -0.898  0.3731 -2.243352 -0.4567 -0.2998
#> HERNU          -1.891 -1.178 -0.156  0.8913 -0.126741  0.4362 -0.5661
#> BARRAS         -1.774  0.413  0.658  0.2287 -0.233837  0.0903  0.2159
#> NOOL           -2.777  1.573  0.607 -1.5555  1.424184  0.4972 -0.5321
#> BOURGUIGNON  -4.414 -1.264 -0.010  0.6668  0.419152 -0.0820 -0.5983
#> Sebrle         3.451 -1.217 -1.678 -0.8087 -0.025053 -0.0828  0.0102
#> Clay            3.316 -1.623 -0.618 -0.3168  0.569165  0.7772  0.2575
#> Karpov          4.070  0.798  1.015  0.3134 -0.797426 -0.3296 -1.3637
#> Macey           1.848  2.064 -0.979  0.5847 -0.000216 -0.1973 -0.2693
#> Warners         1.387 -0.282  2.000 -1.0196 -0.040540 -0.5567 -0.2674
#> Zsivoczky      0.472  0.927 -1.728 -0.1848  0.407303 -0.1138  0.0399
#> Hernu            0.276  1.166  0.171 -0.8487 -0.689480 -0.3317  0.4431
#> Bernard          1.367  1.478  0.831  0.7453  0.859802 -0.3281  0.3636
#> Schwarzl        -0.710 -0.658  1.041 -0.9272 -0.288757 -0.6889  0.5657
#> Pogorelov       -0.214 -0.861  0.298  1.3556 -0.015053 -1.5938  0.7837
#> Schoenbeck     -0.495 -1.300  0.103 -0.2493 -0.645226  0.1617  0.8575
#> Barras          -0.316  0.819 -0.862 -0.5894 -0.779739  1.1742  0.9451
#>                 Dim.8   Dim.9   Dim.10
#> SEBRLE          0.04346 -0.65934  0.0327
#> CLAY             0.61724 -0.06013 -0.3172
#> BERNARD         0.70386  0.17008 -0.0991
#> YURKOV          0.11500 -0.10952 -0.1197
#> ZSIVOCZKY      -0.20234 -0.52310 -0.3484
#> McMULLEN        0.29309 -0.10562 -0.3932
#> MARTINEAU       -0.29163 -0.22342 -0.6164
#> HERNU            1.52940  0.00618  0.5537
#> BARRAS           0.68258 -0.66928  0.5309
#> NOOL             -0.43339 -0.11578 -0.0962
#> BOURGUIGNON    0.56362  0.52581  0.0586
#> Sebrle          -0.03059 -0.84721  0.2197
#> Clay              -0.58064  0.40978 -0.6160
#> Karpov           0.34531  0.19306  0.2172
#> Macey             -0.36322  0.36826  0.2125
#> Warners          -0.10947  0.18028  0.2421
#> Zsivoczky        0.53804  0.58597 -0.1427
#> Hernu             0.24729  0.06691 -0.2087
#> Bernard           0.00617  0.27949  0.3207
#> Schwarzl         -0.68705 -0.00836 -0.3021
#> Pogorelov        -0.03762 -0.13053 -0.0370

```

```

#> Schoenbeck -0.25585 0.56422 0.2968
#> Barras      0.36555 0.10226 0.6119
res.ind$contrib      # Contributions to the PCs
#>           Dim.1   Dim.2   Dim.3   Dim.4   Dim.5   Dim.6   Dim.7
#> SEBRLE      0.0385  5.712  1.39e+00  0.0357  8.09e+00  2.2126  0.62143
#> CLAY        0.6581  13.854 6.46e+00  8.5557  4.03e+00  17.4880 0.65141
#> BERNARD     1.8627  6.144  1.35e-01  19.5783 4.20e-02  1.6502  2.37365
#> YURKOV      0.7969  0.443  2.15e+01  2.5794  9.86e-01  0.1088  1.51656
#> ZSIVOCZKY   0.0118  9.682  5.97e+00  0.0523  2.41e-01  8.2456  0.11192
#> McMULLEN    0.0148  2.325  2.50e+00  9.1353  1.56e+01  0.3579  3.28702
#> MARTINEAU   6.0337  3.904  2.83e+00  0.7386  3.12e+01  2.1441  1.29111
#> HERNU        3.7700  3.284  8.58e-02  4.2151  9.96e-02  1.9566  4.60485
#> BARRAS       3.3194  0.402  1.52e+00  0.2776  3.39e-01  0.0838  0.67004
#> NOOL         8.1299  5.849  1.29e+00  12.8376 1.26e+01  2.5413  4.06767
#> BOURGUIGNON 20.5373  3.776  3.53e-04  2.3588  1.09e+00  0.0691  5.14425
#> Sebrle       12.5584  3.502  9.88e+00  3.4701  3.89e-03  0.0705  0.00148
#> Clay          11.5936  6.232  1.34e+00  0.5325  2.01e+00  6.2097  0.95282
#> Karpov        17.4661  1.507  3.61e+00  0.5210  3.94e+00  1.1168  26.72016
#> Macey         3.6021  10.073 3.36e+00  1.8139  2.89e-07  0.4001  1.04191
#> Warners       2.0291  0.188  1.40e+01  5.5159  1.02e-02  3.1867  1.02738
#> Zsivoczky    0.2344  2.031  1.05e+01  0.1813  1.03e+00  0.1332  0.02289
#> Hernu         0.0805  3.214  1.02e-01  3.8217  2.95e+00  1.1311  2.82103
#> Bernard        1.9708  5.166  2.43e+00  2.9474  4.58e+00  1.1066  1.89945
#> Schwarzl      0.5318  1.025  3.80e+00  4.5612  5.17e-01  4.8796  4.59812
#> Pogorelov     0.0484  1.753  3.11e-01  9.7503  1.40e-03  26.1167  8.82532
#> Schoenbeck    0.2586  3.997  3.72e-02  0.3297  2.58e+00  0.2689  10.56627
#> Barras        0.1052  1.588  2.61e+00  1.8430  3.77e+00  14.1743  12.83542
#>           Dim.8   Dim.9   Dim.10
#> SEBRLE      2.99e-02 12.17748  0.0382
#> CLAY         6.04e+00  0.10126  3.5857
#> BERNARD     7.85e+00  0.81032  0.3499
#> YURKOV      2.09e-01  0.33601  0.5107
#> ZSIVOCZKY   6.49e-01  7.66492  4.3274
#> McMULLEN    1.36e+00  0.31250  5.5105
#> MARTINEAU   1.35e+00  1.39820  13.5440
#> HERNU        3.71e+01  0.00107  10.9278
#> BARRAS       7.38e+00  12.54733 10.0454
#> NOOL         2.98e+00  0.37548  0.3300
#> BOURGUIGNON 5.03e+00  7.74457  0.1222
#> Sebrle       1.48e-02 20.10555  1.7206
#> Clay          5.34e+00  4.70357 13.5271
#> Karpov        1.89e+00  1.04399  1.6819
#> Macey         2.09e+00  3.79877  1.6096
#> Warners       1.90e-01  0.91042  2.0890
#> Zsivoczky    4.59e+00  9.61785  0.7261
#> Hernu         9.69e-01  0.12540  1.5523
#> Bernard        6.02e-04  2.18807  3.6657
#> Schwarzl      7.48e+00  0.00196  3.2536
#> Pogorelov     2.24e-02  0.47727  0.0487
#> Schoenbeck   1.04e+00  8.91730  3.1402
#> Barras        2.12e+00  0.29289  13.3453
res.ind$cos2      # Quality of representation
#>           Dim.1   Dim.2   Dim.3   Dim.4   Dim.5   Dim.6   Dim.7

```

```

#> SEBRLE      0.00753 0.4975 8.13e-02 0.00139 2.69e-01 0.044324 8.91e-03
#> CLAY        0.04870 0.4570 1.44e-01 0.12579 5.08e-02 0.132691 3.54e-03
#> BERNARD     0.19720 0.2900 4.29e-03 0.41182 7.57e-04 0.017913 1.84e-02
#> YURKOV      0.09611 0.0238 7.78e-01 0.06181 2.02e-02 0.001345 1.34e-02
#> ZSIVOCZKY   0.00157 0.5764 2.40e-01 0.00139 5.47e-03 0.112918 1.10e-03
#> McMULLEN    0.00218 0.1522 1.10e-01 0.26649 3.89e-01 0.005388 3.54e-02
#> MARTINEAU   0.40401 0.1165 5.69e-02 0.00983 3.55e-01 0.014721 6.34e-03
#> HERNU        0.39928 0.1551 2.73e-03 0.08870 1.79e-03 0.021248 3.58e-02
#> BARRAS       0.61624 0.0333 8.48e-02 0.01024 1.07e-02 0.001594 9.13e-03
#> NOOL         0.48987 0.1571 2.34e-02 0.15369 1.29e-01 0.015701 1.80e-02
#> BOURGUIGNON 0.85970 0.0705 4.45e-06 0.01962 7.75e-03 0.000297 1.58e-02
#> Sebrle       0.67538 0.0840 1.60e-01 0.03708 3.56e-05 0.000389 5.85e-06
#> Clay          0.68759 0.1648 2.39e-02 0.00627 2.03e-02 0.037763 4.15e-03
#> Karpov        0.78367 0.0301 4.87e-02 0.00464 3.01e-02 0.005138 8.80e-02
#> Macey         0.36344 0.4531 1.02e-01 0.03636 4.95e-09 0.004140 7.71e-03
#> Warners       0.25565 0.0106 5.31e-01 0.13808 2.18e-04 0.041169 9.50e-03
#> Zsivoczky    0.04505 0.1740 6.05e-01 0.00692 3.36e-02 0.002625 3.23e-04
#> Hernu         0.02482 0.4418 9.46e-03 0.23420 1.55e-01 0.035771 6.38e-02
#> Bernard        0.28935 0.3381 1.07e-01 0.08598 1.14e-01 0.016659 2.05e-02
#> Schwarzl      0.11672 0.1003 2.51e-01 0.19889 1.93e-02 0.109806 7.40e-02
#> Pogorelov     0.00780 0.1259 1.50e-02 0.31210 3.85e-05 0.431416 1.04e-01
#> Schoenbeck   0.06707 0.4620 2.90e-03 0.01699 1.14e-01 0.007150 2.01e-01
#> Barras        0.01897 0.1277 1.41e-01 0.06604 1.16e-01 0.262130 1.70e-01
#>           Dim.8   Dim.9   Dim.10
#> SEBRLE        3.89e-04 8.95e-02 0.000221
#> CLAY          2.97e-02 2.82e-04 0.007847
#> BERNARD       5.53e-02 3.23e-03 0.001096
#> YURKOV        1.68e-03 1.53e-03 0.001822
#> ZSIVOCZKY    5.76e-03 3.85e-02 0.017092
#> McMULLEN     1.33e-02 1.73e-03 0.023927
#> MARTINEAU    6.00e-03 3.52e-03 0.026821
#> HERNU         2.61e-01 4.27e-06 0.034229
#> BARRAS        9.12e-02 8.77e-02 0.055153
#> NOOL          1.19e-02 8.51e-04 0.000588
#> BOURGUIGNON  1.40e-02 1.22e-02 0.000151
#> Sebrle        5.30e-05 4.07e-02 0.002737
#> Clay          2.11e-02 1.05e-02 0.023726
#> Karpov        5.64e-03 1.76e-03 0.002232
#> Macey         1.40e-02 1.44e-02 0.004803
#> Warners       1.59e-03 4.32e-03 0.007784
#> Zsivoczky    5.87e-02 6.96e-02 0.004127
#> Hernu         1.99e-02 1.46e-03 0.014160
#> Bernard        5.88e-06 1.21e-02 0.015917
#> Schwarzl      1.09e-01 1.62e-05 0.021117
#> Pogorelov     2.40e-04 2.89e-03 0.000232
#> Schoenbeck   1.79e-02 8.70e-02 0.024083
#> Barras        2.54e-02 1.99e-03 0.071184

```

25.8 Predict using PCA

In this section, we'll show how to predict the coordinates of supplementary individuals and variables using only the information provided by the previously performed PCA.

1. Data: rows 24 to 27 and columns 1 to 10 [in decathlon2 data sets]. The new data must contain columns (variables) with the same names and in the same order as the active data used to compute PCA.

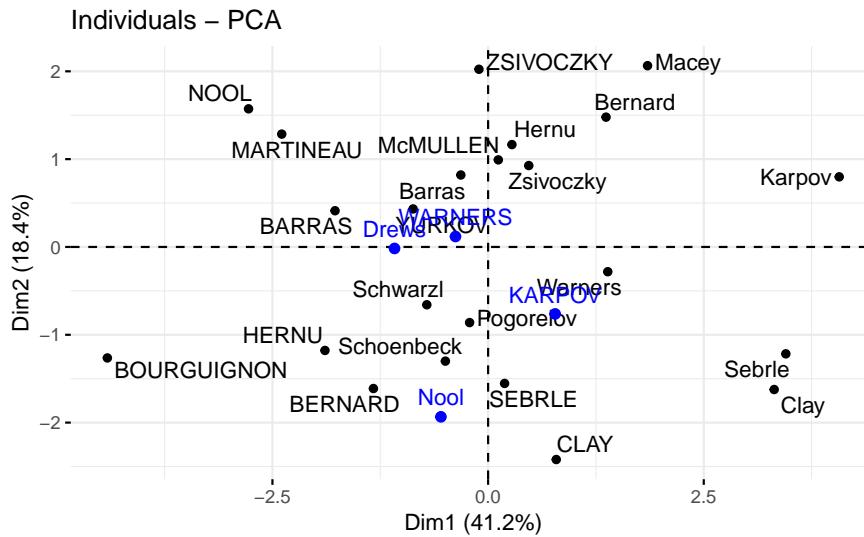
```
# Data for the supplementary individuals
ind.sup <- decathlon2[24:27, 1:10]
ind.sup[, 1:6]
#>      X100m Long.jump Shot.put High.jump X400m X110m.hurdle
#> KARPOV  11.0    7.30   14.8    2.04  48.4       14.1
#> WARNERS 11.1    7.60   14.3    1.98  48.7       14.2
#> Nool     10.8    7.53   14.3    1.88  48.8       14.8
#> Drews    10.9    7.38   13.1    1.88  48.5       14.0
```

2. Predict the coordinates of new individuals data. Use the R base function predict():

```
ind.sup.coord <- predict(res.pca, newdata = ind.sup)
ind.sup.coord[, 1:4]
#>      PC1   PC2   PC3   PC4
#> KARPOV 0.777 -0.762 1.597 1.686
#> WARNERS -0.378 0.119 1.701 -0.691
#> Nool    -0.547 -1.934 0.472 -2.228
#> Drews    -1.085 -0.017 2.982 -1.501
```

3. Graph of individuals including the supplementary individuals:

```
# Plot of active individuals
p <- fviz_pca_ind(res.pca, repel = TRUE)
# Add supplementary individuals
fviz_add(p, ind.sup.coord, color = "blue")
```



The predicted coordinates of individuals can be manually calculated as follow:

1. Center and scale the new individuals data using the center and the scale of the PCA
2. Calculate the predicted coordinates by multiplying the scaled values with the eigenvectors (loadings) of the principal components. The R code below can be used :

```
# Centering and scaling the supplementary individuals
ind.scaled <- scale(ind.sup,
                     center = res.pca$center,
                     scale = res.pca$scale)
```

```
# Coordinates of the individuals
coord_func <- function(ind, loadings){
  r <- loadings*ind
  apply(r, 2, sum)
}

pca.loadings <- res.pca$rotation
ind.sup.coord <- t(apply(ind.scaled, 1, coord_func, pca.loadings ))
ind.sup.coord[, 1:4]
#>      PC1    PC2    PC3    PC4
#> KARPOV  0.777 -0.762 1.597  1.686
#> WARNERS -0.378  0.119 1.701 -0.691
#> Nool    -0.547 -1.934 0.472 -2.228
#> Drews   -1.085 -0.017 2.982 -1.501
```

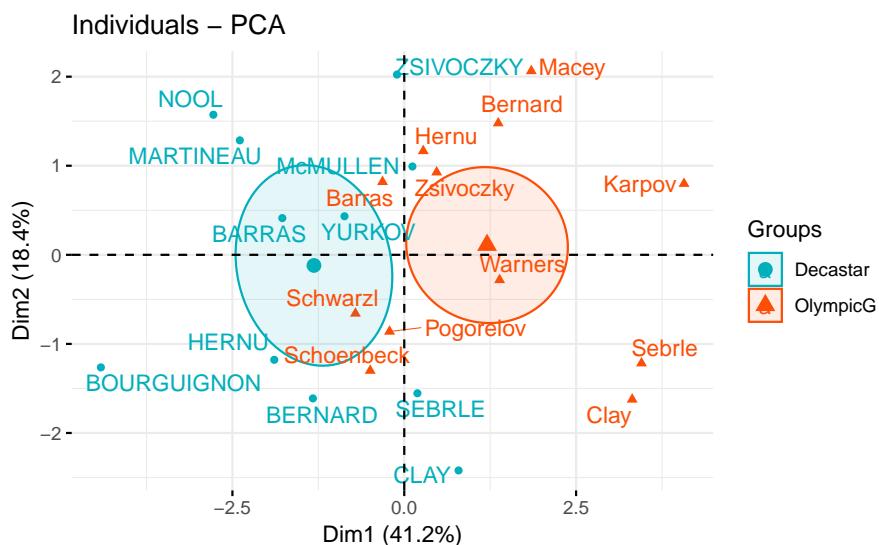
25.9 Supplementary variables

25.9.1 Qualitative / categorical variables

The data sets decathlon2 contain a supplementary qualitative variable at columns 13 corresponding to the type of competitions.

Qualitative / categorical variables can be used to color individuals by groups. The grouping variable should be of same length as the number of active individuals (here 23).

```
groups <- as.factor(decathlon2$Competition[1:23])
fviz_pca_ind(res.pca,
             col.ind = groups, # color by groups
             palette = c("#00AFBB", "#FC4E07"),
             addEllipses = TRUE, # Concentration ellipses
             ellipse.type = "confidence",
             legend.title = "Groups",
             repel = TRUE
)
```



Calculate the coordinates for the levels of grouping variables. The coordinates for a given group is calculated as the mean coordinates of the individuals in the group.

```

library(magrittr) # for pipe %>%
library(dplyr)   # everything else
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union

# 1. Individual coordinates
res.ind <- get_pca_ind(res.pca)
# 2. Coordinate of groups
coord.groups <- res.ind$coord %>%
  as_data_frame() %>%
  select(Dim.1, Dim.2) %>%
  mutate(competition = groups) %>%
  group_by(competition) %>%
  summarise(
    Dim.1 = mean(Dim.1),
    Dim.2 = mean(Dim.2)
  )
#> Warning: `as_data_frame()` is deprecated, use `as_tibble()` (but mind the new semantics).
#> This warning is displayed once per session.
coord.groups
#> # A tibble: 2 x 3
#>   competition Dim.1  Dim.2
#>   <fct>        <dbl>  <dbl>
#> 1 Decastar     -1.31 -0.119
#> 2 OlympicG     1.20  0.109

```

25.9.2 Quantitative variables

Data: columns 11:12. Should be of same length as the number of active individuals (here 23)

```

quanti.sup <- decathlon2[1:23, 11:12, drop = FALSE]
head(quanti.sup)
#>      Rank Points
#> SEBRLE     1  8217
#> CLAY       2  8122
#> BERNARD    4  8067
#> YURKOV     5  8036
#> ZSIVOCZKY  7  8004
#> McMULLEN   8  7995

```

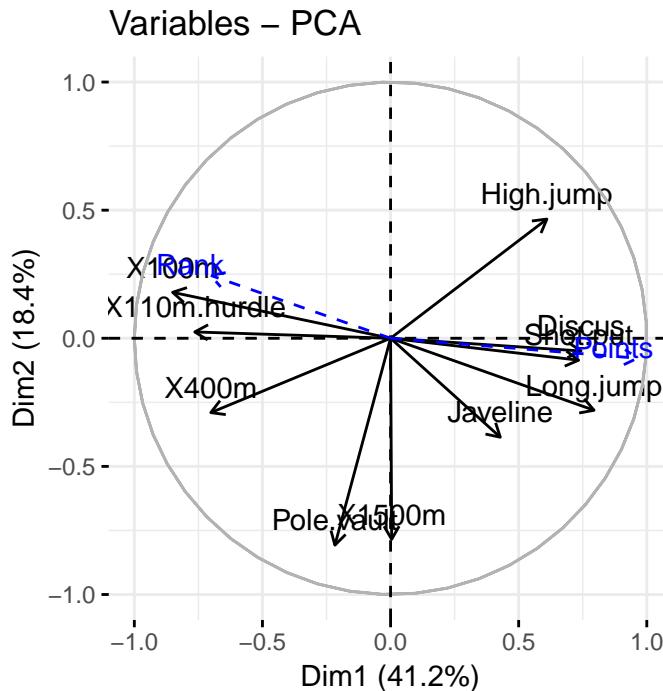
The coordinates of a given quantitative variable are calculated as the correlation between the quantitative variables and the principal components.

```

# Predict coordinates and compute cos2
quanti.coord <- cor(quanti.sup, res.pca$x)
quanti.cos2 <- quanti.coord^2
# Graph of variables including supplementary variables
p <- fviz_pca_var(res.pca)

```

```
fviz_add(p, quanti.coord, color ="blue", geom="arrow")
```



25.10 Theory behind PCA results

25.10.1 PCA results for variables

Here we'll show how to calculate the PCA results for variables: coordinates, cos2 and contributions:

`var.coord = loadings * the component standard deviations` `var.cos2 = var.coord^2` `var.contrib.` The contribution of a variable to a given principal component is (in percentage) : $(\text{var.cos2} * 100) / (\text{total cos2 of the component})$

```
# Helper function
#####
var_coord_func <- function(loadings, comp.sdev){
  loadings*comp.sdev
}

# Compute Coordinates
#####
loadings <- res.pca$rotation
sdev <- res.pca$sdev
var.coord <- t(apply(loadings, 1, var_coord_func, sdev))
head(var.coord[, 1:4])
#>          PC1      PC2      PC3      PC4
#> X100m     -0.851  0.1794 -0.302  0.0336
#> Long.jump    0.794 -0.2809  0.191 -0.1154
#> Shot.put     0.734 -0.0854 -0.518  0.1285
#> High.jump     0.610  0.4652 -0.330  0.1446
#> X400m      -0.702 -0.2902 -0.284  0.4308
#> X110m.hurdle -0.764  0.0247 -0.449 -0.0169
```

```
# Compute Cos2
#####
var.cos2 <- var.coord^2
head(var.cos2[, 1:4])
#>          PC1      PC2      PC3      PC4
#> X100m    0.724  0.032184  0.0909  0.001127
#> Long.jump 0.631  0.078881  0.0363  0.013315
#> Shot.put   0.539  0.007294  0.2679  0.016504
#> High.jump   0.372  0.216424  0.1090  0.020895
#> X400m     0.492  0.084203  0.0804  0.185611
#> X110m.hurdle 0.584  0.000612  0.2015  0.000285

# Compute contributions
#####
comp.cos2 <- apply(var.cos2, 2, sum)
contrib <- function(var.cos2, comp.cos2){var.cos2*100/comp.cos2}
var.contrib <- t(apply(var.cos2, 1, contrib, comp.cos2))
head(var.contrib[, 1:4])
#>          PC1      PC2      PC3      PC4
#> X100m    17.54  1.7505  7.34  0.1376
#> Long.jump 15.29  4.2904  2.93  1.6249
#> Shot.put   13.06  0.3967 21.62  2.0141
#> High.jump   9.02  11.7716  8.79  2.5499
#> X400m     11.94  4.5799  6.49 22.6509
#> X110m.hurdle 14.16  0.0333 16.26  0.0348
```

25.10.2 PCA results for individuals

- `ind.coord = res.pca$x`
- Cos2 of individuals. Two steps:
 - Calculate the square distance between each individual and the PCA center of gravity: $d2 = [(var1_ind_i - mean_var1)/sd_var1]^2 + \dots + [(var10_ind_i - mean_var10)/sd_var10]^2 + \dots + ..$
 - Calculate the cos2 as $ind.coord^2/d2$
- Contributions of individuals to the principal components: $100 * (1 / number_of_individuals) * (ind.coord^2 / comp_sdev^2)$. Note that the sum of all the contributions per column is 100

```
# Coordinates of individuals
#####
ind.coord <- res.pca$x
head(ind.coord[, 1:4])
#>          PC1      PC2      PC3      PC4
#> SEBRLE    0.191 -1.554 -0.628  0.0821
#> CLAY       0.790 -2.420  1.357  1.2698
#> BERNARD   -1.329 -1.612 -0.196 -1.9209
#> YURKOV    -0.869  0.433 -2.474  0.6972
#> ZSIVOCZKY -0.106  2.023  1.305 -0.0993
#> McMULLEN   0.119  0.992  0.844  1.3122

# Cos2 of individuals
#####
# 1. square of the distance between an individual and the
# PCA center of gravity
```

```

center <- res.pca$center
scale<- res.pca$scale

getdistance <- function(ind_row, center, scale){
  return(sum((ind_row-center)/scale)^2))
}

d2 <- apply(decathlon2.active,1, getdistance, center, scale)
# 2. Compute the cos2. The sum of each row is 1
cos2 <- function(ind.coord, d2){return(ind.coord^2/d2)}
ind.cos2 <- apply(ind.coord, 2, cos2, d2)
head(ind.cos2[, 1:4])
#>          PC1     PC2     PC3     PC4
#> SEBRLE   0.00753 0.4975 0.08133 0.00139
#> CLAY     0.04870 0.4570 0.14363 0.12579
#> BERNARD  0.19720 0.2900 0.00429 0.41182
#> YURKOV   0.09611 0.0238 0.77823 0.06181
#> ZSIVOCZKY 0.00157 0.5764 0.23975 0.00139
#> McMULLEN 0.00218 0.1522 0.11014 0.26649

# Contributions of individuals
#####
contrib <- function(ind.coord, comp.sdev, n.ind){
  100*(1/n.ind)*ind.coord^2/comp.sdev^2
}
ind.contrib <- t(apply(ind.coord, 1, contrib,
                        res.pca$sdev, nrow(ind.coord)))
head(ind.contrib[, 1:4])
#>          PC1     PC2     PC3     PC4
#> SEBRLE   0.0385  5.712   1.385   0.0357
#> CLAY     0.6581 13.854   6.460   8.5557
#> BERNARD  1.8627  6.144   0.135 19.5783
#> YURKOV   0.7969  0.443 21.476   2.5794
#> ZSIVOCZKY 0.0118  9.682   5.975   0.0523
#> McMULLEN 0.0148  2.325   2.497  9.1353

```


Chapter 26

Principal Components Methods

<http://www.sthda.com/english/articles/31-principal-component-methods-in-r-practical-guide/112-pca-principal-components-methods>

Principal component analysis (PCA) allows us to summarize and to visualize the information in a data set containing individuals/observations described by multiple inter-correlated quantitative variables. Each variable could be considered as a different dimension. If you have more than 3 variables in your data sets, it could be very difficult to visualize a multi-dimensional hyperspace.

Principal component analysis is used to extract the important information from a multivariate data table and to express this information as a set of few new variables called principal components. These new variables correspond to a linear combination of the originals. The number of principal components is less than or equal to the number of original variables.

The information in a given data set corresponds to the total variation it contains. The goal of PCA is to identify directions (or principal components) along which the variation in the data is maximal.

In other words, PCA reduces the dimensionality of a multivariate data to two or three principal components, that can be visualized graphically, with minimal loss of information.

```
# install.packages(c("FactoMineR", "factoextra"))

library(FactoMineR)
library(factoextra)
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures    rlang
#>   c.quosures    rlang
#>   print.quosures rlang
#> Welcome! Related Books: `Practical Guide To Cluster Analysis in R` at https://goo.gl/13EFCZ

data(decathlon2)
# head(decathlon2)
```

In PCA terminology, our data contains :

- Active individuals (in light blue, rows 1:23) : Individuals that are used during the principal component analysis.
- Supplementary individuals (in dark blue, rows 24:27) : The coordinates of these individuals will be predicted using the PCA information and parameters obtained with active individuals/variables
- Active variables (in pink, columns 1:10) : Variables that are used for the principal component analysis.

- Supplementary variables: As supplementary individuals, the coordinates of these variables will be predicted also. These can be:
 - Supplementary continuous variables (red): Columns 11 and 12 corresponding respectively to the rank and the points of athletes.
 - Supplementary qualitative variables (green): Column 13 corresponding to the two athlete-tic meetings (2004 Olympic Game or 2004 Decastar). This is a categorical (or factor) variable factor. It can be used to color individuals by groups.

We start by subsetting active individuals and active variables for the principal component analysis:

```
decathlon2.active <- decathlon2[1:23, 1:10]
head(decathlon2.active[, 1:6], 4)
#>           X100m Long.jump Shot.put High.jump X400m X110m.hurdle
#> SEBRLE    11.0     7.58   14.8     2.07  49.8      14.7
#> CLAY      10.8     7.40   14.3     1.86  49.4      14.1
#> BERNARD   11.0     7.23   14.2     1.92  48.9      15.0
#> YURKOV   11.3     7.09   15.2     2.10  50.4      15.3
```

26.1 Data standardization

In principal component analysis, variables are often scaled (i.e. standardized). This is particularly recommended when variables are measured in different scales (e.g: kilograms, kilometers, centimeters, ...); otherwise, the PCA outputs obtained will be severely affected.

The goal is to make the variables comparable. Generally variables are scaled to have i) standard deviation one and ii) mean zero.

The function PCA() [FactoMineR package] can be used. A simplified format is:

```
library(FactoMineR)
res.pca <- PCA(decathlon2.active, graph = FALSE)

print(res.pca)
#> **Results for the Principal Component Analysis (PCA)**
#> The analysis was performed on 23 individuals, described by 10 variables
#> *The results are available in the following objects:
#>
#>   name           description
#> 1  "$eig"         "eigenvalues"
#> 2  "$var"          "results for the variables"
#> 3  "$var$coord"   "coord. for the variables"
#> 4  "$var$cor"      "correlations variables - dimensions"
#> 5  "$var$cos2"     "cos2 for the variables"
#> 6  "$var$contrib" "contributions of the variables"
#> 7  "$ind"          "results for the individuals"
#> 8  "$ind$coord"   "coord. for the individuals"
#> 9  "$ind$cos2"    "cos2 for the individuals"
#> 10 "$ind$contrib" "contributions of the individuals"
#> 11 "$call"         "summary statistics"
#> 12 "$call$centre" "mean of the variables"
#> 13 "$call$ecart.type" "standard error of the variables"
#> 14 "$call$row.w"  "weights for the individuals"
#> 15 "$call$col.w"  "weights for the variables"
```

The object that is created using the function `PCA()` contains many information found in many different lists and matrices. These values are described in the next section.

26.2 Eigenvalues / Variances

As described in previous sections, the eigenvalues measure the amount of variation retained by each principal component. Eigenvalues are large for the first PCs and small for the subsequent PCs. That is, the first PCs corresponds to the directions with the maximum amount of variation in the data set.

We examine the eigenvalues to determine the number of principal components to be considered. The eigenvalues and the proportion of variances (i.e., information) retained by the principal components (PCs) can be extracted using the function `get_eigenvalue()` [factoextra package].

```
library(factoextra)
eig.val <- get_eigenvalue(res.pca)
eig.val
#>   eigenvalue variance.percent cumulative.variance.percent
#> Dim.1      4.124          41.24              41.2
#> Dim.2      1.839          18.39              59.6
#> Dim.3      1.239          12.39              72.0
#> Dim.4      0.819           8.19              80.2
#> Dim.5      0.702           7.02              87.2
#> Dim.6      0.423           4.23              91.5
#> Dim.7      0.303           3.03              94.5
#> Dim.8      0.274           2.74              97.2
#> Dim.9      0.155           1.55              98.8
#> Dim.10     0.122           1.22             100.0
```

The sum of all the eigenvalues give a total variance of 10.

The proportion of variation explained by each eigenvalue is given in the second column. For example, 4.124 divided by 10 equals 0.4124, or, about 41.24% of the variation is explained by this first eigenvalue. The cumulative percentage explained is obtained by adding the successive proportions of variation explained to obtain the running total. For instance, 41.242% plus 18.385% equals 59.627%, and so forth. Therefore, about 59.627% of the variation is explained by the first two eigenvalues together.

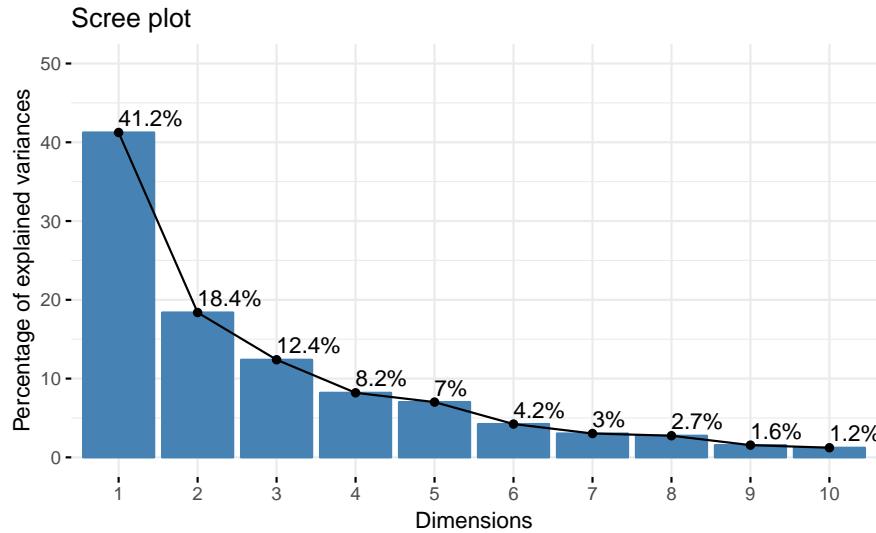
Unfortunately, there is no well-accepted objective way to decide how many principal components are enough. This will depend on the specific field of application and the specific data set. In practice, we tend to look at the first few principal components in order to find interesting patterns in the data.

In our analysis, the first three principal components explain 72% of the variation. This is an acceptably large percentage.

An alternative method to determine the number of principal components is to look at a Scree Plot, which is the plot of eigenvalues ordered from largest to the smallest. The number of component is determined at the point, beyond which the remaining eigenvalues are all relatively small and of comparable size (Jolliffe 2002, Peres-Neto, Jackson, and Somers (2005)).

The scree plot can be produced using the function `fviz_eig()` or `fviz_screeplot()` [factoextra package].

```
fviz_eig(res.pca, addlabels = TRUE, ylim = c(0, 50))
```



From the plot above, we might want to stop at the fifth principal component. 87% of the information (variances) contained in the data are retained by the first five principal components.

26.3 Graph of variables

Results A simple method to extract the results, for variables, from a PCA output is to use the function `get_pca_var()` [factoextra package]. This function provides a list of matrices containing all the results for the active variables (coordinates, correlation between variables and axes, squared cosine and contributions)

```
var <- get_pca_var(res.pca)
var
#> Principal Component Analysis Results for variables
#> -----
#>   Name      Description
#> 1 "$coord" "Coordinates for the variables"
#> 2 "$cor"    "Correlations between variables and dimensions"
#> 3 "$cos2"   "Cos2 for the variables"
#> 4 "$contrib" "contributions of the variables"
```

The components of the `get_pca_var()` can be used in the plot of variables as follow:

- `var$coord`: coordinates of variables to create a scatter plot
- `var$cos2`: represents the quality of representation for variables on the factor map. It's calculated as the squared coordinates: `var.cos2 = var.coord * var.coord`.
- `var$contrib`: contains the contributions (in percentage) of the variables to the principal components. The contribution of a variable (`var`) to a given principal component is (in percentage) : $(var.cos2 * 100) / (\text{total cos2 of the component})$.

Note that, it's possible to plot variables and to color them according to either i) their quality on the factor map (`cos2`) or ii) their contribution values to the principal components (`contrib`).

The different components can be accessed as follow:

```
# Coordinates
head(var$coord)
#>           Dim.1   Dim.2   Dim.3   Dim.4   Dim.5
#> X100m     -0.851  -0.1794  0.302   0.0336  -0.194
#> Long.jump  0.794   0.2809  -0.191  -0.1154   0.233
```

```
#> Shot.put      0.734  0.0854  0.518  0.1285 -0.249
#> High.jump    0.610 -0.4652  0.330  0.1446  0.403
#> X400m        -0.702  0.2902  0.284  0.4308  0.104
#> X110m.hurdle -0.764 -0.0247  0.449 -0.0169  0.224
# Cos2: quality on the factor map
head(var$cos2)
#>                  Dim.1   Dim.2   Dim.3   Dim.4   Dim.5
#> X100m        0.724  0.032184 0.0909  0.001127 0.0378
#> Long.jump    0.631  0.078881 0.0363  0.013315 0.0544
#> Shot.put     0.539  0.007294 0.2679  0.016504 0.0619
#> High.jump    0.372  0.216424 0.1090  0.020895 0.1622
#> X400m        0.492  0.084203 0.0804  0.185611 0.0108
#> X110m.hurdle 0.584  0.000612 0.2015  0.000285 0.0503
# Contributions to the principal components
head(var$contrib)
#>                  Dim.1   Dim.2   Dim.3   Dim.4   Dim.5
#> X100m        17.54   1.7505  7.34   0.1376  5.39
#> Long.jump    15.29   4.2904  2.93   1.6249  7.75
#> Shot.put     13.06   0.3967 21.62   2.0141  8.82
#> High.jump    9.02   11.7716  8.79   2.5499 23.12
#> X400m        11.94   4.5799  6.49  22.6509  1.54
#> X110m.hurdle 14.16   0.0333 16.26   0.0348  7.17
```

In this section, we describe how to visualize variables and draw conclusions about their correlations. Next, we highlight variables according to either i) their quality of representation on the factor map or ii) their contributions to the principal components.

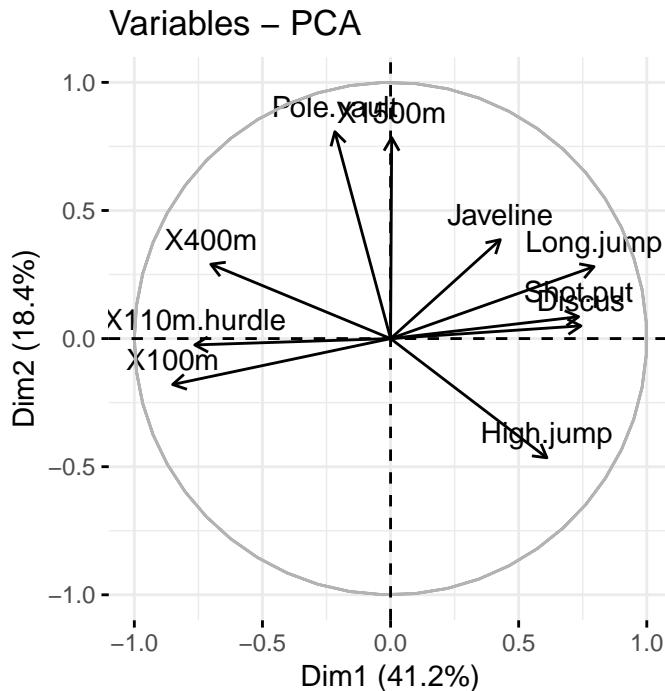
26.4 Correlation circle

The correlation between a variable and a principal component (PC) is used as the coordinates of the variable on the PC. The representation of variables differs from the plot of the observations: The observations are represented by their projections, but the variables are represented by their correlations (Abdi and Williams 2010).

```
# Coordinates of variables
head(var$coord, 4)
#>                  Dim.1   Dim.2   Dim.3   Dim.4   Dim.5
#> X100m       -0.851 -0.1794  0.302  0.0336 -0.194
#> Long.jump    0.794  0.2809 -0.191 -0.1154  0.233
#> Shot.put     0.734  0.0854  0.518  0.1285 -0.249
#> High.jump    0.610 -0.4652  0.330  0.1446  0.403
```

To plot variables, type this:

```
fviz_pca_var(res.pca, col.var = "black")
```



The plot above is also known as variable correlation plots. It shows the relationships between all variables. It can be interpreted as follow:

- Positively correlated variables are grouped together.
- Negatively correlated variables are positioned on opposite sides of the plot origin (opposed quadrants).
- The distance between variables and the origin measures the quality of the variables on the factor map. Variables that are away from the origin are well represented on the factor map.

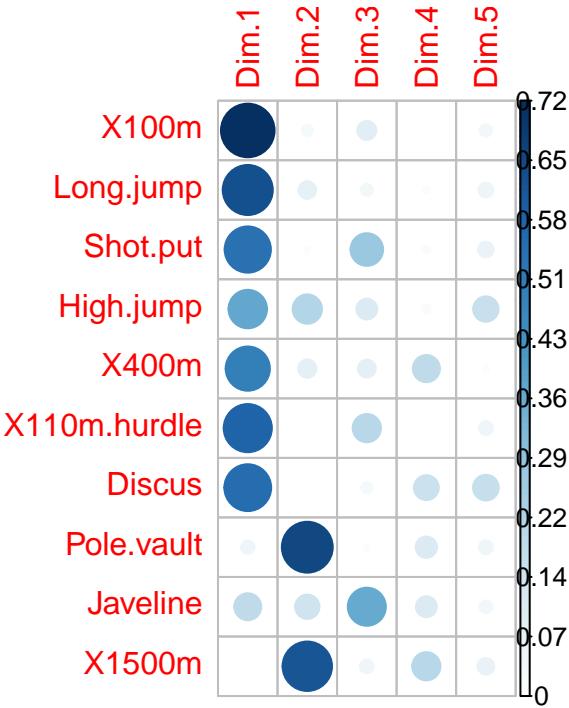
26.5 Quality of representation

The quality of representation of the variables on factor map is called cos2 (square cosine, squared coordinates). You can access to the cos2 as follow:

```
head(var$cos2, 4)
#>           Dim.1   Dim.2   Dim.3   Dim.4   Dim.5
#> X100m      0.724  0.03218  0.0909  0.00113  0.0378
#> Long.jump  0.631  0.07888  0.0363  0.01331  0.0544
#> Shot.put   0.539  0.00729  0.2679  0.01650  0.0619
#> High.jump  0.372  0.21642  0.1090  0.02089  0.1622
```

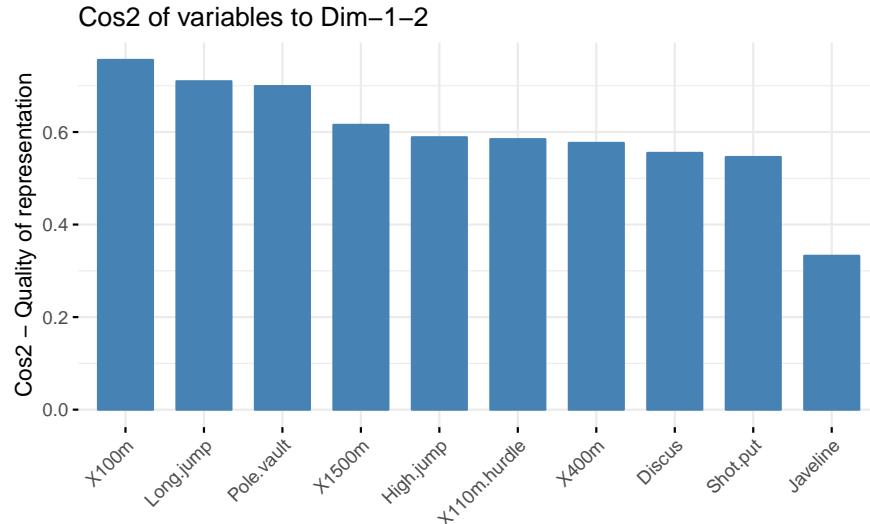
You can visualize the cos2 of variables on all the dimensions using the corrplot package:

```
library(corrplot)
#> corrplot 0.84 loaded
corrplot(var$cos2, is.corr=FALSE)
```



It's also possible to create a bar plot of variables cos2 using the function `fviz_cos2()` [in factoextra]:

```
# Total cos2 of variables on Dim.1 and Dim.2
fviz_cos2(res.pca, choice = "var", axes = 1:2)
```



Note that,

- A high cos2 indicates a good representation of the variable on the principal component. In this case the variable is positioned close to the circumference of the correlation circle.
- A low cos2 indicates that the variable is not perfectly represented by the PCs. In this case the variable is close to the center of the circle.

For a given variable, the sum of the cos2 on all the principal components is equal to one.

If a variable is perfectly represented by only two principal components (Dim.1 & Dim.2), the sum of the cos2 on these two PCs is equal to one. In this case the variables will be positioned on the circle of correlations.

For some of the variables, more than 2 components might be required to perfectly represent the data. In this case the variables are positioned inside the circle of correlations.

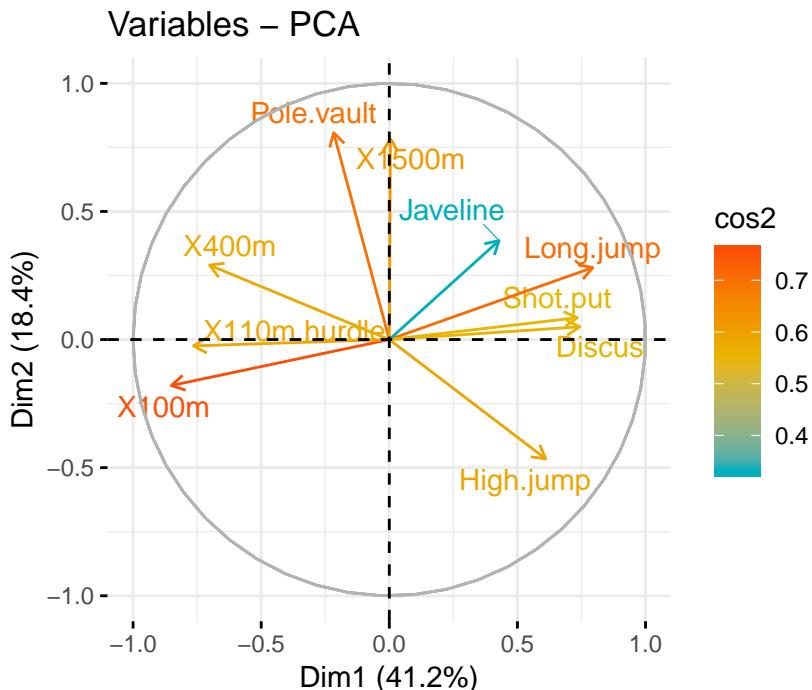
In summary:

- The cos2 values are used to estimate the quality of the representation
- The closer a variable is to the circle of correlations, the better its representation on the factor map (and the more important it is to interpret these components)
- Variables that are closed to the center of the plot are less important for the first components.

It's possible to color variables by their cos2 values using the argument col.var = "cos2". This produces a gradient colors. In this case, the argument gradient.cols can be used to provide a custom color. For instance, gradient.cols = c("white", "blue", "red") means that:

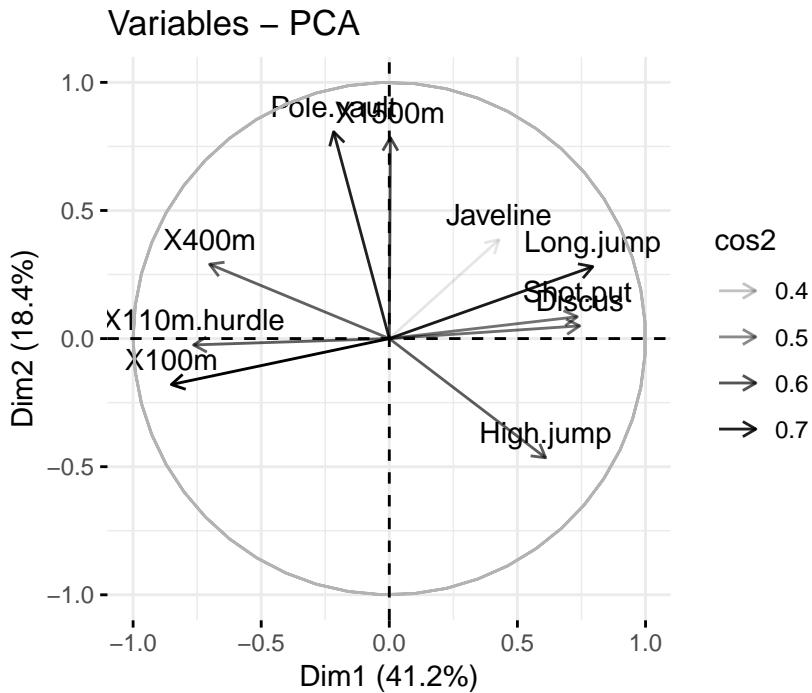
- variables with low cos2 values will be colored in "white"
- variables with mid cos2 values will be colored in "blue"
- variables with high cos2 values will be colored in red

```
# Color by cos2 values: quality on the factor map
fviz_pca_var(res.pca, col.var = "cos2",
             gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07"),
             repel = TRUE # Avoid text overlapping
           )
```



Note that, it's also possible to change the transparency of the variables according to their cos2 values using the option alpha.var = "cos2". For example, type this:

```
# Change the transparency by cos2 values
fviz_pca_var(res.pca, alpha.var = "cos2")
```



26.6 Contributions of variables to PCs

The contributions of variables in accounting for the variability in a given principal component are expressed in percentage.

- Variables that are correlated with PC1 (i.e., Dim.1) and PC2 (i.e., Dim.2) are the most important in explaining the variability in the data set.
- Variables that do not correlate with any PC or correlated with the last dimensions are variables with low contribution and might be removed to simplify the overall analysis.

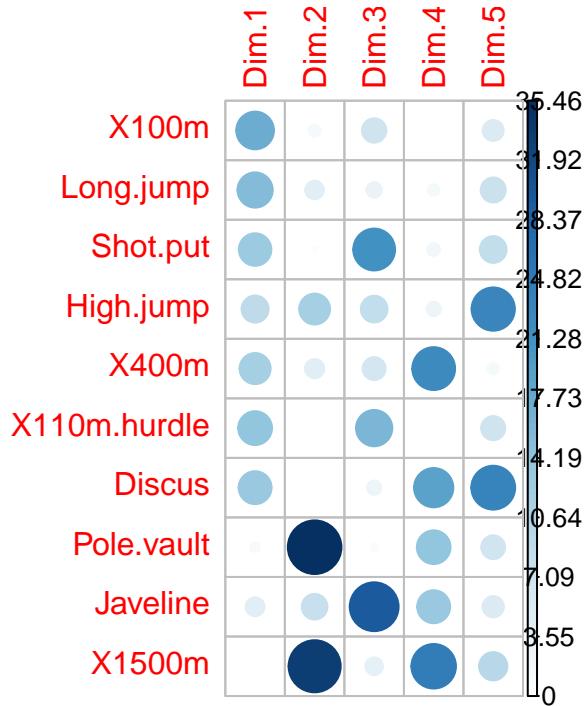
The contribution of variables can be extracted as follow :

```
head(var$contrib, 4)
#>           Dim.1 Dim.2 Dim.3 Dim.4 Dim.5
#> X100m     17.54  1.751  7.34  0.138  5.39
#> Long.jump 15.29  4.290  2.93  1.625  7.75
#> Shot.put   13.06  0.397 21.62  2.014  8.82
#> High.jump  9.02 11.772  8.79  2.550 23.12
```

The larger the value of the contribution, the more the variable contributes to the component.

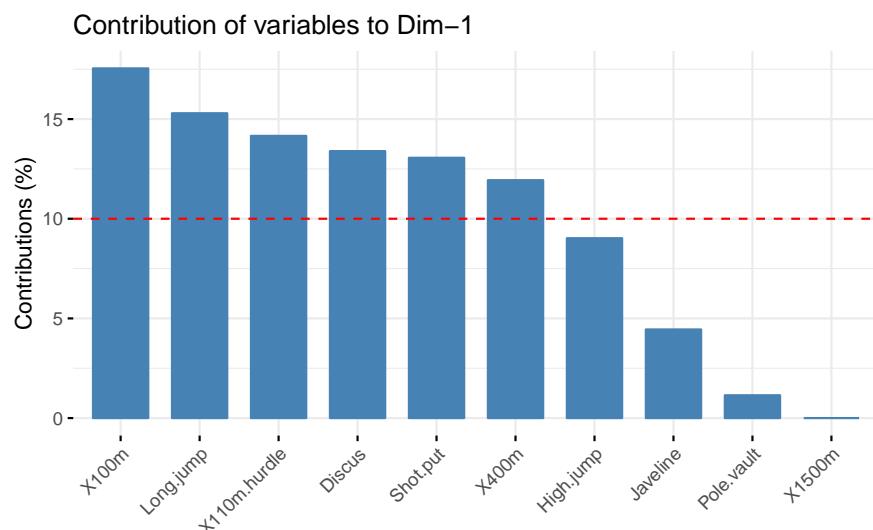
It's possible to use the function corrplot() [corrplot package] to highlight the most contributing variables for each dimension:

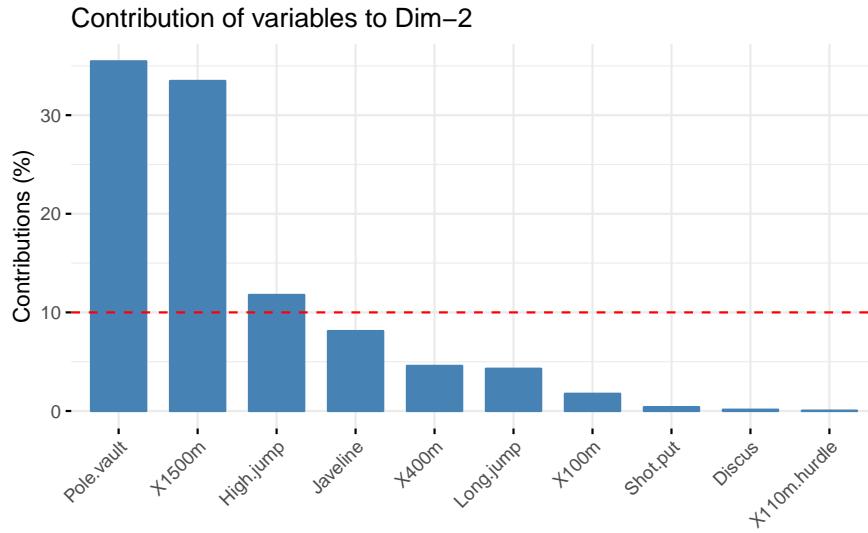
```
library("corrplot")
corrplot(var$contrib, is.corr=FALSE)
```



The function `fviz_contrib()` [factoextra package] can be used to draw a bar plot of variable contributions. If your data contains many variables, you can decide to show only the top contributing variables. The R code below shows the top 10 variables contributing to the principal components:

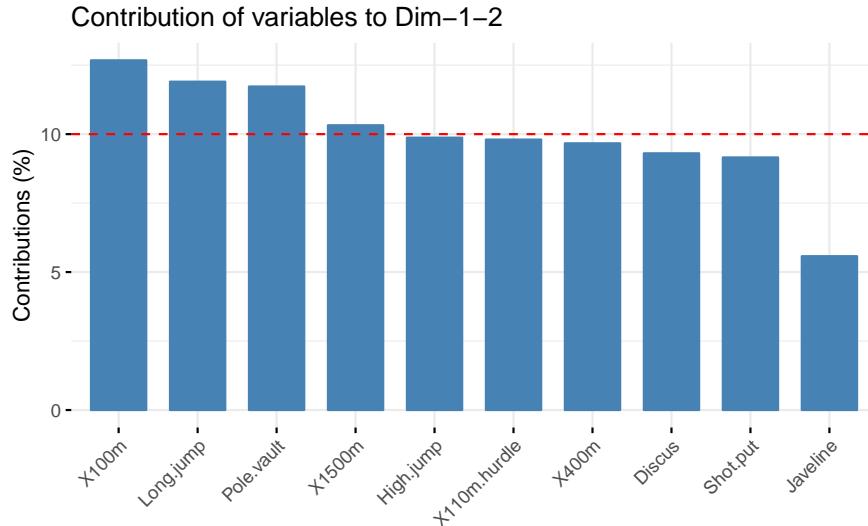
```
# Contributions of variables to PC1
fviz_contrib(res.pca, choice = "var", axes = 1, top = 10)
# Contributions of variables to PC2
fviz_contrib(res.pca, choice = "var", axes = 2, top = 10)
```





The total contribution to PC1 and PC2 is obtained with the following R code:

```
fviz_contrib(res.pca, choice = "var", axes = 1:2, top = 10)
```



The red dashed line on the graph above indicates the expected average contribution. If the contribution of the variables were uniform, the expected value would be $1/\text{length}(\text{variables}) = 1/10 = 10\%$. For a given component, a variable with a contribution larger than this cutoff could be considered as important in contributing to the component.

Note that, the total contribution of a given variable, on explaining the variations retained by two principal components, say PC1 and PC2, is calculated as $\text{contrib} = [(C1 * \text{Eig1}) + (C2 * \text{Eig2})]/(\text{Eig1} + \text{Eig2})$, where

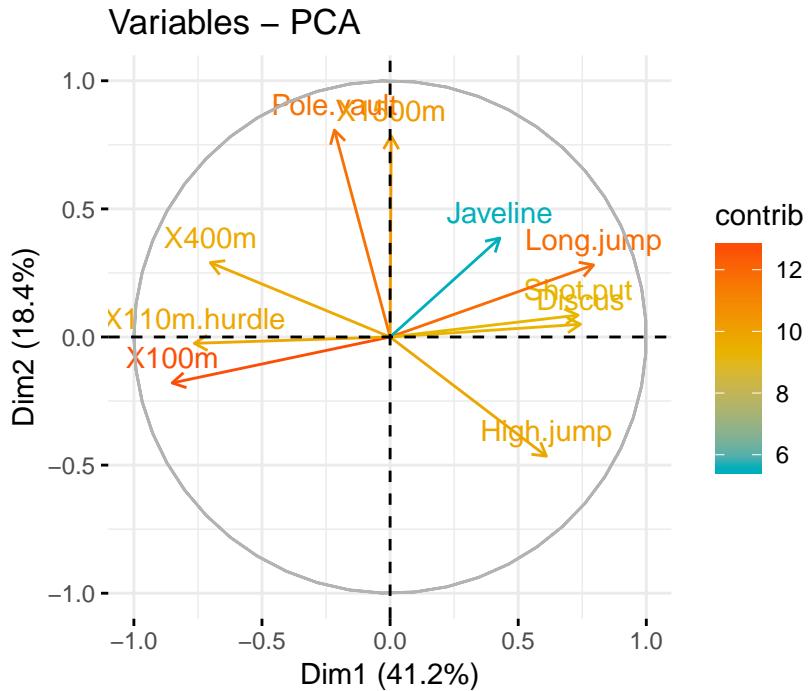
- C1 and C2 are the contributions of the variable on PC1 and PC2, respectively
- Eig1 and Eig2 are the eigenvalues of PC1 and PC2, respectively. Recall that eigenvalues measure the amount of variation retained by each PC.

In this case, the expected average contribution (cutoff) is calculated as follow: As mentioned above, if the contributions of the 10 variables were uniform, the expected average contribution on a given PC would be $1/10 = 10\%$. The expected average contribution of a variable for PC1 and PC2 is : $[(10 * \text{Eig1}) + (10 * \text{Eig2})]/(\text{Eig1} + \text{Eig2})$

It can be seen that the variables - X100m, Long.jump and Pole.vault - contribute the most to the dimensions 1 and 2.

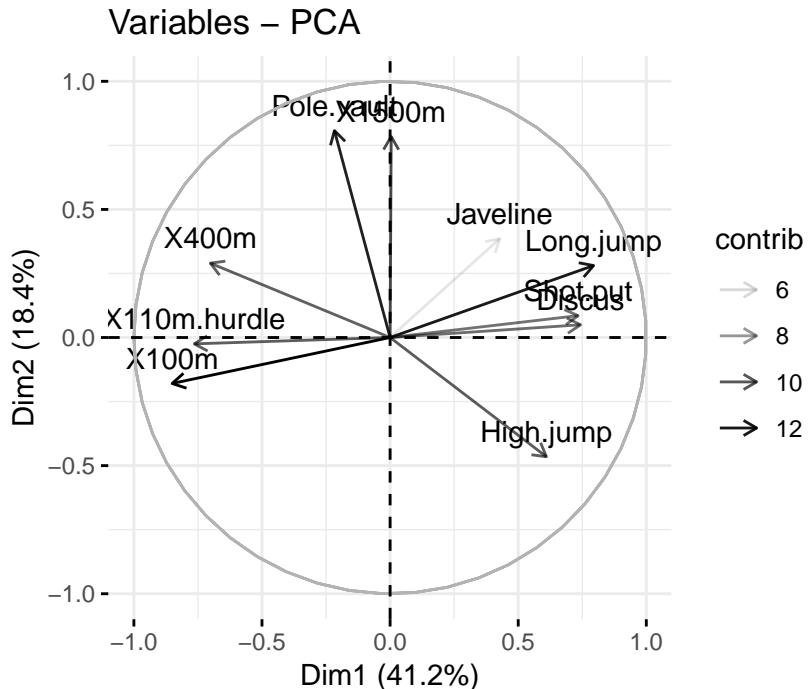
The most important (or, contributing) variables can be highlighted on the correlation plot as follow:

```
fviz_pca_var(res.pca, col.var = "contrib",
             gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07")
           )
```



Note that, it's also possible to change the transparency of variables according to their contrib values using the option alpha.var = "contrib". For example, type this:

```
# Change the transparency by contrib values
fviz_pca_var(res.pca, alpha.var = "contrib")
```

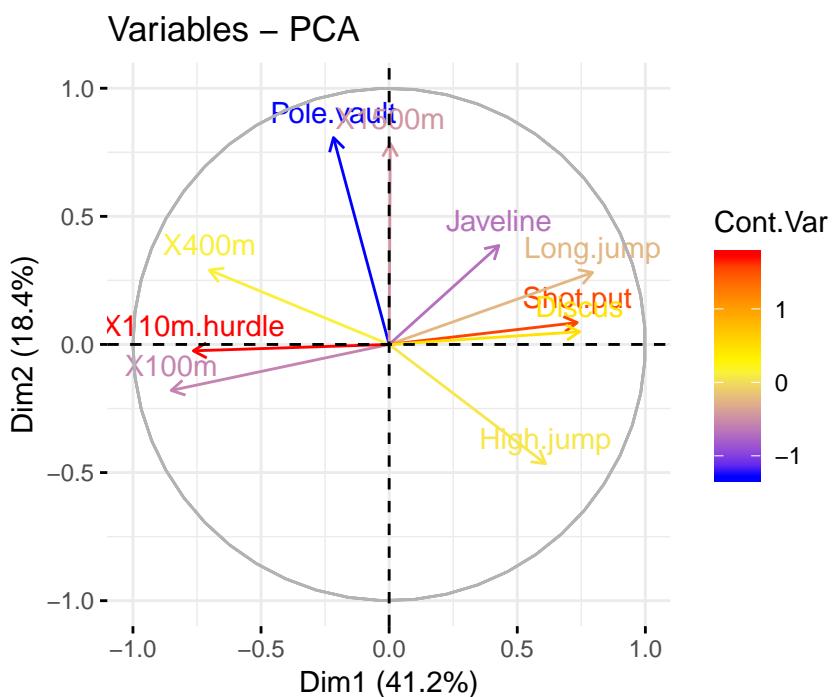


26.7 Color by a custom continuous variable

In the previous sections, we showed how to color variables by their contributions and their cos2. Note that, it's possible to color variables by any custom continuous variable. The coloring variable should have the same length as the number of active variables in the PCA (here $n = 10$).

For example, type this:

```
# Create a random continuous variable of length 10
set.seed(123)
my.cont.var <- rnorm(10)
# Color variables by the continuous variable
fviz_pca_var(res.pca, col.var = my.cont.var,
              gradient.cols = c("blue", "yellow", "red"),
              legend.title = "Cont.Var")
```



26.8 Color by groups

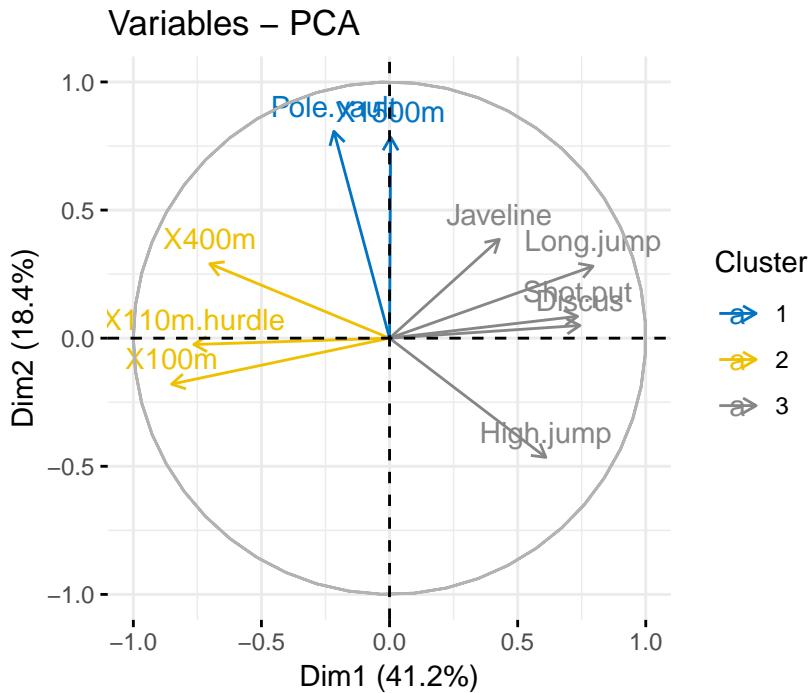
It's also possible to change the color of variables by groups defined by a qualitative/categorical variable, also called factor in R terminology.

As we don't have any grouping variable in our data sets for classifying variables, we'll create it.

In the following demo example, we start by classifying the variables into 3 groups using the kmeans clustering algorithm. Next, we use the clusters returned by the kmeans algorithm to color variables.

```
# Create a grouping variable using kmeans
# Create 3 groups of variables (centers = 3)
set.seed(123)
res.km <- kmeans(var$coord, centers = 3, nstart = 25)
grp <- as.factor(res.km$cluster)
# Color variables by groups
```

```
fviz_pca_var(res.pca, col.var = grp,
             palette = c("#0073C2FF", "#EFC000FF", "#868686FF"),
             legend.title = "Cluster")
```



26.9 Dimension description

In the section `?(pca-variable-contributions)`, we described how to highlight variables according to their contributions to the principal components.

Note also that, the function `dimdesc()` [in FactoMineR], for dimension description, can be used to identify the most significantly associated variables with a given principal component . It can be used as follow:

```
res.desc <- dimdesc(res.pca, axes = c(1,2), proba = 0.05)
# Description of dimension 1
res.desc$Dim.1
#> $quanti
#>           correlation p.value
#> Long.jump          0.794 6.06e-06
#> Discus              0.743 4.84e-05
#> Shot.put             0.734 6.72e-05
#> High.jump            0.610 1.99e-03
#> Javeline              0.428 4.15e-02
#> X400m                -0.702 1.91e-04
#> X110m.hurdle         -0.764 2.20e-05
#> X100m                -0.851 2.73e-07

# Description of dimension 2
res.desc$Dim.2
#> $quanti
#>           correlation p.value
#> Pole.vault           0.807 3.21e-06
```

```
#> X1500m      0.784 9.38e-06
#> High.jump   -0.465 2.53e-02
```

26.10 Graph of individuals

Results The results, for individuals can be extracted using the function `get_pca_ind()` [factoextra package]. Similarly to the `get_pca_var()`, the function `get_pca_ind()` provides a list of matrices containing all the results for the individuals (coordinates, correlation between individuals and axes, squared cosine and contributions)

```
ind <- get_pca_ind(res.pca)
ind
#> Principal Component Analysis Results for individuals
#> -----
#>   Name      Description
#> 1 "$coord" "Coordinates for the individuals"
#> 2 "$cos2"   "Cos2 for the individuals"
#> 3 "$contrib" "contributions of the individuals"
```

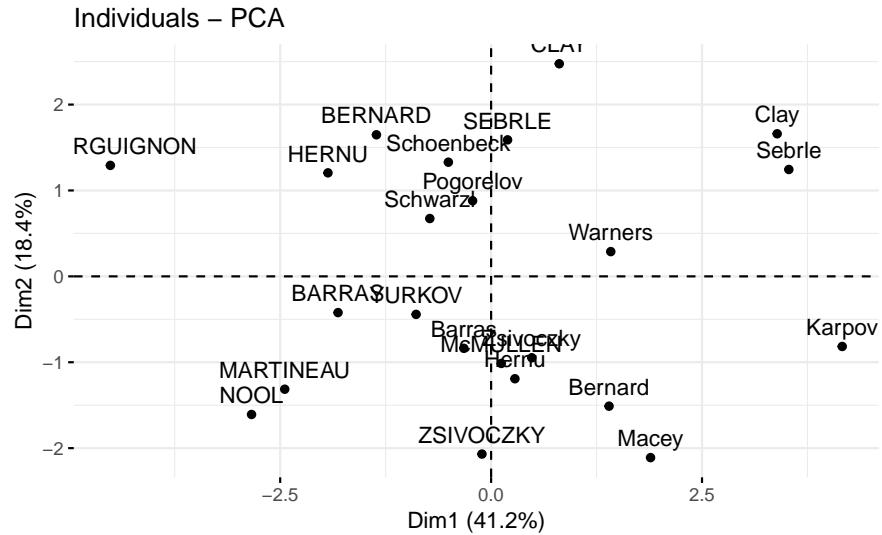
To get access to the different components, use this:

```
# Coordinates of individuals
head(ind$coord)
#>           Dim.1  Dim.2  Dim.3  Dim.4  Dim.5
#> SEBRLE     0.196  1.589  0.642  0.0839  1.1683
#> CLAY        0.808  2.475 -1.387  1.2984 -0.8250
#> BERNARD    -1.359  1.648  0.201 -1.9641  0.0842
#> YURKOV     -0.889 -0.443  2.530  0.7129  0.4078
#> ZSIVOCZKY  -0.108 -2.069 -1.334 -0.1015 -0.2015
#> McMULLEN    0.121 -1.014 -0.863  1.3416  1.6215
# Quality of individuals
head(ind$cos2)
#>           Dim.1  Dim.2  Dim.3  Dim.4  Dim.5
#> SEBRLE     0.00753 0.4975 0.08133 0.00139 0.268903
#> CLAY        0.04870 0.4570 0.14363 0.12579 0.050785
#> BERNARD    0.19720 0.2900 0.00429 0.41182 0.000757
#> YURKOV     0.09611 0.0238 0.77823 0.06181 0.020228
#> ZSIVOCZKY  0.00157 0.5764 0.23975 0.00139 0.005465
#> McMULLEN   0.00218 0.1522 0.11014 0.26649 0.389262
# Contributions of individuals
head(ind$contrib)
#>           Dim.1  Dim.2  Dim.3  Dim.4  Dim.5
#> SEBRLE     0.0403  5.971  1.448  0.0373  8.4589
#> CLAY        0.6881 14.484  6.754  8.9446  4.2179
#> BERNARD    1.9474  6.423  0.141 20.4682  0.0439
#> YURKOV     0.8331  0.463 22.452  2.6966  1.0308
#> ZSIVOCZKY  0.0123 10.122  6.246  0.0547  0.2515
#> McMULLEN   0.0155  2.431  2.610  9.5506 16.2949
```

26.11 Plots: quality and contribution

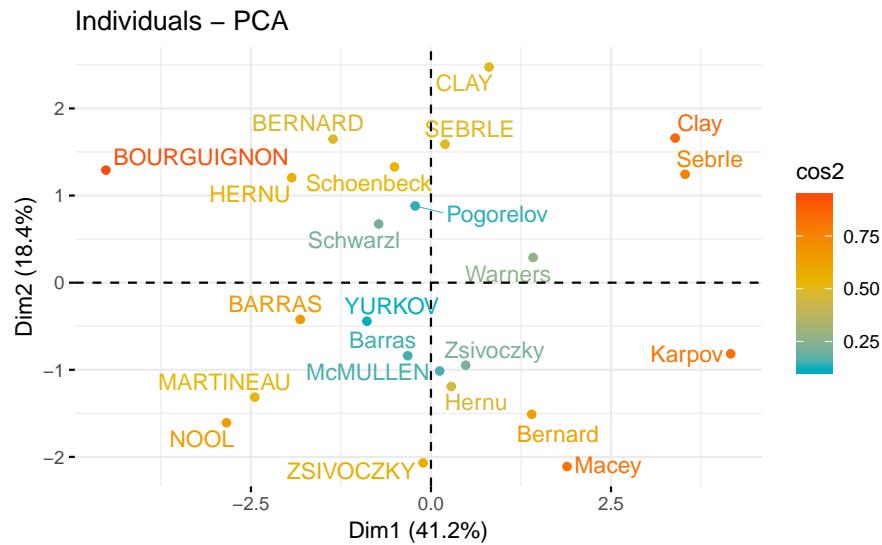
The fviz_pca_ind() is used to produce the graph of individuals. To create a simple plot, type this:

```
fviz_pca_ind(res.pca)
```



Like variables, it's also possible to color individuals by their cos2 values:

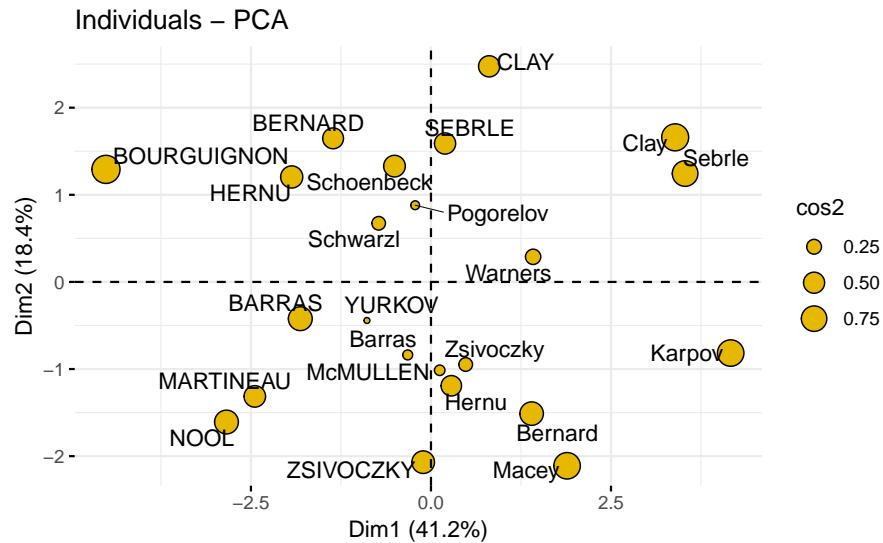
```
fviz_pca_ind(res.pca, col.ind = "cos2",
             gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07"),
             repel = TRUE # Avoid text overlapping (slow if many points)
           )
```



Note that, individuals that are similar are grouped together on the plot.

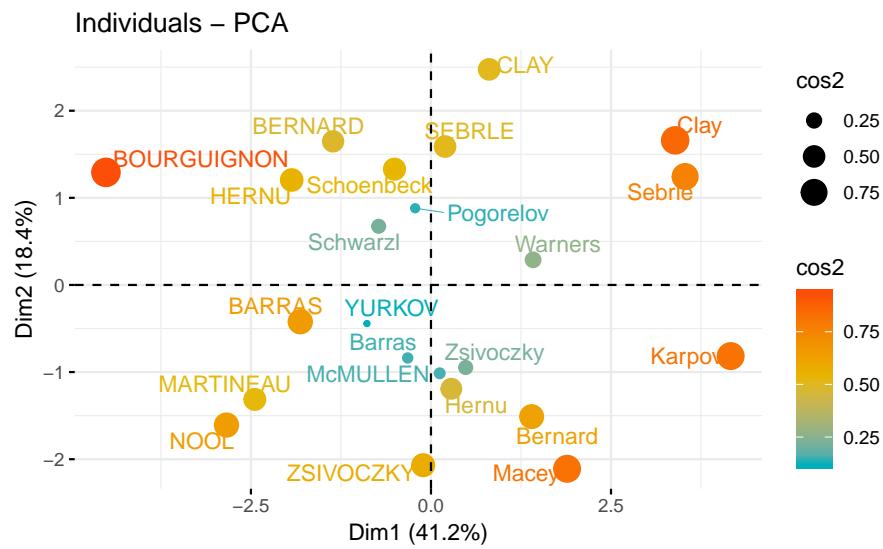
You can also change the point size according the cos2 of the corresponding individuals:

```
fviz_pca_ind(res.pca, pointsize = "cos2",
             pointshape = 21, fill = "#E7B800",
             repel = TRUE # Avoid text overlapping (slow if many points)
           )
```



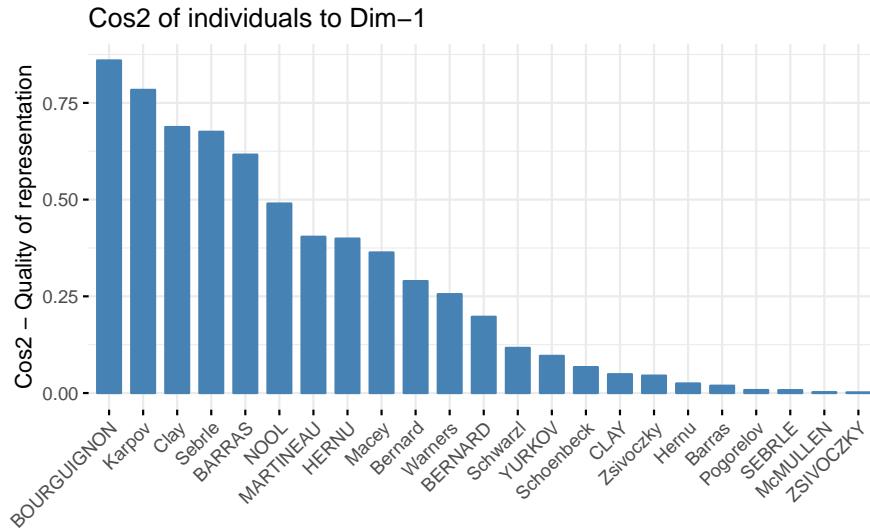
To change both point size and color by cos2, try this:

```
fviz_pca_ind(res.pca, col.ind = "cos2", pointsize = "cos2",
             gradient.cols = c("#00AFBB", "#E7B800", "#FC4E07"),
             repel = TRUE # Avoid text overlapping (slow if many points)
)
```



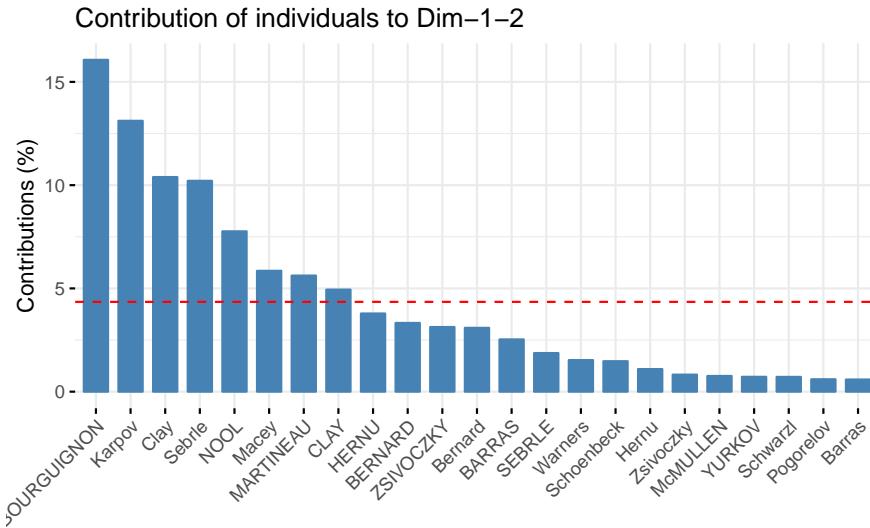
To create a bar plot of the quality of representation (cos2) of individuals on the factor map, you can use the function fviz_cos2() as previously described for variables:

```
fviz_cos2(res.pca, choice = "ind")
```



To visualize the contribution of individuals to the first two principal components, type this:

```
# Total contribution on PC1 and PC2
fviz_contrib(res.pca, choice = "ind", axes = 1:2)
```

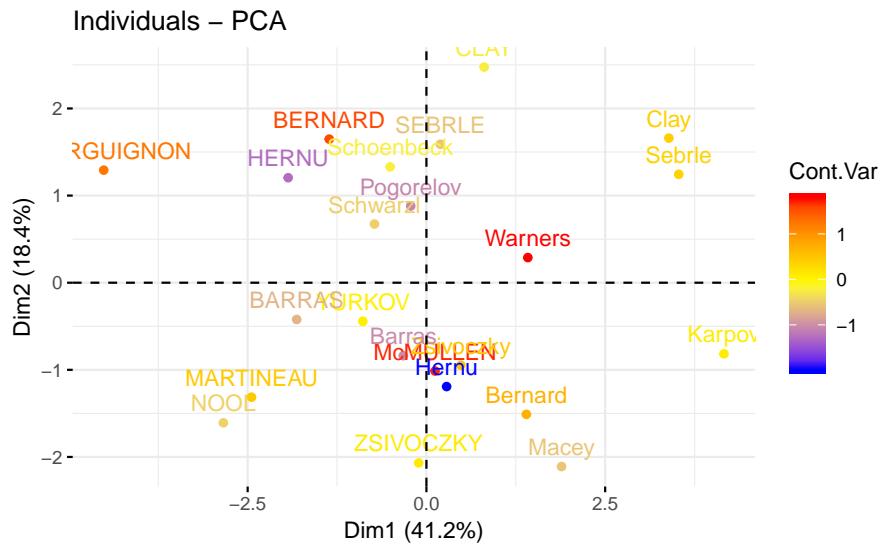


26.12 Color by a custom continuous variable

As for variables, individuals can be colored by any custom continuous variable by specifying the argument col.ind.

For example, type this:

```
# Create a random continuous variable of length 23,
# Same length as the number of active individuals in the PCA
set.seed(123)
my.cont.var <- rnorm(23)
# Color individuals by the continuous variable
fviz_pca_ind(res.pca, col.ind = my.cont.var,
              gradient.cols = c("blue", "yellow", "red"),
              legend.title = "Cont.Var")
```



26.13 Color by groups

Here, we describe how to color individuals by group. Additionally, we show how to add concentration ellipses and confidence ellipses by groups. For this, we'll use the iris data as demo data sets.

Iris data sets look like this:

```
head(iris, 3)
#>   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1      5.1       3.5      1.4       0.2   setosa
#> 2      4.9       3.0      1.4       0.2   setosa
#> 3      4.7       3.2      1.3       0.2   setosa
```

The column “Species” will be used as grouping variable. We start by computing principal component analysis as follow:

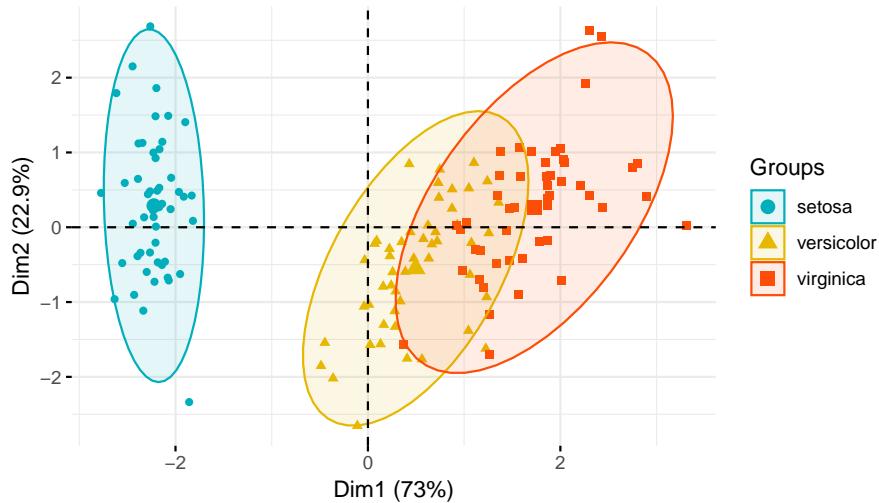
```
# The variable Species (index = 5) is removed
# before PCA analysis
iris.pca <- PCA(iris[,-5], graph = FALSE)
```

In the R code below: the argument habillage or col.ind can be used to specify the factor variable for coloring the individuals by groups.

To add a concentration ellipse around each group, specify the argument addEllipses = TRUE. The argument palette can be used to change group colors.

```
fviz_pca_ind(iris.pca,
              geom.ind = "point", # show points only (nbut not "text")
              col.ind = iris$Species, # color by groups
              palette = c("#00AFBB", "#E7B800", "#FC4E07"),
              addEllipses = TRUE, # Concentration ellipses
              legend.title = "Groups"
            )
```

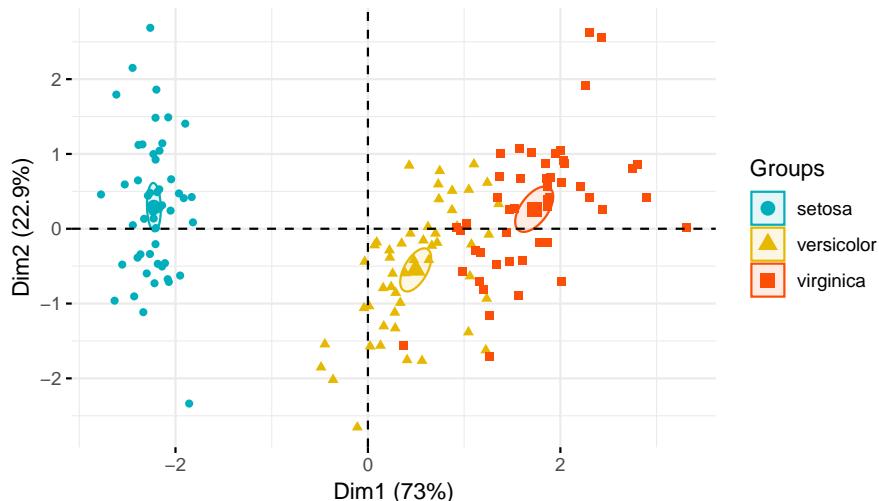
Individuals – PCA



To remove the group mean point, specify the argument `mean.point = FALSE`. If you want confidence ellipses instead of concentration ellipses, use `ellipse.type = "confidence"`.

```
# Add confidence ellipses
fviz_pca_ind(iris.pca, geom.ind = "point", col.ind = iris$Species,
             palette = c("#00AFBB", "#E7B800", "#FC4E07"),
             addEllipses = TRUE, ellipse.type = "confidence",
             legend.title = "Groups"
)
```

Individuals – PCA



Note that, allowed values for palette include:

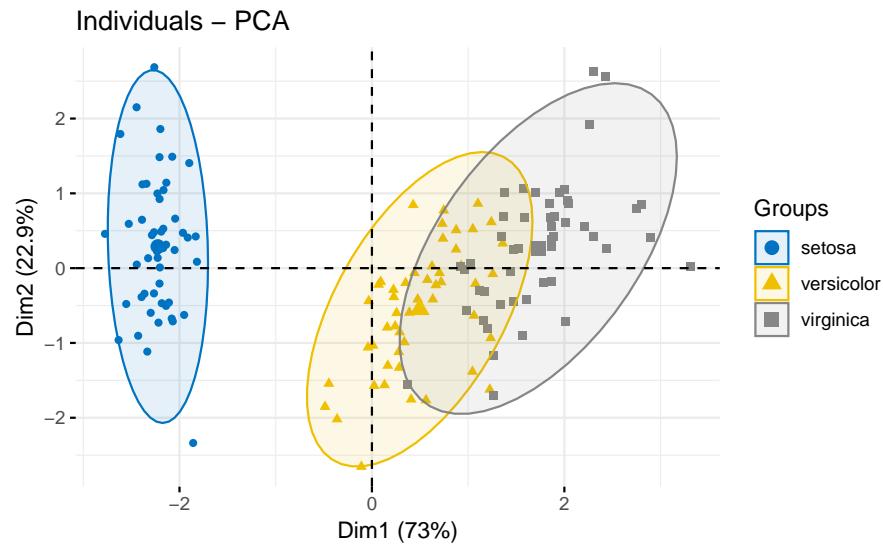
- “grey” for grey color palettes;
- brewer palettes e.g. “RdBu”, “Blues”, ...; To view all, type this in R: `RColorBrewer::display.brewer.all()`.
- custom color palette e.g. `c("blue", "red")`; and scientific journal palettes from `ggsci` R package, e.g.: “npg”, “aaas”, * “lancet”, “jco”, “ucscgb”, “uchicago”, “simpsons” and “rickandmorty”. For example, to use the `jco` (journal of clinical oncology) color palette, type this:

```
fviz_pca_ind(iris.pca,
             label = "none", # hide individual labels
             habillage = iris$Species, # color by groups
```

```

    addEllipses = TRUE, # Concentration ellipses
    palette = "jco"
)

```



26.14 Graph customization

Note that, `fviz_pca_ind()` and `fviz_pca_var()` and related functions are wrapper around the core function `fviz()` [in factoextra]. `fviz()` is a wrapper around the function `ggscatter()` [in ggpubr]. Therefore, further arguments, to be passed to the function `fviz()` and `ggscatter()`, can be specified in `fviz_pca_ind()` and `fviz_pca_var()`.

Here, we present some of these additional arguments to customize the PCA graph of variables and individuals.

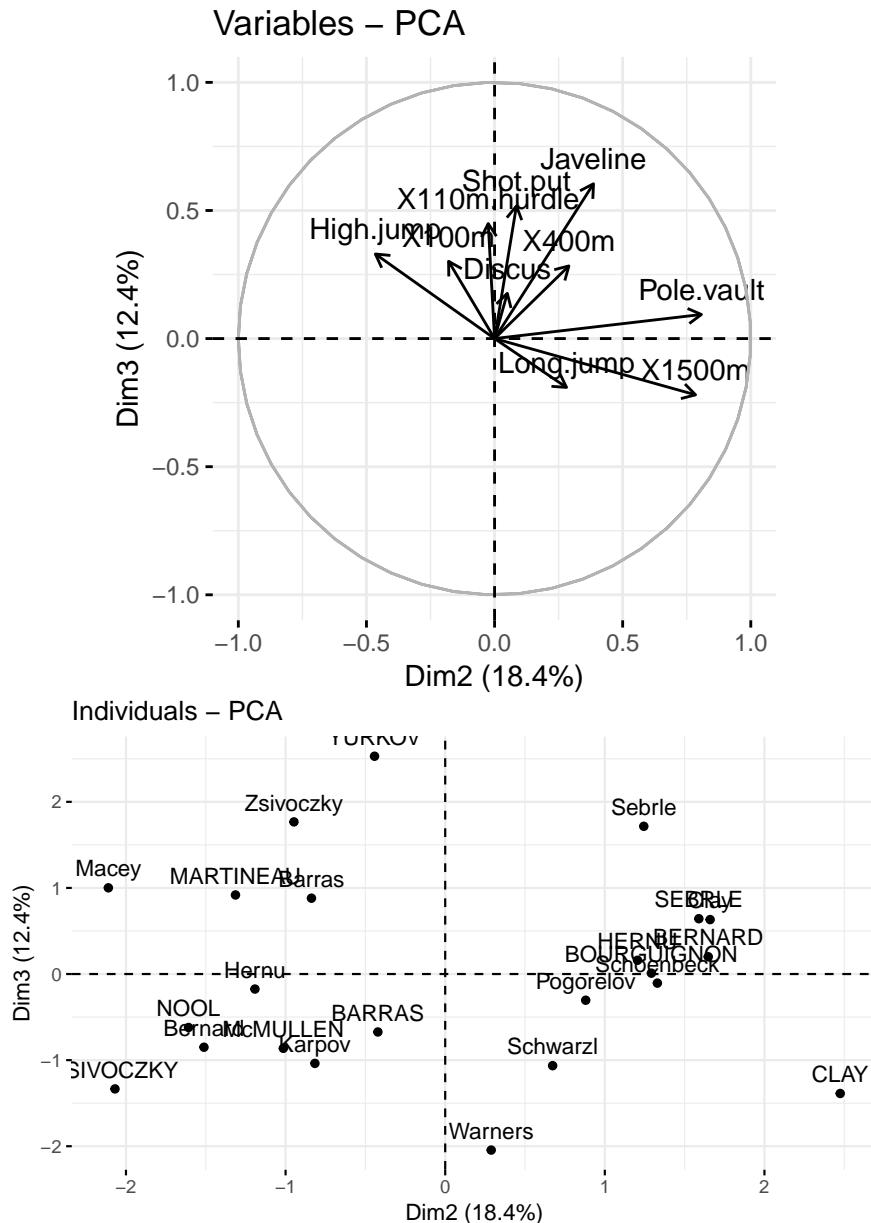
26.14.1 Dimensions

By default, variables/individuals are represented on dimensions 1 and 2. If you want to visualize them on dimensions 2 and 3, for example, you should specify the argument `axes = c(2, 3)`.

```

# Variables on dimensions 2 and 3
fviz_pca_var(res.pca, axes = c(2, 3))
# Individuals on dimensions 2 and 3
fviz_pca_ind(res.pca, axes = c(2, 3))

```

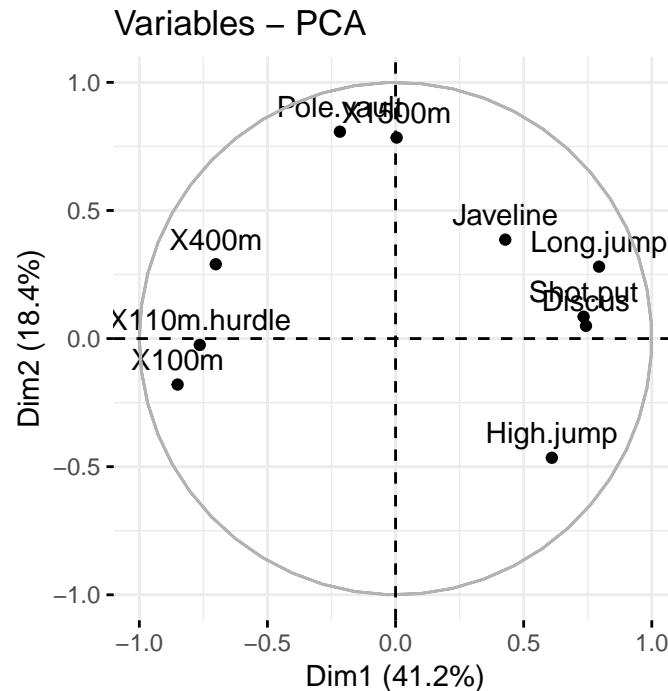


Plot elements: point, text, arrow The argument geom (for geometry) and derivatives are used to specify the geometry elements or graphical elements to be used for plotting.

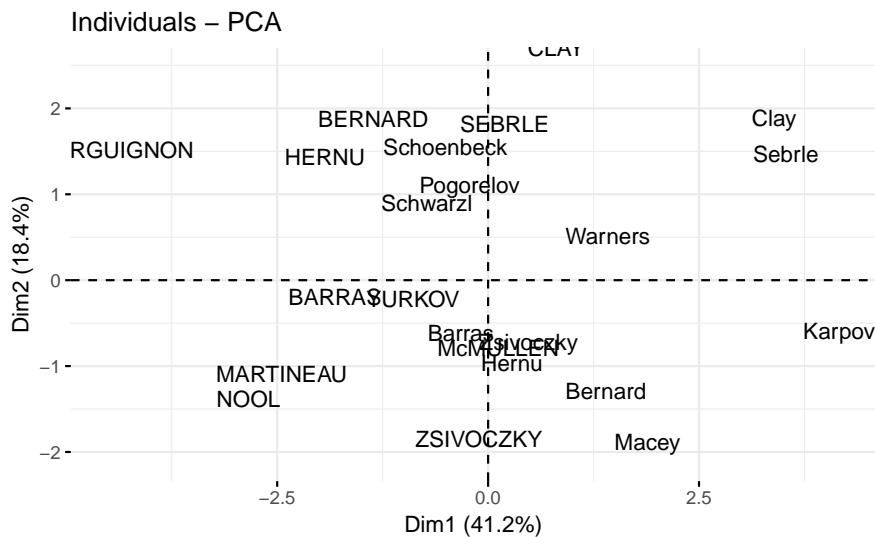
1. geom.var: a text specifying the geometry to be used for plotting variables. Allowed values are the combination of c("point", "arrow", "text").
 - Use geom.var = "point", to show only points;
 - Use geom.var = "text" to show only text labels;
 - Use geom.var = c("point", "text") to show both points and text labels
 - Use geom.var = c("arrow", "text") to show arrows and labels (default).

For example, type this:

```
# Show variable points and text labels
fviz_pca_var(res.pca, geom.var = c("point", "text"))
```



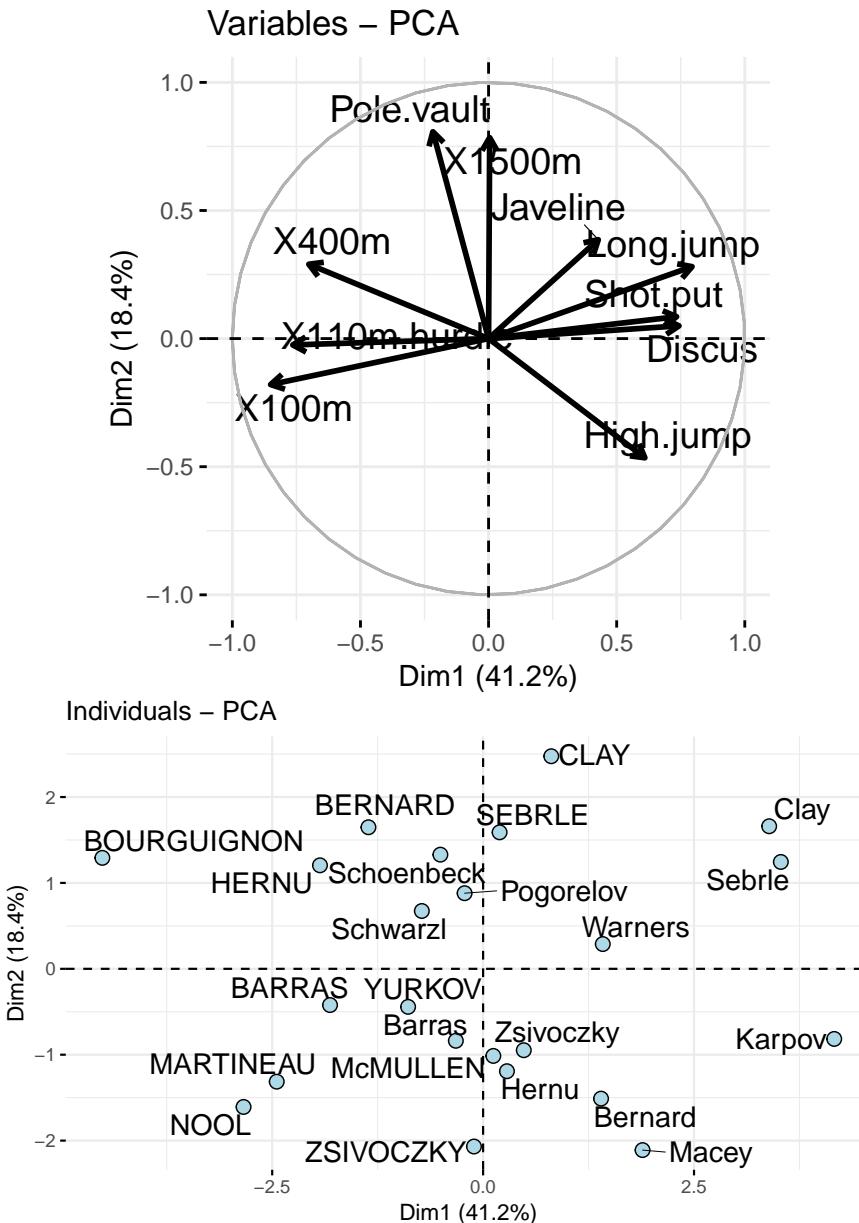
```
# Show individuals text labels only
fviz_pca_ind(res.pca, geom.ind = "text")
```



26.15 Size and shape of plot elements

```
# Change the size of arrows and labels
fviz_pca_var(res.pca, arrowsize = 1, labelszie = 5,
             repel = TRUE)
# Change points size, shape and fill color
# Change labelsize
fviz_pca_ind(res.pca,
             pointsize = 3, pointshape = 21, fill = "lightblue",
```

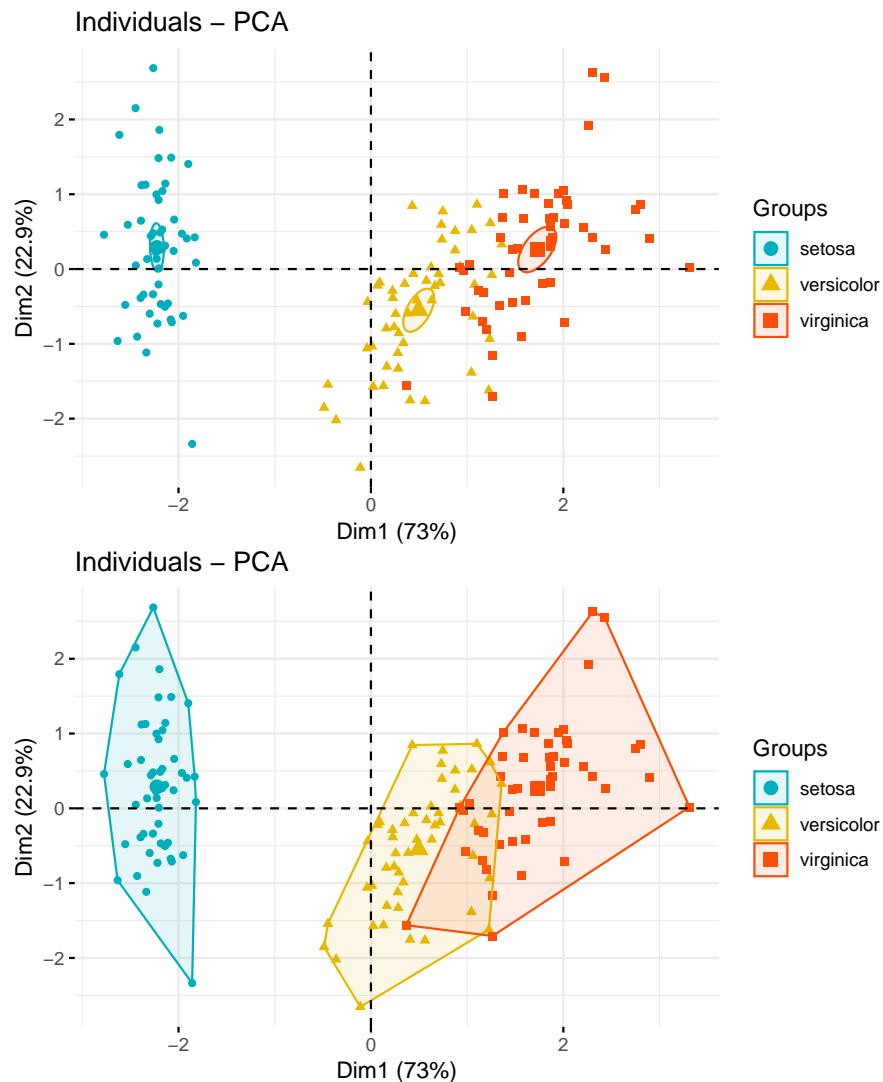
```
labelsize = 5, repel = TRUE)
```



26.16 Ellipses

```
# Add confidence ellipses
fviz_pca_ind(iris.pca, geom.ind = "point",
              col.ind = iris$Species, # color by groups
              palette = c("#00AFBB", "#E7B800", "#FC4E07"),
              addEllipses = TRUE, ellipse.type = "confidence",
              legend.title = "Groups"
)
# Convex hull
```

```
fviz_pca_ind(iris.pca, geom.ind = "point",
             col.ind = iris$Species, # color by groups
             palette = c("#00AFBB", "#E7B800", "#FC4E07"),
             addEllipses = TRUE, ellipse.type = "convex",
             legend.title = "Groups"
)
```



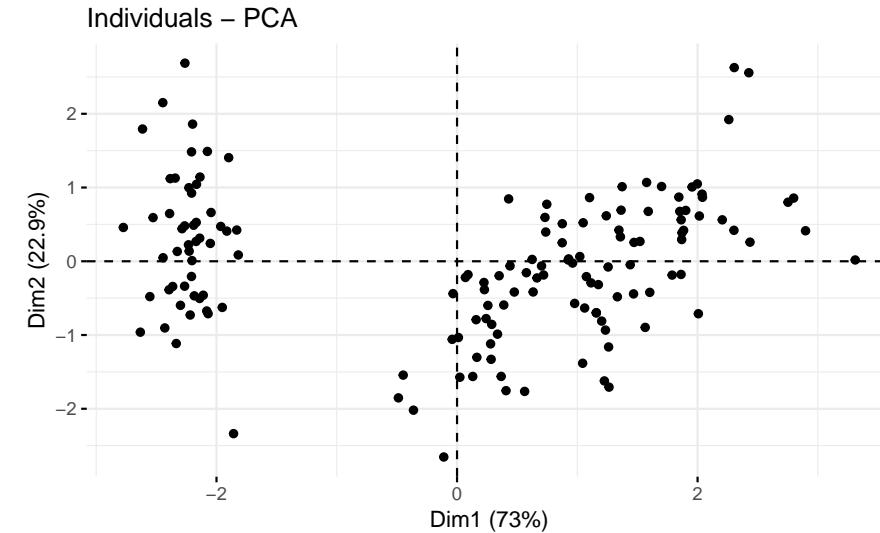
26.17 Group mean points

When coloring individuals by groups (section `?color-ind-by-groups`), the mean points of groups (barycenters) are also displayed by default.

To remove the mean points, use the argument `mean.point = FALSE`.

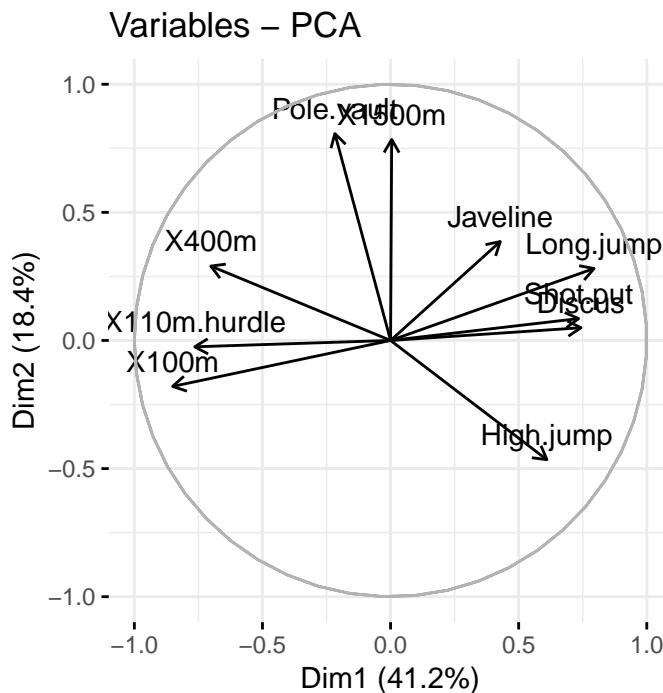
```
fviz_pca_ind(iris.pca,
             geom.ind = "point", # show points only (but not "text")
             group.ind = iris$Species, # color by groups
             legend.title = "Groups",
```

```
mean.point = FALSE)
```



26.18 Axis lines

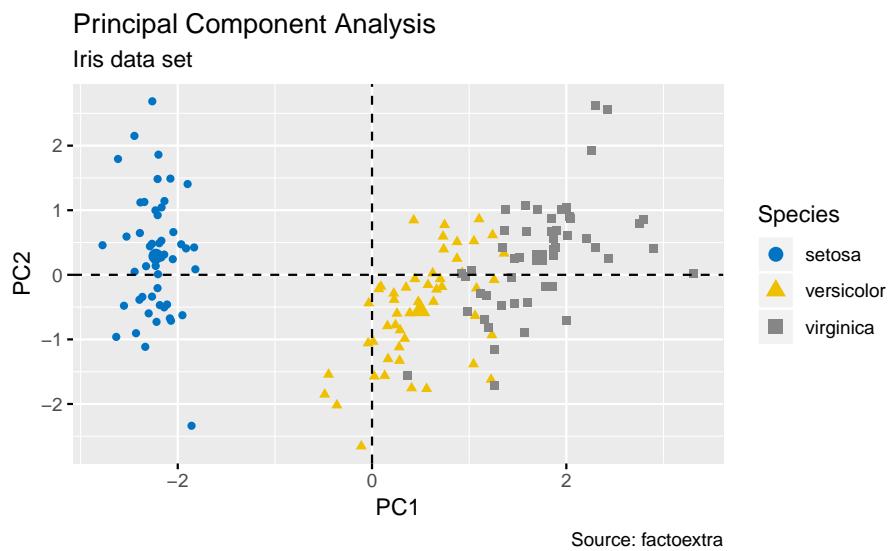
```
fviz_pca_var(res.pca, axes.linetype = "blank")
```



26.19 Graphical parameters

To change easily the graphical of any ggplots, you can use the function `ggpar()` [ggpubr package]

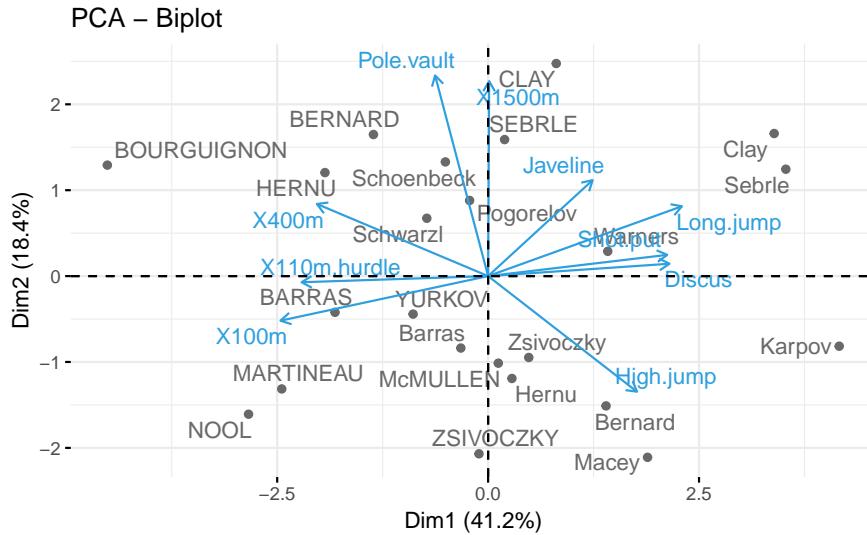
```
ind.p <- fviz_pca_ind(iris.pca, geom = "point", col.ind = iris$Species)
ggpubr::ggpar(ind.p,
  title = "Principal Component Analysis",
  subtitle = "Iris data set",
  caption = "Source: factoextra",
  xlab = "PC1", ylab = "PC2",
  legend.title = "Species", legend.position = "top",
  ggtheme = theme_gray(), palette = "jco"
)
```



26.20 Biplot

To make a simple biplot of individuals and variables, type this:

```
fviz_pca_biplot(res.pca, repel = TRUE,
  col.var = "#2E9FDF", # Variables color
  col.ind = "#696969" # Individuals color
)
```

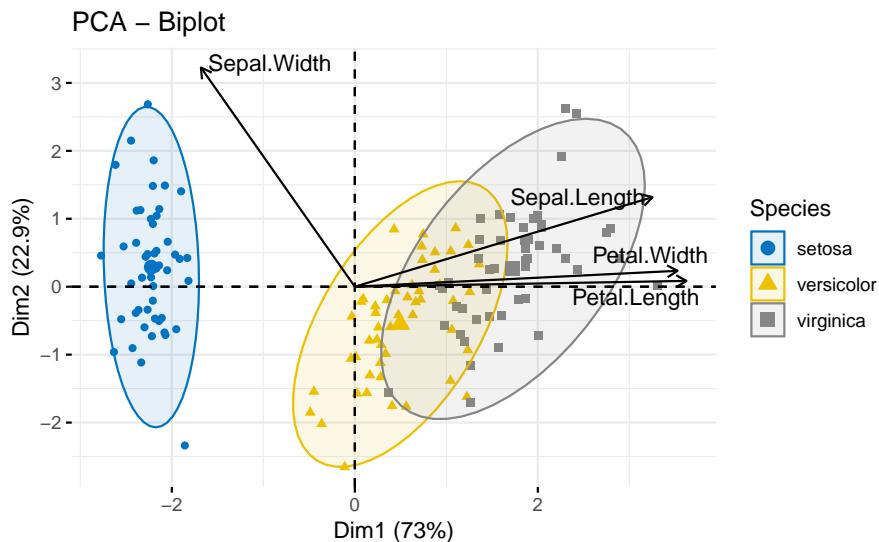


Note that, the biplot might be only useful when there is a low number of variables and individuals in the data set; otherwise the final plot would be unreadable.

Note also that, the coordinate of individuals and variables are not constructed on the same space. Therefore, in the biplot, you should mainly focus on the direction of variables but not on their absolute positions on the plot.

Roughly speaking a biplot can be interpreted as follow: * an individual that is on the same side of a given variable has a high value for this variable; * an individual that is on the opposite side of a given variable has a low value for this variable.

```
fviz_pca_biplot(iris.pca,
  col.ind = iris$Species, palette = "jco",
  addEllipses = TRUE, label = "var",
  col.var = "black", repel = TRUE,
  legend.title = "Species")
```



In the following example, we want to color both individuals and variables by groups. The trick is to use pointshape = 21 for individual points. This particular point shape can be filled by a color using the argument fill.ind. The border line color of individual points is set to “black” using col.ind. To color variable by groups, the argument col.var will be used.

To customize individuals and variable colors, we use the helper functions `fill_palette()` and `color_palette()` [in `ggpubr` package].

```
fviz_pca_biplot(iris.pca,
  # Fill individuals by groups
  geom.ind = "point",
  pointshape = 21,
  pointsize = 2.5,
  fill.ind = iris$Species,
  col.ind = "black",
  # Color variable by groups
  col.var = factor(c("sepal", "sepal", "petal", "petal")),

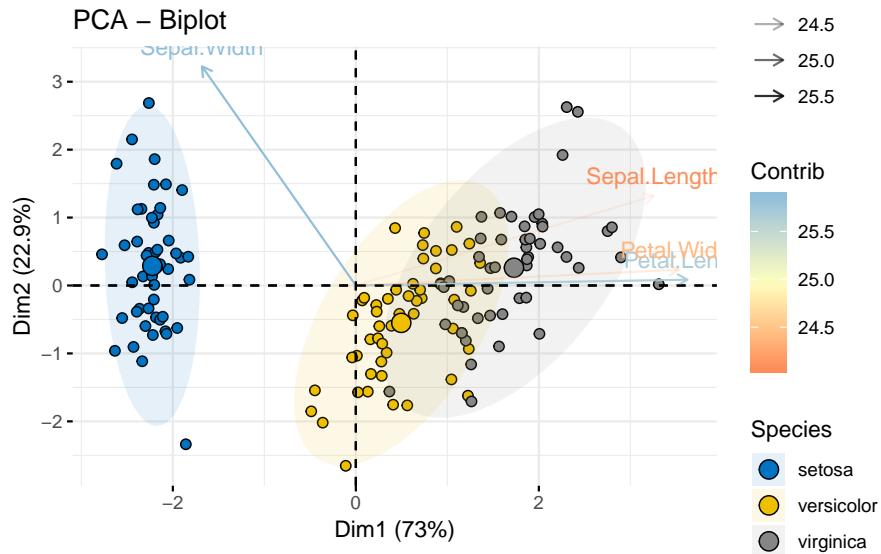
  legend.title = list(fill = "Species", color = "Clusters"),
  repel = TRUE      # Avoid label overplotting
) +
  ggpubr::fill_palette("jco") +      # Individual fill color
  ggpubr::color_palette("npg")       # Variable colors
```



Another complex example is to color individuals by groups (discrete color) and variables by their contributions to the principal components (gradient colors). Additionally, we'll change the transparency of variables by their contributions using the argument `alpha.var`.

```
fviz_pca_biplot(iris.pca,
  # Individuals
  geom.ind = "point",
  fill.ind = iris$Species, col.ind = "black",
  pointshape = 21, pointsize = 2,
  palette = "jco",
  addEllipses = TRUE,
  # Variables
  alpha.var = "contrib", col.var = "contrib",
  gradient.cols = "RdYlBu",

  legend.title = list(fill = "Species", color = "Contrib",
                      alpha = "Contrib")
)
```



26.21 Supplementary elements

Definition and types As described above (section ?(pca-data-format)), the decathlon2 data sets contain supplementary continuous variables (quanti.sup, columns 11:12), supplementary qualitative variables (quali.sup, column 13) and supplementary individuals (ind.sup, rows 24:27).

Supplementary variables and individuals are not used for the determination of the principal components. Their coordinates are predicted using only the information provided by the performed principal component analysis on active variables/individuals.

Specification in PCA To specify supplementary individuals and variables, the function PCA() can be used as follow:

```
res.pca <- PCA(decathlon2, ind.sup = 24:27,
                  quanti.sup = 11:12, quali.sup = 13, graph=FALSE)
```

26.22 Quantitative variables

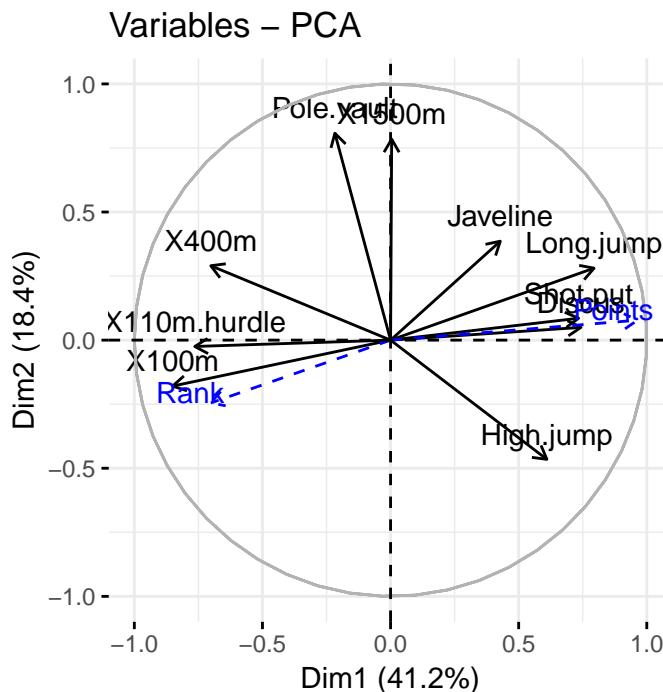
Predicted results (coordinates, correlation and cos2) for the supplementary quantitative variables:

```
res.pca$quanti.sup
#> $coord
#>      Dim.1   Dim.2   Dim.3   Dim.4   Dim.5
#> Rank -0.701 -0.2452 -0.183  0.0558 -0.0738
#> Points 0.964  0.0777  0.158 -0.1662 -0.0311
#>
#> $cor
#>      Dim.1   Dim.2   Dim.3   Dim.4   Dim.5
#> Rank -0.701 -0.2452 -0.183  0.0558 -0.0738
#> Points 0.964  0.0777  0.158 -0.1662 -0.0311
#>
#> $cos2
#>      Dim.1   Dim.2   Dim.3   Dim.4   Dim.5
```

```
#> Rank  0.492 0.06012 0.0336 0.00311 0.00545
#> Points 0.929 0.00603 0.0250 0.02763 0.00097
```

Visualize all variables (active and supplementary ones):

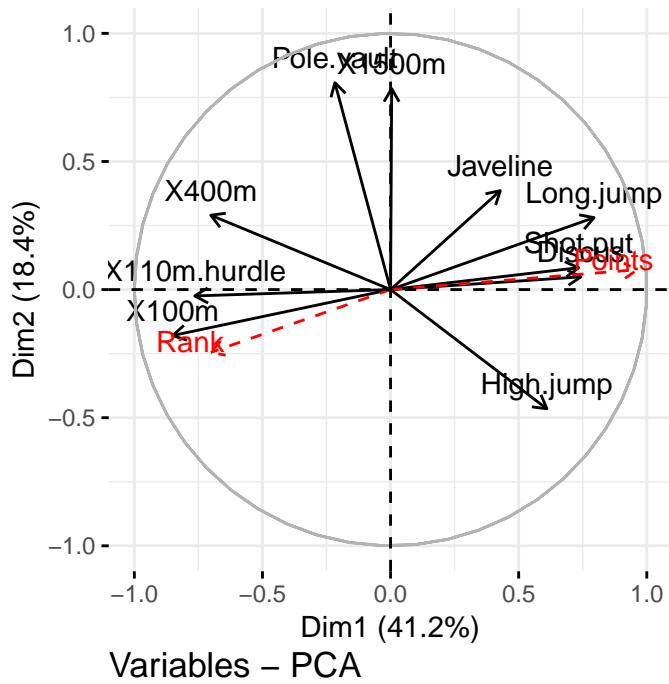
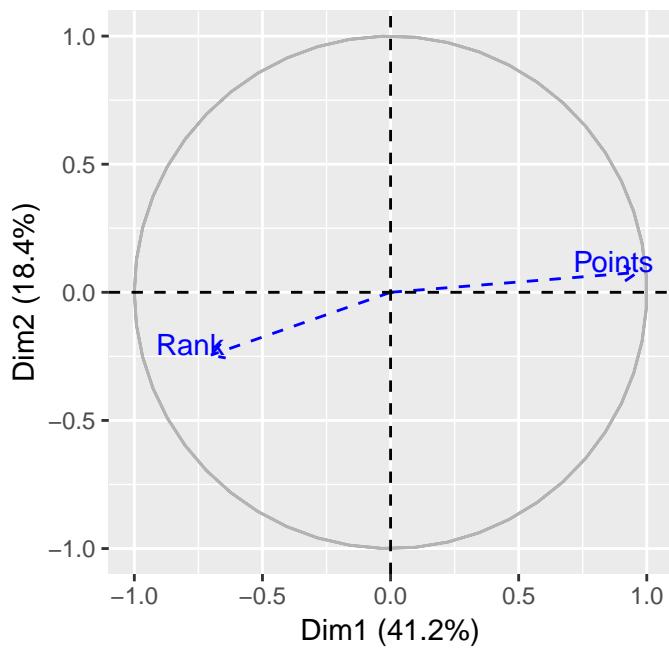
```
fviz_pca_var(res.pca)
```

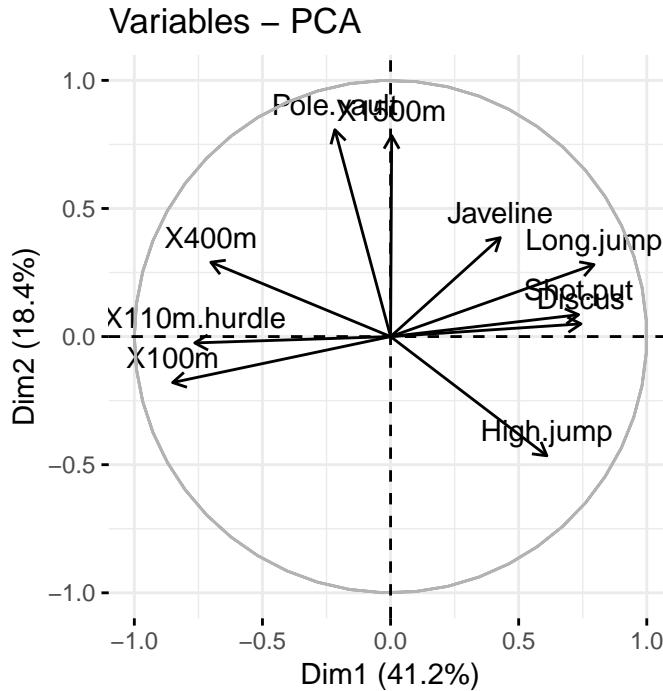


Note that, by default, supplementary quantitative variables are shown in blue color and dashed lines.

Further arguments to customize the plot:

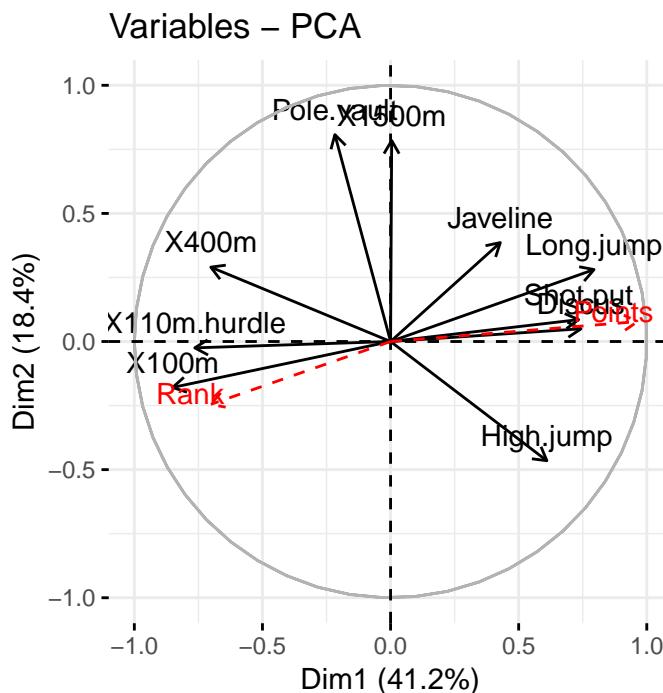
```
# Change color of variables
fviz_pca_var(res.pca,
             col.var = "black",      # Active variables
             col.quanti.sup = "red" # Suppl. quantitative variables
)
# Hide active variables on the plot,
# show only supplementary variables
fviz_pca_var(res.pca, invisible = "var")
# Hide supplementary variables
fviz_pca_var(res.pca, invisible = "quanti.sup")
```

Variables – PCA**Variables – PCA**



Using the `fviz_pca_var()`, the quantitative supplementary variables are displayed automatically on the correlation circle plot. Note that, you can add the `quanti.sup` variables manually, using the `fviz_add()` function, for further customization. An example is shown below.

```
# Plot of active variables
p <- fviz_pca_var(res.pca, invisible = "quanti.sup")
# Add supplementary active variables
fviz_add(p, res.pca$quanti.sup$coord,
          geom = c("arrow", "text"),
          color = "red")
```



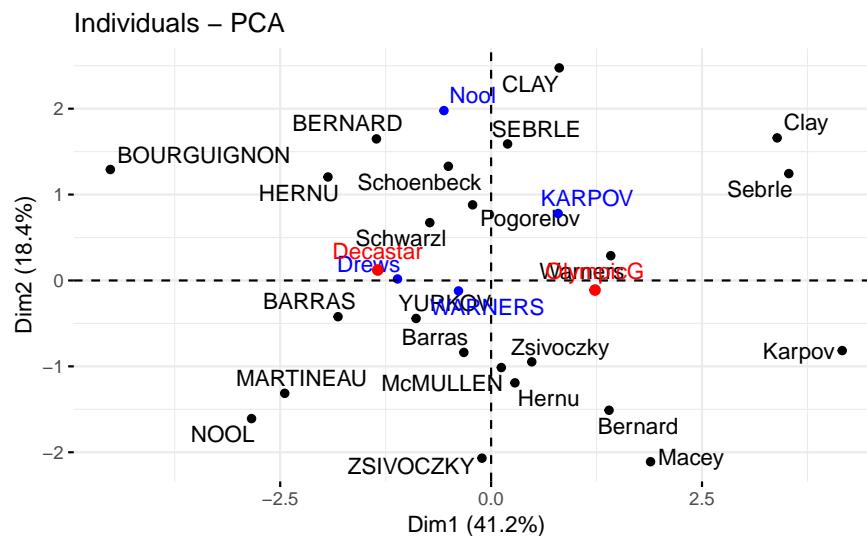
26.23 Individuals

Predicted results for the supplementary individuals (ind.sup):

```
res.pca$ind.sup
#> $coord
#>           Dim.1   Dim.2   Dim.3   Dim.4   Dim.5
#> KARPOV    0.795  0.7795 -1.633  1.724 -0.7507
#> WARNERS   -0.386 -0.1216 -1.739 -0.706 -0.0323
#> Nool      -0.559  1.9775 -0.483 -2.278 -0.2546
#> Drews     -1.109  0.0174 -3.049 -1.534 -0.3264
#>
#> $cos2
#>           Dim.1   Dim.2   Dim.3   Dim.4   Dim.5
#> KARPOV    0.0510 4.91e-02 0.2155 0.2403 0.045549
#> WARNERS   0.0242 2.40e-03 0.4904 0.0809 0.000169
#> Nool      0.0290 3.62e-01 0.0216 0.4811 0.006008
#> Drews     0.0921 2.27e-05 0.6956 0.1762 0.007974
#>
#> $dist
#> KARPOV WARNERS   Nool   Drews
#> 3.52    2.48     3.28   3.66
```

Visualize all individuals (active and supplementary ones). On the graph, you can add also the supplementary qualitative variables (quali.sup), which coordinates is accessible using res.pca\$quali.sup\$coord.

```
p <- fviz_pca_ind(res.pca, col.ind.sup = "blue", repel = TRUE)
p <- fviz_add(p, res.pca$quali.sup$coord, color = "red")
p
```



Supplementary individuals are shown in blue. The levels of the supplementary qualitative variable are shown in red color.

26.24 Qualitative variables

In the previous section, we showed that you can add the supplementary qualitative variables on individuals plot using fviz_add().

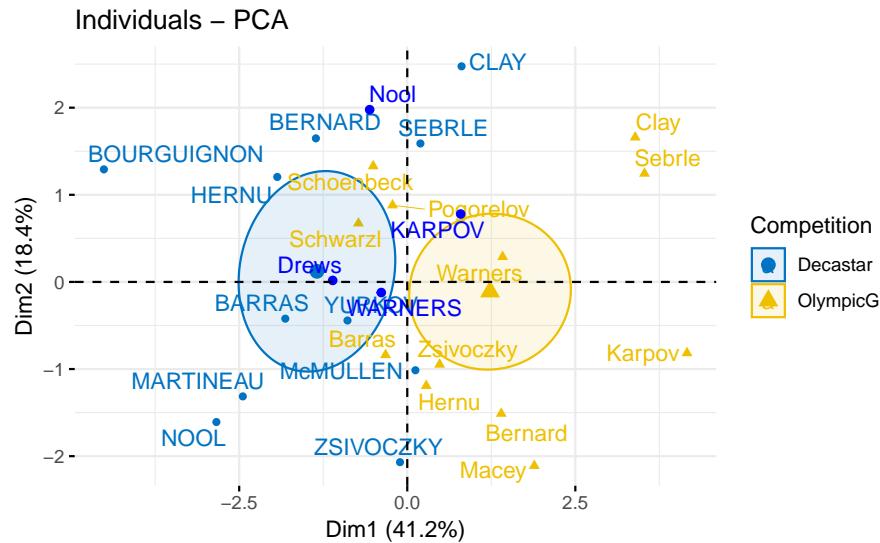
Note that, the supplementary qualitative variables can be also used for coloring individuals by groups. This can help to interpret the data. The data sets decathlon2 contain a supplementary qualitative variable at columns 13 corresponding to the type of competitions.

The results concerning the supplementary qualitative variable are:

```
res.pca$quali
#> $coord
#>           Dim.1   Dim.2   Dim.3   Dim.4   Dim.5
#> Decastar -1.34  0.122 -0.0379  0.181  0.134
#> OlympicG  1.23 -0.112  0.0347 -0.166 -0.123
#>
#> $cos2
#>           Dim.1   Dim.2   Dim.3   Dim.4   Dim.5
#> Decastar 0.905 0.00744 0.00072 0.0164 0.00905
#> OlympicG 0.905 0.00744 0.00072 0.0164 0.00905
#>
#> $v.test
#>           Dim.1   Dim.2   Dim.3   Dim.4   Dim.5
#> Decastar -2.97  0.403 -0.153  0.897  0.72
#> OlympicG  2.97 -0.403  0.153 -0.897 -0.72
#>
#> $dist
#> Decastar OlympicG
#>      1.41      1.29
#>
#> $eta2
#>           Dim.1   Dim.2   Dim.3   Dim.4   Dim.5
#> Competition 0.401 0.0074 0.00106 0.0366 0.0236
```

To color individuals by a supplementary qualitative variable, the argument habillage is used to specify the index of the supplementary qualitative variable. Historically, this argument name comes from the FactoMineR package. It's a french word meaning "dressing" in english. To keep consistency between FactoMineR and factoextra, we decided to keep the same argument name

```
fviz_pca_ind(res.pca, habillage = 13,
             addEllipses = TRUE, ellipse.type = "confidence",
             palette = "jco", repel = TRUE)
```

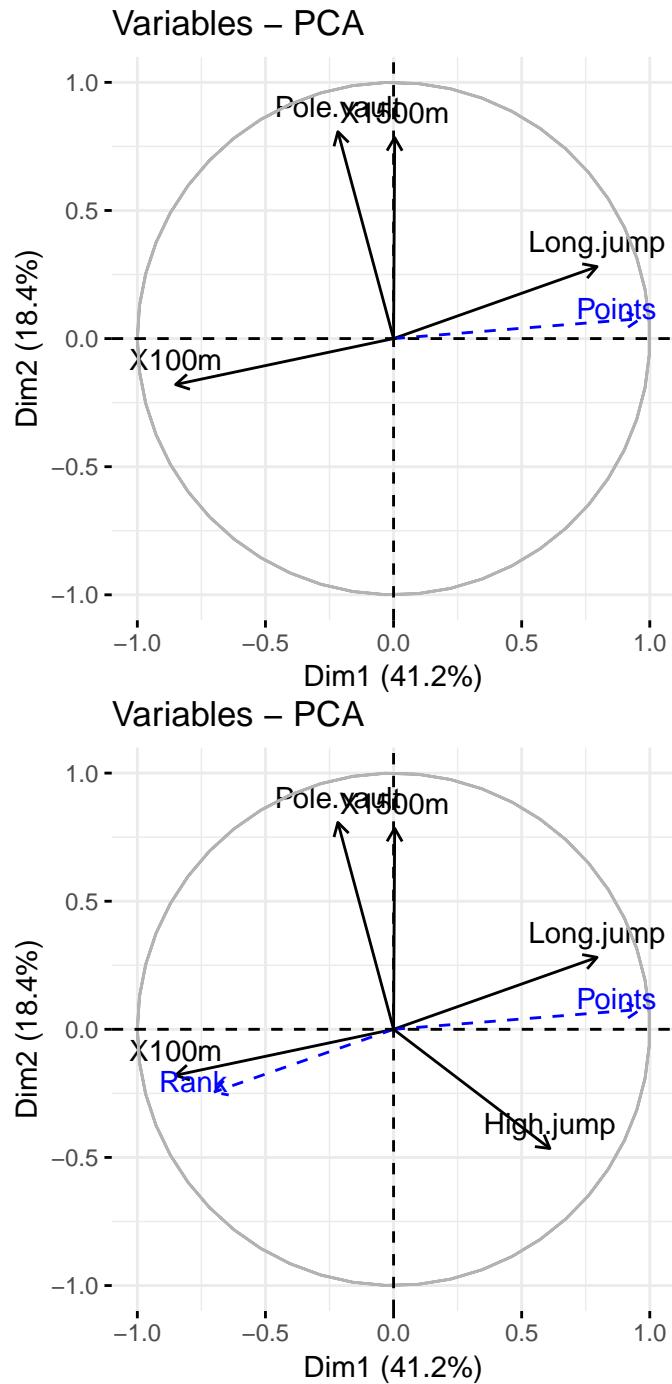


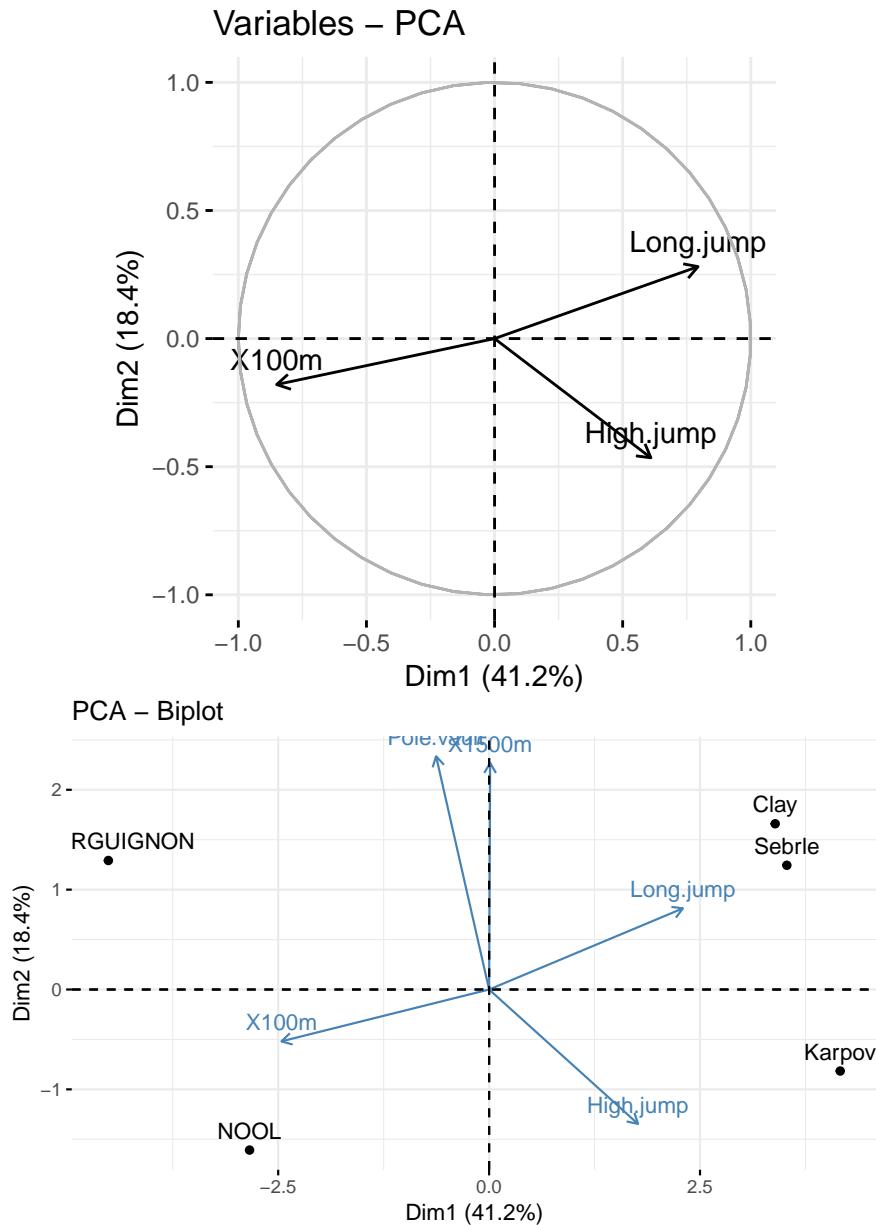
Recall that, to remove the mean points of groups, specify the argument `mean.point = FALSE`.

26.25 Filtering results

If you have many individuals/variable, it's possible to visualize only some of them using the arguments `select.ind` and `select.var`.

```
# Visualize variable with cos2 >= 0.6
fviz_pca_var(res.pca, select.var = list(cos2 = 0.6))
# Top 5 active variables with the highest cos2
fviz_pca_var(res.pca, select.var= list(cos2 = 5))
# Select by names
name <- list(name = c("Long.jump", "High.jump", "X100m"))
fviz_pca_var(res.pca, select.var = name)
# top 5 contributing individuals and variable
fviz_pca_biplot(res.pca, select.ind = list(contrib = 5),
                select.var = list(contrib = 5),
                ggtheme = theme_minimal())
```





When the selection is done according to the contribution values, supplementary individuals/variables are not shown because they don't contribute to the construction of the axes.

26.26 Exporting results

Export plots to PDF/PNG files The factoextra package produces a ggplot2-based graphs. To save any ggplots, the standard R code is as follow:

```
# Print the plot to a pdf file
pdf("myplot.pdf")
print(myplot)
dev.off()
```

In the following examples, we'll show you how to save the different graphs into pdf or png files.

The first step is to create the plots you want as an R object:

```
# Scree plot
scree.plot <- fviz_eig(res.pca)
# Plot of individuals
ind.plot <- fviz_pca_ind(res.pca)
# Plot of variables
var.plot <- fviz_pca_var(res.pca)

pdf(file.path(data_out_dir, "PCA.pdf")) # Create a new pdf device
print(scree.plot)
print(ind.plot)
print(var.plot)
dev.off() # Close the pdf device
#> pdf
#> 2
```

Note that, using the above R code will create the PDF file into your current working directory.

To see the path of your current working directory, type getwd() in the R console.

To print each plot to specific png file, the R code looks like this:

```
# Print scree plot to a png file
png(file.path(data_out_dir, "pca-scree-plot.png"))
print(scree.plot)
dev.off()
#> pdf
#> 2
# Print individuals plot to a png file
png(file.path(data_out_dir, "pca-variables.png"))
print(var.plot)
dev.off()
#> pdf
#> 2
# Print variables plot to a png file
png(file.path(data_out_dir, "pca-individuals.png"))
print(ind.plot)
dev.off()
#> pdf
#> 2
```

Another alternative, to export ggplots, is to use the function ggexport() [in ggpibr package]. We like ggexport(), because it's very simple. With one line R code, it allows us to export individual plots to a file (pdf, eps or png) (one plot per page). It can also arrange the plots (2 plot per page, for example) before exporting them. The examples below demonstrates how to export ggplots using ggexport().

Export individual plots to a pdf file (one plot per page):

```
library(ggpibr)
#> Loading required package: magrittr
ggexport(plotlist = list(scree.plot, ind.plot, var.plot),
        filename = file.path(data_out_dir, "PCA.pdf"))
#> file saved to /home/datasience/repos/machine-learning-rsuite/export/PCA.pdf
```

Arrange and export. Specify nrow and ncol to display multiple plots on the same page:

```
ggexport(plotlist = list(scree.plot, ind.plot, var.plot),
        nrow = 2, ncol = 2,
```

```

    filename = file.path(data_out_dir, "PCA.pdf"))
#> file saved to /home/datasience/repos/machine-learning-rsuite/export/PCA.pdf

```

Export plots to png files. If you specify a list of plots, then multiple png files will be automatically created to hold each plot.

```

ggexport(plotlist = list(scree.plot, ind.plot, var.plot),
        filename = file.path(data_out_dir, "PCA.png"))
#> [1] "/home/datasience/repos/machine-learning-rsuite/export/PCA%03d.png"
#> file saved to /home/datasience/repos/machine-learning-rsuite/export/PCA%03d.png

```

26.27 Export results to txt/csv files

All the outputs of the PCA (individuals/variables coordinates, contributions, etc) can be exported at once, into a TXT/CSV file, using the function write.infile() [in FactoMineR] package:

```

# Export into a TXT file
write.infile(res.pca, file.path(data_out_dir, "pca.txt"), sep = "\t")
# Export into a CSV file
write.infile(res.pca, file.path(data_out_dir, "pca.csv"), sep = ";")

```

26.28 Summary

In conclusion, we described how to perform and interpret principal component analysis (PCA). We computed PCA using the PCA() function [FactoMineR]. Next, we used the factoextra R package to produce ggplot2-based visualization of the PCA results.

There are other functions [packages] to compute PCA in R:

1. Using prcomp() [stats]

```

res.pca <- prcomp(iris[, -5], scale. = TRUE)

res.pca <- princomp(iris[, -5], cor = TRUE)

```

3. Using dudi.pca() [ade4]

```

library(ade4)
#>
#> Attaching package: 'ade4'
#> The following object is masked from 'package:FactoMineR':
#>
#>     reconst
res.pca <- dudi.pca(iris[, -5], scannf = FALSE, nf = 5)

```

4. Using epPCA() [ExPosition]

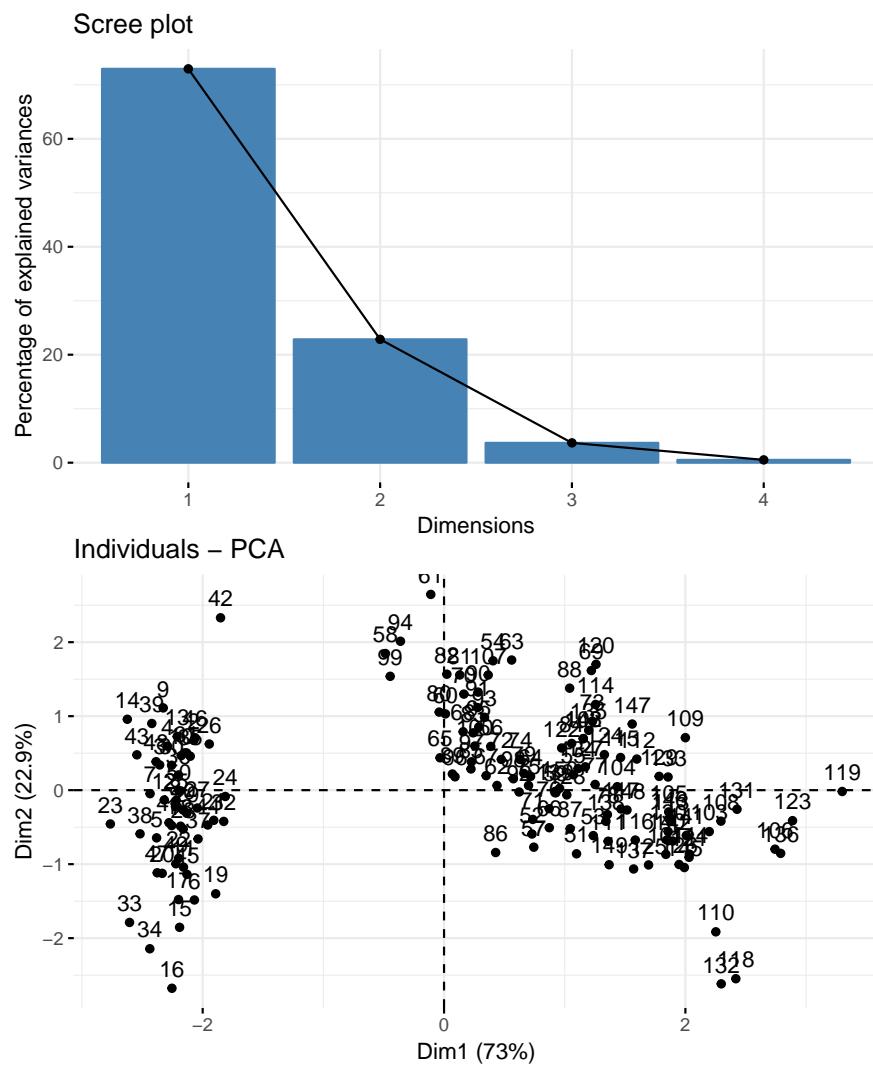
```

library(ExPosition)
#> Loading required package: prettyGraphs
res.pca <- epPCA(iris[, -5], graph = FALSE)

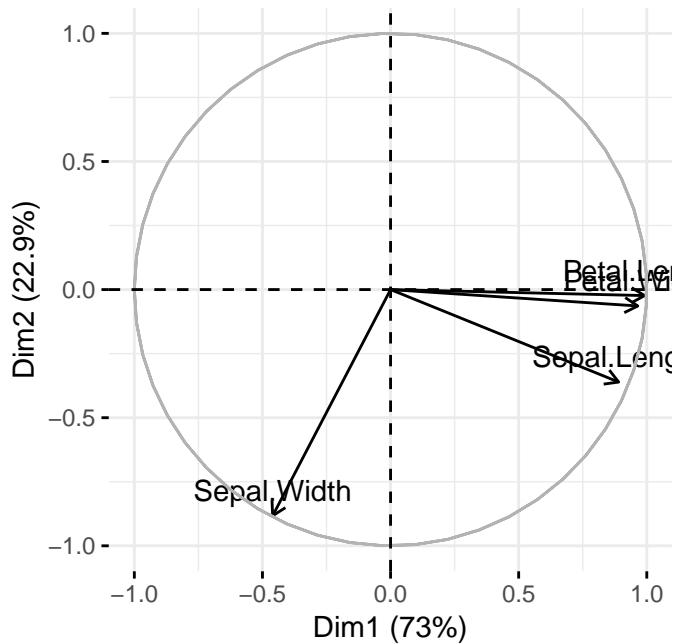
```

No matter what functions you decide to use, in the list above, the factoextra package can handle the output for creating beautiful plots similar to what we described in the previous sections for FactoMineR:

```
fviz_eig(res.pca)      # Scree plot  
fviz_pca_ind(res.pca) # Graph of individuals  
fviz_pca_var(res.pca) # Graph of variables
```



Variables – PCA



Chapter 27

Biplot of the Iris data set

```
# devtools::install_github("vqv/ggbiplot")
library(ggbiplot)
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures    rlang
#>   c.quosures    rlang
#>   print.quosures rlang
#> Loading required package: plyr
#> Loading required package: scales
#> Loading required package: grid

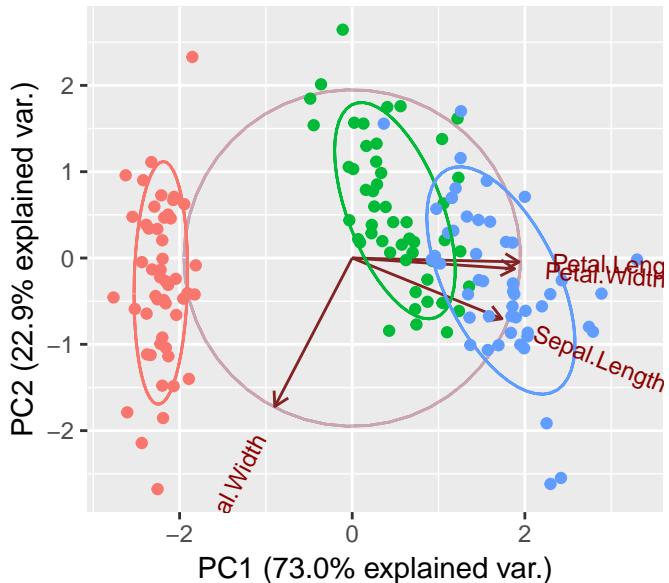
iris.pca <- prcomp(iris[, 1:4], center = TRUE, scale = TRUE)
print(iris.pca)
#> Standard deviations (1, ..., p=4):
#> [1] 1.708 0.956 0.383 0.144
#>
#> Rotation (n x k) = (4 x 4):
#>          PC1     PC2     PC3     PC4
#> Sepal.Length  0.521 -0.3774  0.720  0.261
#> Sepal.Width   -0.269 -0.9233 -0.244 -0.124
#> Petal.Length   0.580 -0.0245 -0.142 -0.801
#> Petal.Width    0.565 -0.0669 -0.634  0.524

summary(iris.pca)
#> Importance of components:
#>             PC1     PC2     PC3     PC4
#> Standard deviation  1.71  0.956  0.3831 0.14393
#> Proportion of Variance 0.73  0.229  0.0367 0.00518
#> Cumulative Proportion 0.73  0.958  0.9948 1.00000

g <- ggbiplot(iris.pca,
               obs.scale = 1,
               var.scale = 1,
               groups = iris$Species,
               ellipse = TRUE,
               circle = TRUE) +
  scale_color_discrete(name = "") +
```

```
theme(legend.direction = "horizontal", legend.position = "top")
print(g)
```





The PC1 axis explains 0.730 of the variance, while the PC2 axis explains 0.229 of the variance.

27.1 Iris: underlying principal components

```
# Run PCA here with prcomp()
iris.pca <- prcomp(iris[, 1:4], center = TRUE, scale = TRUE)

print(iris.pca)
#> Standard deviations (1, ..., p=4):
#> [1] 1.708 0.956 0.383 0.144
#>
#> # Rotation (n x k) = (4 x 4):
#>          PC1        PC2        PC3        PC4
#> Sepal.Length  0.521 -0.3774  0.720  0.261
#> Sepal.Width   -0.269 -0.9233 -0.244 -0.124
#> Petal.Length   0.580 -0.0245 -0.142 -0.801
#> Petal.Width    0.565 -0.0669 -0.634  0.524

# Now, compute the new dataset aligned to the PCs by
# using the predict() function .
df.new <- predict(iris.pca, iris[, 1:4])
head(df.new)
#>          PC1        PC2        PC3        PC4
#> [1,] -2.26 -0.478  0.1273  0.02409
#> [2,] -2.07  0.672  0.2338  0.10266
#> [3,] -2.36  0.341 -0.0441  0.02828
```

```

#> [4,] -2.29  0.595 -0.0910 -0.06574
#> [5,] -2.38 -0.645 -0.0157 -0.03580
#> [6,] -2.07 -1.484 -0.0269  0.00659

# Show the PCA model's sdev values are the square root
# of the projected variances, which are along the diagonal
# of the covariance matrix of the projected data.
iris.pca$sdev^2
#> [1] 2.9185 0.9140 0.1468 0.0207

# # Compute covariance matrix for new dataset.
# Recall that the standard deviation is the square root of the variance.
round(cov(df.new), 5)
#>      PC1   PC2   PC3   PC4
#> PC1 2.92 0.000 0.000 0.0000
#> PC2 0.00 0.914 0.000 0.0000
#> PC3 0.00 0.000 0.147 0.0000
#> PC4 0.00 0.000 0.000 0.0207

```

27.2 Iris. Compute the eigenvectors and eigenvalues

```

# Scale and center the data.
df.scaled <- scale(iris[, 1:4], center = TRUE, scale = TRUE)

# Compute the covariance matrix.
cov.df.scaled <- cov(df.scaled)

# Compute the eigenvectors and eigen values.
# Each eigenvector (column) is a principal component.
# Each eigenvalue is the variance explained by the
# associated eigenvector.
eigenInformation <- eigen(cov.df.scaled)

print(eigenInformation)
#> eigen() decomposition
#> $values
#> [1] 2.9185 0.9140 0.1468 0.0207
#>
#> $vectors
#>      [,1]     [,2]     [,3]     [,4]
#> [1,] 0.521 -0.3774  0.720  0.261
#> [2,] -0.269 -0.9233 -0.244 -0.124
#> [3,]  0.580 -0.0245 -0.142 -0.801
#> [4,]  0.565 -0.0669 -0.634  0.524

# Now, compute the new dataset aligned to the PCs by
# multiplying the eigenvector and data matrices.

# Create transposes in preparation for matrix multiplication
eigenvectors.t <- t(eigenInformation$vectors)      # 4x4
df.scaled.t <- t(df.scaled)      # 4x150

```

```
# Perform matrix multiplication.
df.new <- eigenvectors.t %*% df.scaled.t    # 4x150

# Create new data frame. First take transpose and
# then add column names.
df.new.t <- t(df.new)      # 150x4
colnames(df.new.t) <- c("PC1", "PC2", "PC3", "PC4")

head(df.new.t)
#>      PC1     PC2     PC3     PC4
#> [1,] -2.26 -0.478  0.1273  0.02409
#> [2,] -2.07  0.672  0.2338  0.10266
#> [3,] -2.36  0.341 -0.0441  0.02828
#> [4,] -2.29  0.595 -0.0910 -0.06574
#> [5,] -2.38 -0.645 -0.0157 -0.03580
#> [6,] -2.07 -1.484 -0.0269  0.00659

# Compute covariance matrix for new dataset
round(cov(df.new.t), 5)
#>      PC1     PC2     PC3     PC4
#> PC1  2.92  0.000  0.000  0.0000
#> PC2  0.00  0.914  0.000  0.0000
#> PC3  0.00  0.000  0.147  0.0000
#> PC4  0.00  0.000  0.000  0.0207
```

Chapter 28

Logistic Regression. Diabetes

28.1 Introduction

Source: <https://github.com/AntoineGuillot2/Logistic-Regression-R/blob/master/script.R> Source: <http://enhancedatascience.com/2017/04/26/r-basics-logistic-regression-with-r/> Data: <https://www.kaggle.com/uciml/pima-indians-diabetes-database>

The goal of logistic regression is to predict whether an outcome will be positive (aka 1) or negative (i.e: 0). Some real life example could be:

- Will Emmanuel Macron win the French Presidential election or will he lose?
- Does Mr.X has this illness or not?
- Will this visitor click on my link or not?

So, logistic regression can be used in a lot of binary classification cases and will often be run before more advanced methods. For this tutorial, we will use the diabetes detection dataset from Kaggle.

This dataset contains data from Pima Indians Women such as the number of pregnancies, the blood pressure, the skin thickness, ... the goal of the tutorial is to be able to detect diabetes using only these measures.

28.2 Exploring the data

As usual, first, let's take a look at our data. You can download the data here then please put the file diabetes.csv in your working directory. With the summary function, we can easily summarise the different variables:

```
library(ggplot2)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures    rlang
#>   c.quosures    rlang
#>   print.quosures rlang
library(data.table)

DiabetesData <- data.table(read.csv(file.path(data_raw_dir, 'diabetes.csv')))

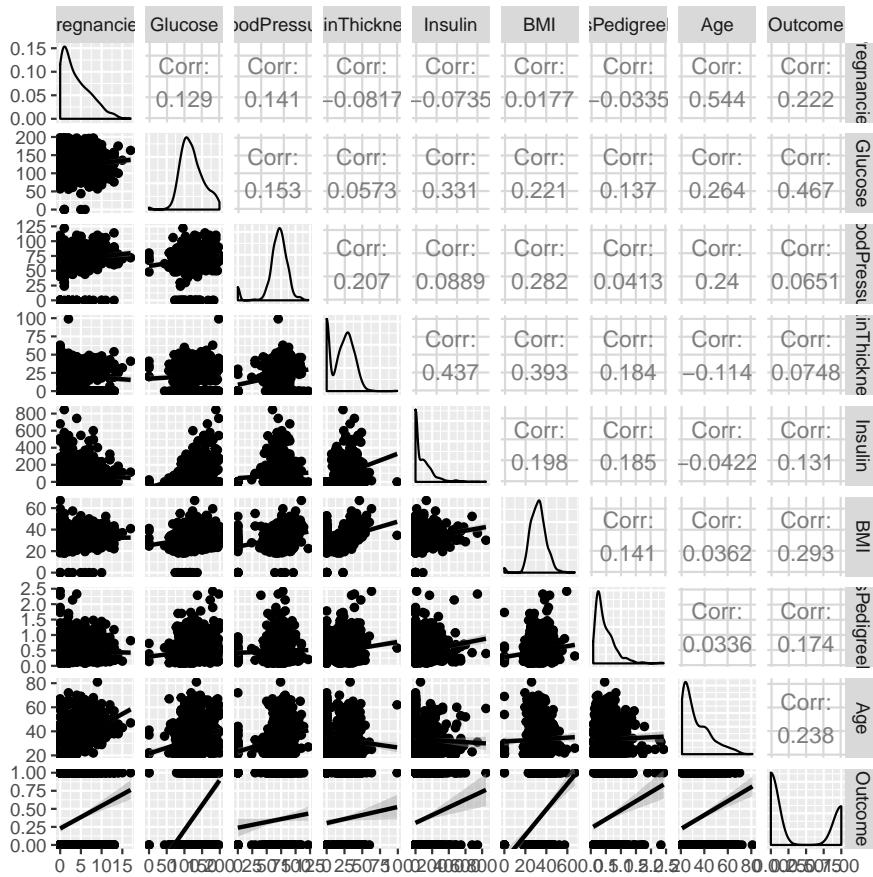
# Quick data summary
summary(DiabetesData)
#>   Pregnancies      Glucose      BloodPressure      SkinThickness
```

```
#> Min. : 0.00 Min. : 0 Min. : 0.0 Min. : 0.0
#> 1st Qu.: 1.00 1st Qu.: 99 1st Qu.: 62.0 1st Qu.: 0.0
#> Median : 3.00 Median :117 Median : 72.0 Median :23.0
#> Mean : 3.85 Mean :121 Mean : 69.1 Mean :20.5
#> 3rd Qu.: 6.00 3rd Qu.:140 3rd Qu.: 80.0 3rd Qu.:32.0
#> Max. :17.00 Max. :199 Max. :122.0 Max. :99.0
#> Insulin BMI DiabetesPedigreeFunction Age
#> Min. : 0 Min. : 0.0 Min. :0.078 Min. :21.0
#> 1st Qu.: 0 1st Qu.:27.3 1st Qu.:0.244 1st Qu.:24.0
#> Median : 30 Median :32.0 Median :0.372 Median :29.0
#> Mean : 80 Mean :32.0 Mean :0.472 Mean :33.2
#> 3rd Qu.:127 3rd Qu.:36.6 3rd Qu.:0.626 3rd Qu.:41.0
#> Max. :846 Max. :67.1 Max. :2.420 Max. :81.0
#> Outcome
#> Min. :0.000
#> 1st Qu.:0.000
#> Median :0.000
#> Mean :0.349
#> 3rd Qu.:1.000
#> Max. :1.000
```

The mean of the outcome is 0.35 which shows an imbalance between the classes. However, the imbalance should not be too strong to be a problem.

To understand the relationship between variables, a Scatter Plot Matrix will be used. To plot it, the GGally package was used.

```
# Scatter plot matrix
library(GGally)
#> Registered S3 method overwritten by 'GGally':
#>   method from
#>   +.gg ggplot2
ggpairs(DiabetesData, lower = list(continuous='smooth'))
```



The correlations between explanatory variables do not seem too strong. Hence the model is not likely to suffer from multicollinearity. All explanatory variable are correlated with the Outcome. At first sight, glucose rate is the most important factor to detect the outcome.

28.3 Logistic regression with R

After variable exploration, a first model can be fitted using the `glm` function. With `stargazer`, it is easy to get nice output in ASCII or even Latex.

```
# first model: all features
glm1 = glm(Outcome ~ .,
            DiabetesData,
            family = binomial(link="logit"))

summary(glm1)
#>
#> Call:
#> glm(formula = Outcome ~ ., family = binomial(link = "logit"),
#>      data = DiabetesData)
#>
#> Deviance Residuals:
#>      Min        1Q     Median        3Q       Max
#> -2.557   -0.727   -0.416    0.727    2.930
#>
#> Coefficients:
```

```

#>                               Estimate Std. Error z value Pr(>|z|)
#> (Intercept)                 -8.404696  0.716636 -11.73 < 2e-16 ***
#> Pregnancies                  0.123182  0.032078   3.84  0.00012 ***
#> Glucose                      0.035164  0.003709   9.48 < 2e-16 ***
#> BloodPressure                -0.013296  0.005234  -2.54  0.01107 *
#> SkinThickness                0.000619  0.006899   0.09  0.92852
#> Insulin                      -0.001192  0.000901  -1.32  0.18607
#> BMI                          0.089701  0.015088   5.95  2.8e-09 ***
#> DiabetesPedigreeFunction    0.945180  0.299147   3.16  0.00158 **
#> Age                          0.014869  0.009335   1.59  0.11119
#> ---
#> Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> (Dispersion parameter for binomial family taken to be 1)
#>
#> Null deviance: 993.48 on 767 degrees of freedom
#> Residual deviance: 723.45 on 759 degrees of freedom
#> AIC: 741.4
#>
#> Number of Fisher Scoring iterations: 5
require(stargazer)
#> Loading required package: stargazer
#>
#> Please cite as:
#> Hlavac, Marek (2018). stargazer: Well-Formatted Regression and Summary Statistics Tables.
#> R package version 5.2.2. https://CRAN.R-project.org/package=stargazer
stargazer(glm1, type='text')
#>
#> =====
#>                               Dependent variable:
#>                               -----
#>                               Outcome
#> -----
#> Pregnancies                  0.123***  

#>                               (0.032)  

#>
#> Glucose                      0.035***  

#>                               (0.004)  

#>
#> BloodPressure                -0.013**  

#>                               (0.005)  

#>
#> SkinThickness                0.001  

#>                               (0.007)  

#>
#> Insulin                      -0.001  

#>                               (0.001)  

#>
#> BMI                          0.090***  

#>                               (0.015)  

#>
#> DiabetesPedigreeFunction    0.945***  

#>                               (0.299)

```

```
#>
#> Age          0.015
#>             (0.009)
#>
#> Constant     -8.400*** 
#>             (0.717)
#>
#> -----
#> Observations 768
#> Log Likelihood -362.000
#> Akaike Inf. Crit. 741.000
#> =====
#> Note: *p<0.1; **p<0.05; ***p<0.01
```

The overall model is significant. As expected the glucose rate has the lowest p-value of all the variables. However, Age, Insulin and Skin Thickness are not good predictors of Diabetes.

28.4 A second model

Since some variables are not significant, removing them is a good way to improve model robustness. In the second model, SkinThickness, Insulin, and Age are removed.

```
# second model: selected features
glm2 = glm(Outcome~.,
            data = DiabetesData[,c(1:3,6:7,9), with=F],
            family = binomial(link="logit"))

summary(glm2)
#>
#> Call:
#> glm(formula = Outcome ~ ., family = binomial(link = "logit"),
#>       data = DiabetesData[, c(1:3, 6:7, 9), with = F])
#>
#> Deviance Residuals:
#>    Min      1Q  Median      3Q      Max
#> -2.793  -0.736  -0.419   0.725   2.955
#>
#> Coefficients:
#>              Estimate Std. Error z value Pr(>|z|)
#> (Intercept) -7.95495   0.67582 -11.77 < 2e-16 ***
#> Pregnancies  0.15349   0.02784   5.51 3.5e-08 ***
#> Glucose      0.03466   0.00339  10.21 < 2e-16 ***
#> BloodPressure -0.01201  0.00503  -2.39  0.017 *
#> BMI          0.08483   0.01412   6.01 1.9e-09 ***
#> DiabetesPedigreeFunction 0.91063   0.29403   3.10  0.002 **
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> (Dispersion parameter for binomial family taken to be 1)
#>
#> Null deviance: 993.48 on 767 degrees of freedom
#> Residual deviance: 728.56 on 762 degrees of freedom
#> AIC: 740.6
```

```
#>
#> Number of Fisher Scoring iterations: 5
```

28.5 Classification rate and confusion matrix

Now that we have our model, let's access its performance.

```
# Correctly classified observations
mean((glm2$fitted.values>0.5)==DiabetesData$Outcome)
#> [1] 0.775
```

Around 77.4% of all observations are correctly classified. Due to class imbalance, we need to go further with a confusion matrix.

```
## Confusion matrix count
RP=sum((glm2$fitted.values>=0.5)==DiabetesData$Outcome & DiabetesData$Outcome==1)
FP=sum((glm2$fitted.values>=0.5) != DiabetesData$Outcome & DiabetesData$Outcome==0)
RN=sum((glm2$fitted.values>=0.5)==DiabetesData$Outcome & DiabetesData$Outcome==0)
FN=sum((glm2$fitted.values>=0.5) != DiabetesData$Outcome & DiabetesData$Outcome==1)
confMat<-matrix(c(RP,FP,FN,RN),ncol = 2)
colnames(confMat)<-c("Pred Diabetes", "Pred no diabetes")
rownames(confMat)<-c("Real Diabetes", "Real no diabetes")
confMat
#>
#>             Pred Diabetes Pred no diabetes
#> Real Diabetes           154            114
#> Real no diabetes         59            441
```

The model is good to detect people who do not have diabetes. However, its performance on ill people is not great (only 154 out of 268 have been correctly classified).

You can also get the percentage of Real/False Positive/Negative:

```
# Confusion matrix proportion
RPR=RP/sum(DiabetesData$Outcome==1)*100
FNR=FN/sum(DiabetesData$Outcome==1)*100
FPR=FP/sum(DiabetesData$Outcome==0)*100
RNR=RN/sum(DiabetesData$Outcome==0)*100
confMat<-matrix(c(RPR,FPR,FNR,RNR),ncol = 2)
colnames(confMat)<-c("Pred Diabetes", "Pred no diabetes")
rownames(confMat)<-c("Real Diabetes", "Real no diabetes")
confMat
#>
#>             Pred Diabetes Pred no diabetes
#> Real Diabetes          57.5            42.5
#> Real no diabetes        11.8            88.2
```

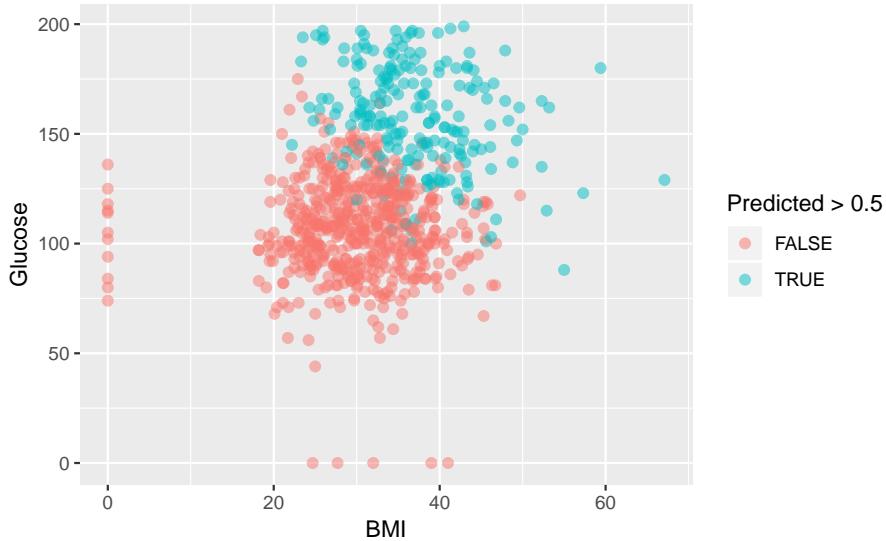
And here is the matrix, 57.5% of people with diabetes are correctly classified. A way to improve the false negative rate would lower the detection threshold. However, as a consequence, the false positive rate would increase.

28.6 Plots and decision boundaries

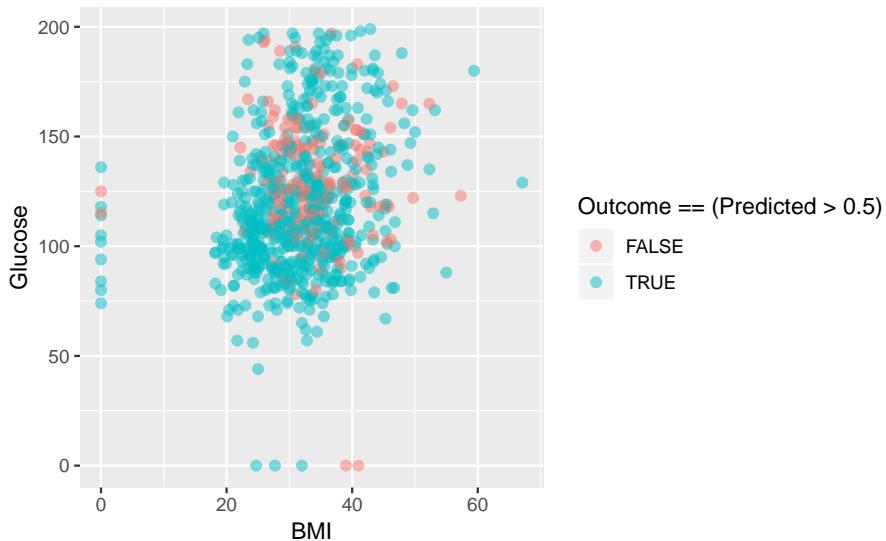
The two strongest predictors of the outcome are Glucose rate and BMI. High glucose rate and high BMI are strong indicators of Diabetes.

```
# Plot and decision boundaries
DiabetesData$Predicted <- glm2$fitted.values

ggplot(DiabetesData, aes(x = BMI, y = Glucose, color = Predicted > 0.5)) +
  geom_point(size=2, alpha=0.5)
```



```
ggplot(DiabetesData, aes(x=BMI, y = Glucose, color=Outcome == (Predicted > 0.5))) +
  geom_point(size=2, alpha=0.5)
```



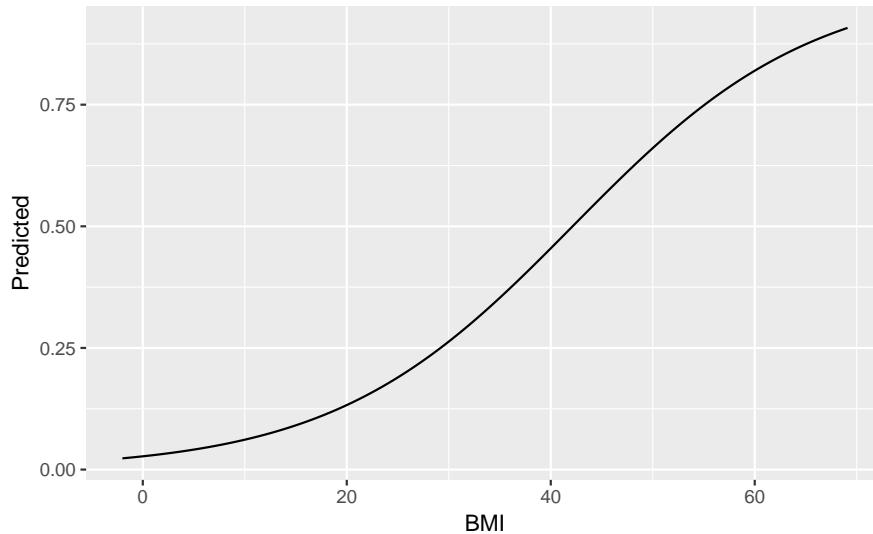
We can also plot both BMI and glucose against the outcomes, the other variables are taken at their mean level.

```
range(DiabetesData$BMI)
#> [1] 0.0 67.1

# BMI vs predicted
BMI_plot = data.frame(BMI = ((min(DiabetesData$BMI)-2)*100):
                      (max(DiabetesData$BMI+2)*100))/100,
                      Glucose = mean(DiabetesData$Glucose),
                      Pregnancies = mean(DiabetesData$Pregnancies),
```

```
BloodPressure = mean(DiabetesData$BloodPressure),
DiabetesPedigreeFunction = mean(DiabetesData$DiabetesPedigreeFunction))

BMI_plot$Predicted = predict(glm2, newdata = BMI_plot, type = 'response')
ggplot(BMI_plot, aes(x = BMI, y = Predicted)) +
  geom_line()
```

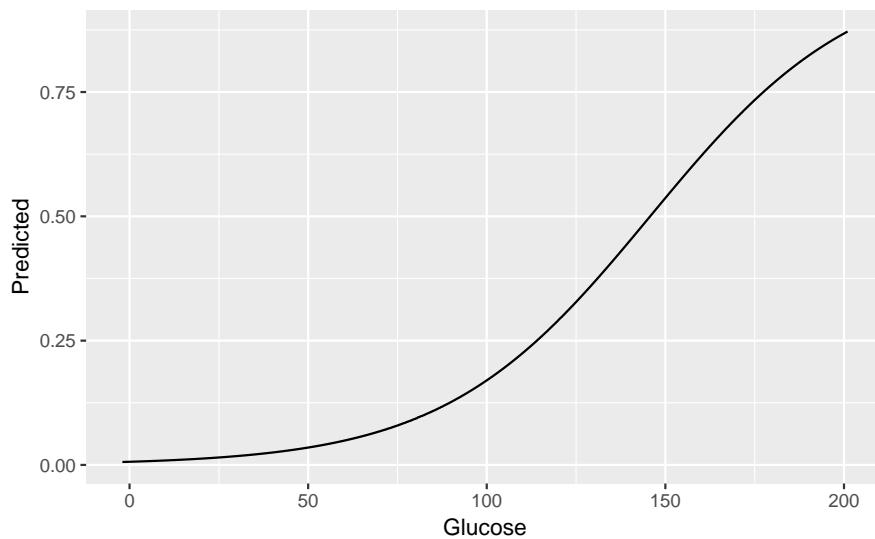


```
range(BMI_plot$BMI)
#> [1] -2.0 69.1

range(DiabetesData$Glucose)
#> [1] 0 199

# Glucose vs predicted
Glucose_plot=data.frame(Glucose =
  ((min(DiabetesData$Glucose)-2)*100):
  ((max(DiabetesData$Glucose)+2)*100))/100,
  BMI=mean(DiabetesData$BMI),
  Pregnancies=mean(DiabetesData$Pregnancies),
  BloodPressure=mean(DiabetesData$BloodPressure),
  DiabetesPedigreeFunction=mean(DiabetesData$DiabetesPedigreeFunction))

Glucose_plot$Predicted = predict(glm2, newdata = Glucose_plot, type = 'response')
ggplot(Glucose_plot, aes(x = Glucose, y = Predicted)) +
  geom_line()
```



```
range(Glucose_plot$Glucose)
#> [1] -2 201
```


Chapter 29

Sensitivity analysis for neural networks

29.1 Introduction

<https://beckmw.wordpress.com/tag/nnet/>

I've made quite a few blog posts about neural networks and some of the diagnostic tools that can be used to 'demystify' the information contained in these models. Frankly, I'm kind of sick of writing about neural networks but I wanted to share one last tool I've implemented in R. I'm a strong believer that supervised neural networks can be used for much more than prediction, as is the common assumption by most researchers. I hope that my collection of posts, including this one, has shown the versatility of these models to develop inference into causation. To date, I've authored posts on visualizing neural networks, animating neural networks, and determining importance of model inputs. This post will describe a function for a sensitivity analysis of a neural network. Specifically, I will describe an approach to evaluate the form of the relationship of a response variable with the explanatory variables used in the model.

The general goal of a sensitivity analysis is similar to evaluating relative importance of explanatory variables, with a few important distinctions. For both analyses, we are interested in the relationships between explanatory and response variables as described by the model in the hope that the neural network has explained some real-world phenomenon. Using Garson's algorithm,¹ we can get an idea of the magnitude and sign of the relationship between variables relative to each other. Conversely, the sensitivity analysis allows us to obtain information about the form of the relationship between variables rather than a categorical description, such as variable x is positively and strongly related to y. For example, how does a response variable change in relation to increasing or decreasing values of a given explanatory variable? Is it a linear response, non-linear, uni-modal, no response, etc.? Furthermore, how does the form of the response change given values of the other explanatory variables in the model? We might expect that the relationship between a response and explanatory variable might differ given the context of the other explanatory variables (i.e., an interaction may be present). The sensitivity analysis can provide this information.

As with most of my posts, I've created the sensitivity analysis function using ideas from other people that are much more clever than me. I've simply converted these ideas into a useful form in R. Ultimate credit for the sensitivity analysis goes to Sovan Lek (and colleagues), who developed the approach in the mid-1990s. The 'Lek-profile method' is described briefly in Lek et al. 1996² and in more detail in Gevrey et al. 2003.³ I'll provide a brief summary here since the method is pretty simple. In fact, the profile method can be extended to any statistical model and is not specific to neural networks, although it is one of few methods used to evaluate the latter. For any statistical model where multiple response variables are related to multiple explanatory variables, we choose one response and one explanatory variable. We obtain predictions of the response variable across the range of values for the given explanatory variable. All other explanatory variables

are held constant at a given set of respective values (e.g., minimum, 20th percentile, maximum). The final product is a set of response curves for one response variable across the range of values for one explanatory variable, while holding all other explanatory variables constant. This is implemented in R by creating a matrix of values for explanatory variables where the number of rows is the number of observations and the number of columns is the number of explanatory variables. All explanatory variables are held at their mean (or other constant value) while the variable of interest is sequenced from its minimum to maximum value across the range of observations. This matrix (actually a data frame) is then used to predict values of the response variable from a fitted model object. This is repeated for different variables.

I'll illustrate the function using simulated data, as I've done in previous posts. The exception here is that I'll be using two response variables instead of one. The two response variables are linear combinations of eight explanatory variables, with random error components taken from a normal distribution. The relationships between the variables are determined by the arbitrary set of parameters (`parms1` and `parms2`). The explanatory variables are partially correlated and taken from a multivariate normal distribution.

```

require(clusterGeneration)
#> Loading required package: clusterGeneration
#> Loading required package: MASS
require(nnet)
#> Loading required package: nnet

#define number of variables and observations
set.seed(2)
num.vars<-8
num.obs<-10000

#define correlation matrix for explanatory variables
#define actual parameter values
cov.mat<-genPositiveDefMat(num.vars,covMethod=c("unifcorrmat"))$Sigma
rand.vars<-mvrnorm(num.obs,rep(0,num.vars),Sigma=cov.mat)
parms1<-runif(num.vars,-10,10)
y1<-rand.vars %*% matrix(parms1) + rnorm(num.obs, sd=20)
parms2<-runif(num.vars,-10,10)
y2<-rand.vars %*% matrix(parms2) + rnorm(num.obs, sd=20)

#prep data and create neural network
rand.vars<-data.frame(rand.vars)
resp<-apply(cbind(y1,y2),2, function(y) (y-min(y))/(max(y)-min(y)))
resp<-data.frame(resp)
names(resp)<-c('Y1','Y2')

mod1 <- nnet(rand.vars,resp,size=8,linout=T)
#> # weights:  90
#> initial value 30121.205794
#> iter  10 value 130.537462
#> iter  20 value 57.187090
#> iter  30 value 47.285919
#> iter  40 value 42.778564
#> iter  50 value 39.837784
#> iter  60 value 36.694632
#> iter  70 value 35.140948
#> iter  80 value 34.268819
#> iter  90 value 33.772282
#> iter 100 value 33.472654
#> final  value 33.472654

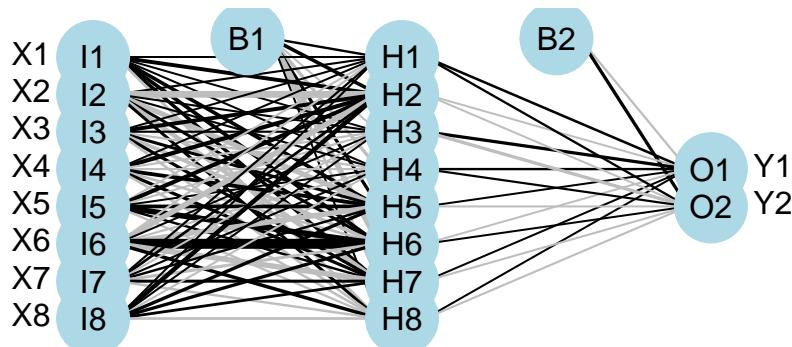
```

```
#> stopped after 100 iterations

#import the function from Github
library(devtools)

# source_url('https://gist.githubusercontent.com/fawda123/7471137/raw/466c1474d0a505ff044412703516c34f1
source("nnet_plot_update.r")

#plot each model
plot.nnet(mod1)
#> Loading required package: scales
#> Loading required package: reshape
```



29.2 The Lek profile function

We've created a neural network that hopefully describes the relationship of two response variables with eight explanatory variables. The sensitivity analysis lets us visualize these relationships. The Lek profile function can be used once we have a neural network model in our workspace. The function is imported and used as follows:

```
# source('https://gist.githubusercontent.com/fawda123/6860630/raw/b8bf4a6c88d6b392b1bfa6ef24759ae98f318
source("lek_fun.r")

lek.fun(mod1)
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures    rlang
#>   c.quosures    rlang
#>   print.quosures rlang
```

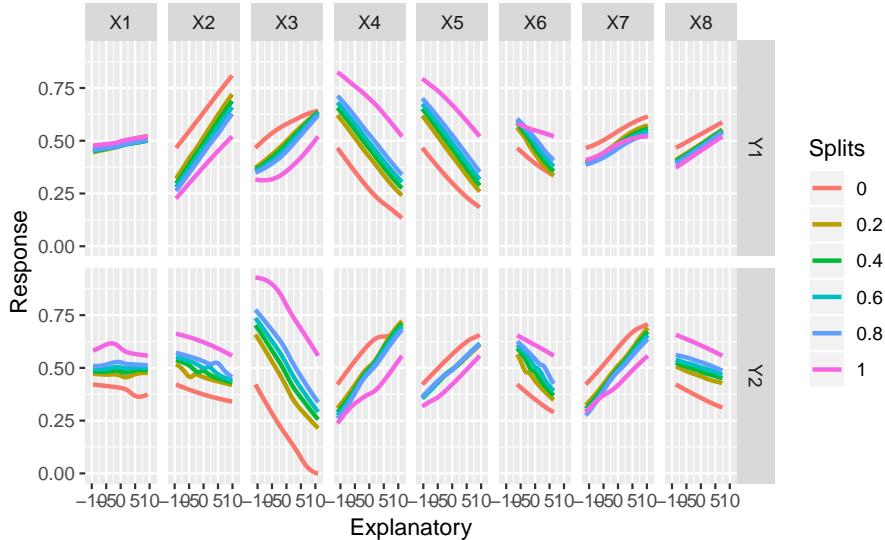


Fig: Sensitivity analysis of the two response variables in the neural network model to individual explanatory variables. Splits represent the quantile values at which the remaining explanatory variables were held constant. The function can be obtained here

By default, the function runs a sensitivity analysis for all variables. This creates a busy plot so we may want to look at specific variables of interest. Maybe we want to evaluate different quantile values as well. These options can be changed using the arguments.

```
lek.fun(mod1, var.sens=c('X2', 'X5'), split.vals=seq(0,1,by=0.05))
```

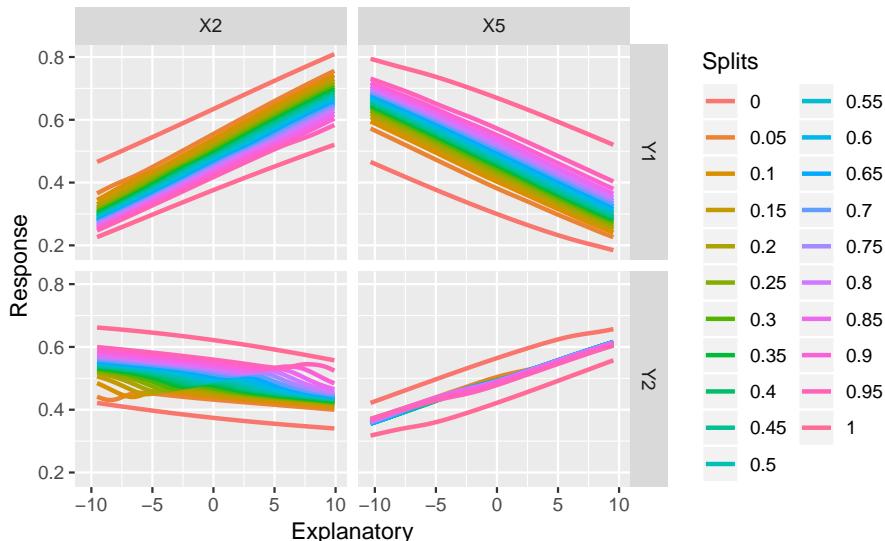
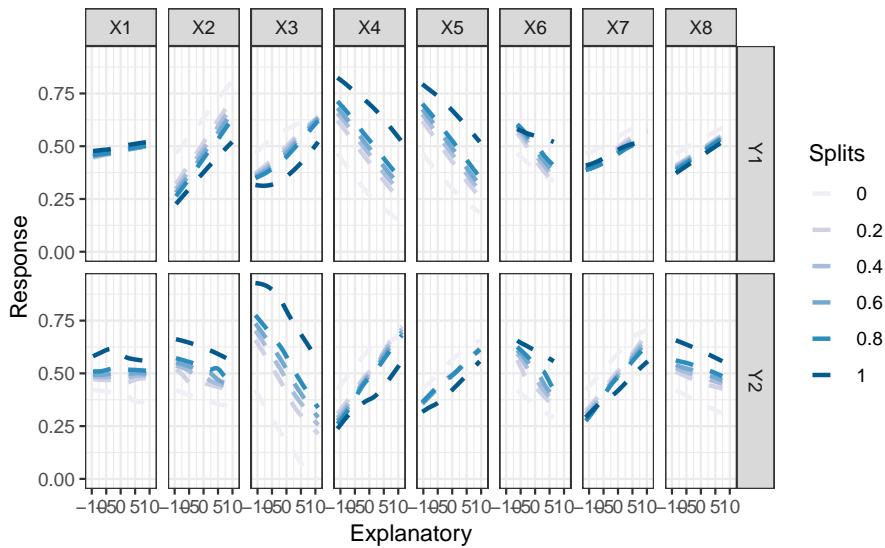


Fig: Sensitivity analysis of the two response variables in relation to explanatory variables X_2 and X_5 and different quantile values for the remaining variables.

The function also returns a ggplot2 object that can be further modified. You may prefer a different theme, color, or line type, for example.

```
p1<-lek.fun(mod1)
class(p1)
#> [1] "gg"      "ggplot"
# [1] "gg"      "ggplot"
```

```
p1 +
  theme_bw() +
  scale_colour_brewer(palette="PuBu") +
  scale_linetype_manual(values=rep('dashed',6)) +
  scale_size_manual(values=rep(1,6))
#> Scale for 'linetype' is already present. Adding another scale for
#> 'linetype', which will replace the existing scale.
#> Scale for 'size' is already present. Adding another scale for 'size',
#> which will replace the existing scale.
```



29.3 Getting a dataframe from lek

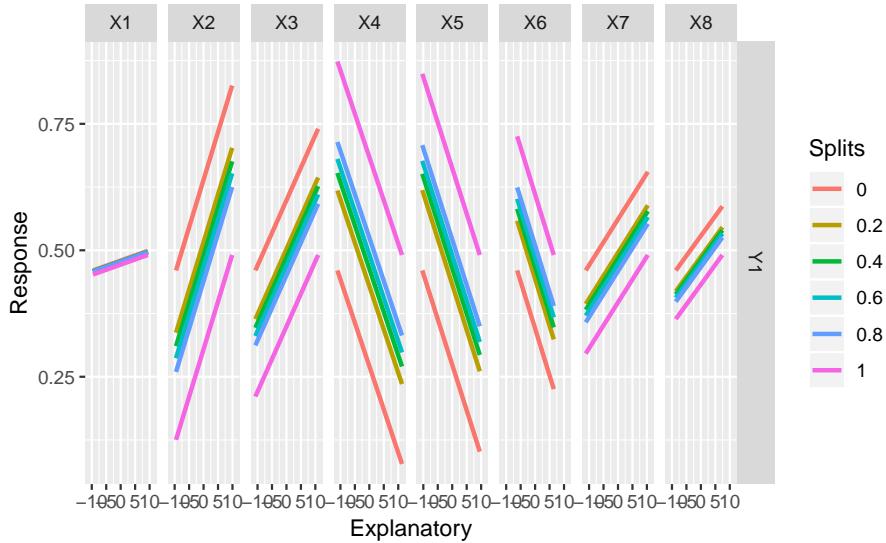
Finally, the actual values from the sensitivity analysis can be returned if you'd prefer that instead. The output is a data frame in long form that was created using `melt.list` from the `reshape` package for compatibility with `ggplot2`. The six columns indicate values for explanatory variables on the x-axes, names of the response variables, predicted values of the response variables, quantiles at which other explanatory variables were held constant, and names of the explanatory variables on the x-axes.

```
head(lek.fun(mod1, val.out = TRUE))
#>   Explanatory resp.name Response Splits exp.name
#> 1      -9.58      Y1    0.466     0      X1
#> 2      -9.39      Y1    0.466     0      X1
#> 3      -9.19      Y1    0.467     0      X1
#> 4      -9.00      Y1    0.467     0      X1
#> 5      -8.81      Y1    0.468     0      X1
#> 6      -8.62      Y1    0.468     0      X1
```

29.4 The lek function works with lm

I mentioned earlier that the function is not unique to neural networks and can work with other models created in R. I haven't done an extensive test of the function, but I'm fairly certain that it will work if the model object has a predict method (e.g., `predict.lm`). Here's an example using the function to evaluate a multiple linear regression for one of the response variables.

```
mod2 <- lm(Y1 ~ ., data = cbind(resp[, 'Y1', drop = F], rand.vars))
lek.fun(mod2)
```



This function has little relevance for conventional models like linear regression since a wealth of `diagnostic` tools are already available (e.g., effects plots, add/drop procedures, outlier tests, etc.). The application of the function to neural networks provides insight into the relationships described by the models, insights that to my knowledge, cannot be obtained using current tools in R. This post concludes my contribution of diagnostic tools for neural networks in R and I hope that they have been useful to some of you. I have spent the last year or so working with neural networks and my opinion of their utility is mixed. I see advantages in the use of highly flexible computer-based algorithms, although in most cases similar conclusions can be made using more conventional analyses. I suggest that neural networks only be used *if there is an extremely high sample size* and other methods have proven inconclusive. Feel free to voice your opinions or suggestions in the comments.

29.5 lek function works with RSNNS

```
require(clusterGeneration)
require(RSNNS)
#> Loading required package: RSNNS
#> Loading required package: Rcpp
require(devtools)

#define number of variables and observations
set.seed(2)
num.vars<-8
num.obs<-10000

#define correlation matrix for explanatory variables
#define actual parameter values
cov.mat <-genPositiveDefMat(num.vars,covMethod=c("unifcorrmat"))$Sigma
rand.vars <-mvrnorm(num.obs,rep(0,num.vars),Sigma=cov.mat)
parms1 <-runif(num.vars,-10,10)
y1 <-rand.vars %*% matrix(parms1) + rnorm(num.obs, sd=20)
parms2 <-runif(num.vars,-10,10)
```

```

y2 <- rand.vars %*% matrix(parms2) + rnorm(num.obs, sd=20)

#prep data and create neural network
rand.vars <- data.frame(rand.vars)
resp <- apply(cbind(y1,y2), 2, function(y) (y-min(y))/(max(y)-min(y)))
resp <- data.frame(resp)
names(resp)<-c('Y1','Y2')

tibble::as_tibble(rand.vars)
#> # A tibble: 10,000 x 8
#>   X1     X2     X3     X4     X5     X6     X7     X8
#>   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
#> 1  1.61   2.13   2.13   3.97  -1.34   2.00   3.11  -2.55
#> 2 -1.25   3.07  -0.325  1.61  -0.484  2.28   2.98  -1.71
#> 3 -3.17  -1.29  -1.77  -1.66  -0.549 -3.19   1.07   1.81
#> 4 -2.39   3.28  -3.42  -0.160 -1.52   2.67   7.05  -1.14
#> 5 -1.55  -0.181 -1.14   2.27  -1.68  -1.67   3.08   0.334
#> 6  0.0690 -1.54  -2.98   2.84   1.42   1.31   1.82   2.07
#> # ... with 9,994 more rows

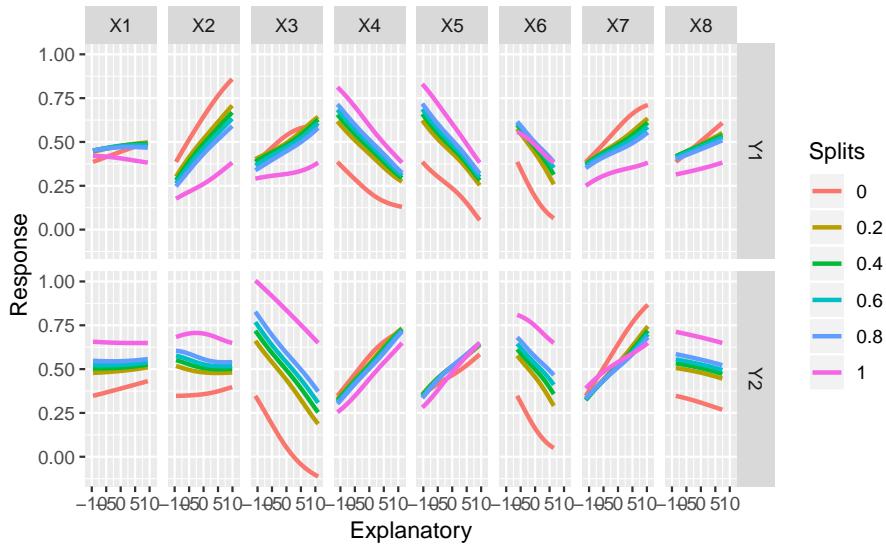
tibble::as_tibble(resp)
#> # A tibble: 10,000 x 2
#>   Y1     Y2
#>   <dbl> <dbl>
#> 1 0.461 0.500
#> 2 0.416 0.509
#> 3 0.534 0.675
#> 4 0.548 0.619
#> 5 0.519 0.659
#> 6 0.389 0.622
#> # ... with 9,994 more rows

# create neural network model
mod2 <- mlp(rand.vars, resp, size = 8, linOut = T)

#import sensitivity analysis function
source_url('https://gist.githubusercontent.com/fawda123/6860630/raw/b8bf4a6c88d6b392b1bfa6ef24759ae98f3'
#> SHA-1 hash of file is 4a2d33b94a08f46a94518207a4ae7cc412845222

#sensitivity analysis, note 'exp.in' argument
lek.fun(mod2, exp.in = rand.vars)

```



Chapter 30

References

- 1 Garson GD. 1991. Interpreting neural network connection weights. *Artificial Intelligence Expert.* 6:46–51.
- 2 Lek S, Delacoste M, Baran P, Dimopoulos I, Lauga J, Aulagnier S. 1996. Application of neural networks to modelling nonlinear relationships in Ecology. *Ecological Modelling.* 90:39-52.
- 3 Gevrey M, Dimopoulos I, Lek S. 2003. Review and comparison of methods to study the contribution of variables in artificial neural network models. *Ecological Modelling.* 160:249-264.

Chapter 31

What is .hat in regression output

<https://stats.stackexchange.com/a/256364/154908>

Q. The augment() function in the broom package for R creates a dataframe of predicted values from a regression model. Columns created include the fitted values, the standard error of the fit and Cook's distance. They also include something with which I'm not familiar and that is the column .hat.

```
library(broom)
data(mtcars)

m1 <- lm(mpg ~ wt, data = mtcars)

head(augment(m1))
#> # A tibble: 6 x 10
#>   .rownames   mpg     wt .fitted .se.fit .resid   .hat .sigma .cooksdi
#>   <chr>     <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>   <dbl>
#> 1 Mazda RX4  21     2.62    23.3    0.634  -2.28   0.0433  3.07  1.33e-2
#> 2 Mazda RX-4 21     2.88    21.9    0.571  -0.920  0.0352  3.09  1.72e-3
#> 3 Datsun 7-210 22.8  2.32    24.9    0.736  -2.09   0.0584  3.07  1.54e-2
#> 4 Hornet 4-Door 21.4  3.22    20.1    0.538   1.30   0.0313  3.09  3.02e-3
#> 5 Hornet S-300 18.7  3.44    18.9    0.553  -0.200  0.0329  3.10  7.60e-5
#> 6 Valiant     18.1  3.46    18.8    0.555  -0.693  0.0332  3.10  9.21e-4
#> # ... with 1 more variable: .std.resid <dbl>

# .hat vector
augment(m1)$hat
#> [1] 0.0433 0.0352 0.0584 0.0313 0.0329 0.0332 0.0354 0.0313 0.0314 0.0329
#> [11] 0.0329 0.0558 0.0401 0.0419 0.1705 0.1953 0.1838 0.0661 0.1177 0.0956
#> [21] 0.0503 0.0343 0.0328 0.0443 0.0445 0.0866 0.0704 0.1291 0.0313 0.0380
#> [31] 0.0354 0.0377
```

Can anyone explain what this value is, and is it different between linear regression and logistic regression?

A. Those would be the diagonal elements of the hat-matrix which describe the leverage each point has on its fitted values.

If one fits:

$$\vec{Y} = \mathbf{X}\vec{\beta} + \vec{\epsilon}$$

then:

$$\mathbf{H} = \mathbf{X}(\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T$$

In this example:

$$\begin{pmatrix} Y_1 \\ \vdots \\ Y_{32} \end{pmatrix} = \begin{pmatrix} 1 & 2.620 \\ \vdots & \\ 1 & 2.780 \end{pmatrix} \cdot \begin{pmatrix} \beta_0 \\ \beta_1 \end{pmatrix} + \begin{pmatrix} \epsilon_1 \\ \vdots \\ \epsilon_{32} \end{pmatrix}$$

Then calculating this \mathbf{H} matrix results in:

```
library(MASS)

wt <- mtcars[, 6]

X <- matrix(cbind(rep(1, length(wt)), wt), ncol=2)

H <- X %*% ginv(t(X) %*% X) %*% t(X)
```

Where this last matrix is a 32×32 matrix and contains these hat values on the diagonal.

```
X                      32x2
t(X)                   2x32
X %*% t(X)            32x32
t(X) %*% X             2x2
ginv(t(X) %*% X)      2x2
ginv(t(X) %*% X) %*% t(X) 2x32
X %*% ginv(t(X) %*% X) 32x2

dim(ginv(t(X) %*% X) %*% t(X))
#> [1] 2 32

x1 <- X %*% ginv(t(X) %*% X)
dim(x1)
#> [1] 32 2
dim(x1 %*% t(X))
#> [1] 32 32

x2 <- ginv(t(X) %*% X) %*% t(X)
dim(x2)
#> [1] 2 32
dim(X %*% x2)
#> [1] 32 32

# this last matrix is a 32x32 matrix and contains these hat values on the diagonal.
diag(H)
#> [1] 0.0433 0.0352 0.0584 0.0313 0.0329 0.0332 0.0354 0.0313 0.0314 0.0329
#> [11] 0.0329 0.0558 0.0401 0.0419 0.1705 0.1953 0.1838 0.0661 0.1177 0.0956
#> [21] 0.0503 0.0343 0.0328 0.0443 0.0445 0.0866 0.0704 0.1291 0.0313 0.0380
#> [31] 0.0354 0.0377
```

Chapter 32

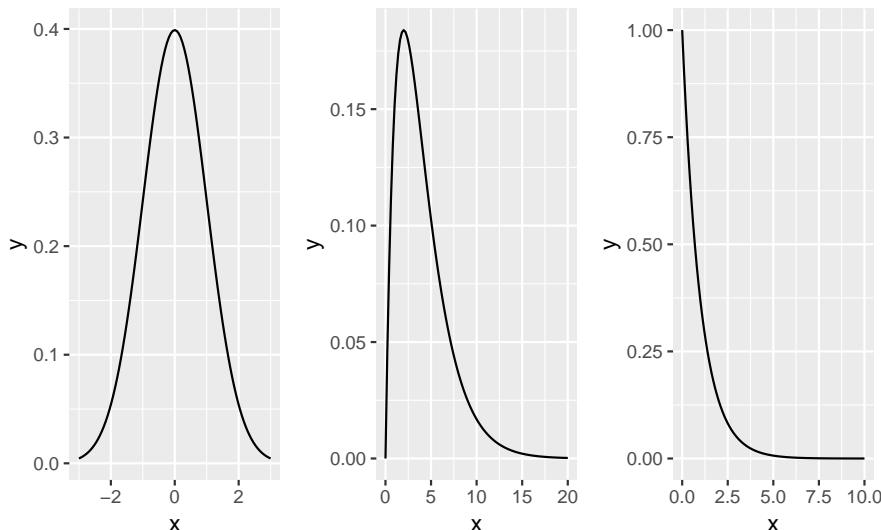
Q-Q normal to compare data to distributions

32.1 Introduction

<https://mgimond.github.io/ES218/Week06a.html>

Thus far, we have used the quantile-quantile plots to compare the distributions between two empirical (i.e. observational) datasets. This is sometimes referred to as an empirical **Q-Q** plot. We can also use the q-q plot to compare an *empirical* observation to a *theoretical* observation (i.e. one defined mathematically). Such a plot is usually referred to as a **theoretical Q-Q plot**. Examples of popular theoretical observations are the normal distribution (aka the Gaussian distribution), the **chi-square** distribution, and the exponential distribution just to name a few.

```
#> Registered S3 methods overwritten by 'ggplot2':  
#>   method      from  
#>   [.quosures    rlang  
#>   c.quosures    rlang  
#>   print.quosures rlang
```



32.2 Why we want to compare empirical vs theoretical distributions

There are many reasons we might want to compare empirical data to theoretical distributions:

- A theoretical distribution is easy to parameterize. For example, if the shape of the distribution of a batch of numbers can be approximated by a normal distribution we can reduce the complexity of our data to just two values: the mean and the standard deviation.
- If data can be approximated by certain theoretical distributions, then many mainstream statistical procedures can be applied to the data.
- In inferential statistics, knowing that a sample was derived from a population whose distribution follows a theoretical distribution allows us to derive certain properties of the population from the sample. For example, if we know that a sample comes from a normally distributed population, we can define confidence intervals for the sample mean using a **t-distribution**.
- Modeling the distribution of the observed data can provide insight into the underlying process that generated the data.

But very few empirical datasets follow any theoretical distributions exactly. So the questions usually ends up being “how well does theoretical distribution X fit my data?”

The theoretical quantile-quantile plot is a tool to explore how a batch of numbers deviates from a theoretical distribution and to visually assess whether the difference is significant for the purpose of the analysis. In the following examples, we will compare empirical data to the normal distribution using the normal quantile-quantile plot.

32.3 The normal q-q plot

The normal q-q plot is just a special case of the empirical q-q plot we’ve explored so far; the difference being that we assign the normal distribution quantiles to the x-axis.

32.3.1 Drawing a normal q-q plot from scratch

In the following example, we’ll compare the Alto 1 group to a normal distribution. First, we’ll extract the Alto 1 height values and save them as an atomic vector object using dplyr’s piping operations.

However, dplyr’s operations will return a dataframe—even if a single column is selected. To force the output to an atomic vector, we’ll pipe the subset to `pull(height)` which will extract the height column into a plain vector element.

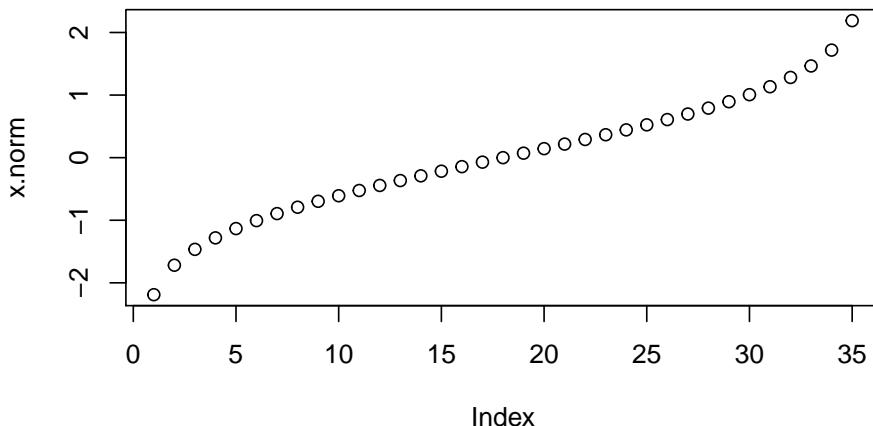
```
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following object is masked from 'package:gridExtra':
#>
#>     combine
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
```

```
df    <- lattice::singer
alto <- df %>%
  filter(voice.part == "Alto 1") %>%
  arrange(height) %>%
  pull(height) %>%
  print
#> [1] 60 61 61 61 61 62 62 62 63 63 63 63 64 64 64 65 65 65 65 66 66 66 66
#> [24] 66 66 66 67 67 67 67 68 68 68 69 70 72
```

Next, we need to find the matching normal distribution quantiles. We first find the f-values for alto, then use qnorm to find the matching normal distribution values from those same f-values

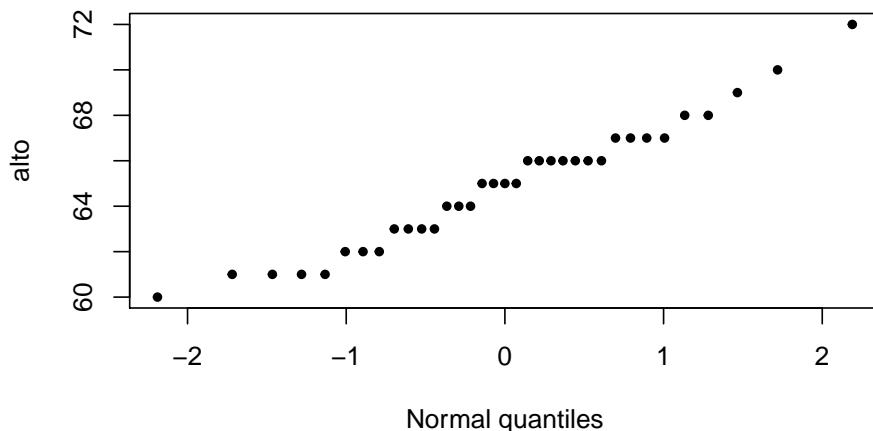
```
i <- 1:length(alto)
fi <- (i - 0.5) / length(alto)
fi
#> [1] 0.0143 0.0429 0.0714 0.1000 0.1286 0.1571 0.1857 0.2143 0.2429 0.2714
#> [11] 0.3000 0.3286 0.3571 0.3857 0.4143 0.4429 0.4714 0.5000 0.5286 0.5571
#> [21] 0.5857 0.6143 0.6429 0.6714 0.7000 0.7286 0.7571 0.7857 0.8143 0.8429
#> [31] 0.8714 0.9000 0.9286 0.9571 0.9857
x.norm <- qnorm(fi)
x.norm
#> [1] -2.1893 -1.7185 -1.4652 -1.2816 -1.1332 -1.0063 -0.8938 -0.7916
#> [9] -0.6971 -0.6085 -0.5244 -0.4439 -0.3661 -0.2905 -0.2165 -0.1437
#> [17] -0.0717 0.0000 0.0717 0.1437 0.2165 0.2905 0.3661 0.4439
#> [25] 0.5244 0.6085 0.6971 0.7916 0.8938 1.0063 1.1332 1.2816
#> [33] 1.4652 1.7185 2.1893
```

```
plot(x.norm)
```



Now we can plot the sorted alto values against the normal values.

```
plot(alto ~ x.norm, type="p", xlab="Normal quantiles", pch=20)
```



When comparing a batch of numbers to a theoretical distribution on a q-q plot, we are looking for significant deviation from a straight line. To make it easier to judge straightness, we can fit a line to the points. Note that we are not creating a 45° (or $x=y$) slope; the range of values between both sets of numbers do not match. Here, we are only seeking the straightness of the points.

There are many ways one can fit a line to the data, Cleveland opts to fit a line to the first and third quartile of the q-q plot. The following chunk of code identifies the quantiles for both the alto dataset and the theoretical normal distribution. It then computes the slope and intercept from these coordinates.

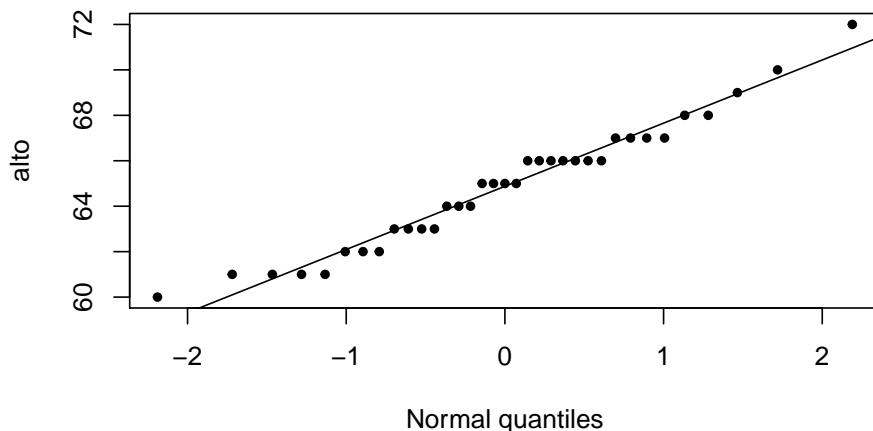
```
# Find 1st and 3rd quartile for the Alto 1 data
y <- quantile(alto, c(0.25, 0.75), type=5)
y
#> 25% 75%
#> 63.0 66.8

# Find the 1st and 3rd quartile of the normal distribution
x <- qnorm(c(0.25, 0.75))
x
#> [1] -0.674  0.674

# Now we can compute the intercept and slope of the line that passes
# through these points
slope <- diff(y) / diff(x)
int   <- y[1] - slope * x[1]
```

Next, we add the line to the plot.

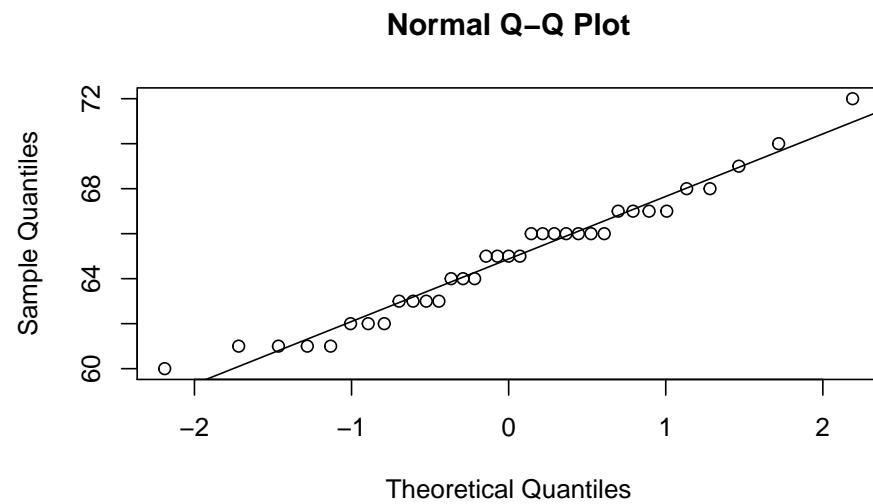
```
plot(alto ~ x.norm, type="p", xlab="Normal quantiles", pch=20)
abline(a=int, b=slope )
```



32.4 Using R's built-in functions

R has two built-in functions that facilitate the plot building task when comparing a batch to a normal distribution: `qqnorm` and `qqline`. Note that the function `qqline` allows the user to define the quantile method via the `qtype=` parameter. Here, we set it to 5 to match our choice of f-value calculation.

```
qqnorm(alto)          # plot the points
qqline(alto, qtype=5) # plot the line
```



That's it. Just two lines of code!

32.5 Using the ggplot2 plotting environment

We can take advantage of the `stat_qq()` function to plot the points, but the equation for the line must be computed manually (as was done earlier). Those steps will be repeated here.

```
# normal distribution
library(ggplot2)

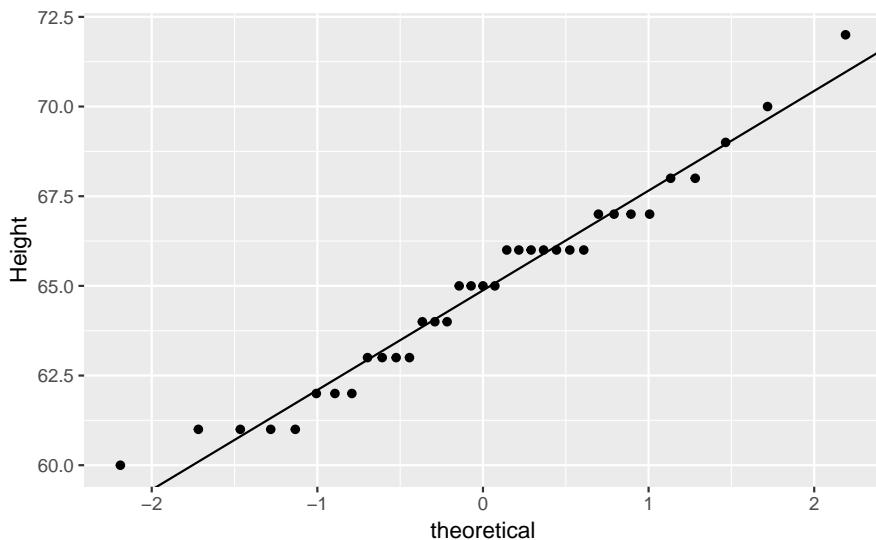
# Find the slope and intercept of the line that passes through the 1st and 3rd
# quartile of the normal q-q plot
```

```

y      <- quantile(alto, c(0.25, 0.75), type=5) # Find the 1st and 3rd quartiles
x      <- qnorm( c(0.25, 0.75))                # Find the matching normal values on the x-axis
slope <- diff(y) / diff(x)                      # Compute the line slope
int   <- y[1] - slope * x[1]                     # Compute the line intercept

# Generate normal q-q plot
ggplot() + aes(sample=alto) +
  stat_qq(distribution=qnorm) +
  geom_abline(intercept=int, slope=slope) +
  ylab("Height")

```



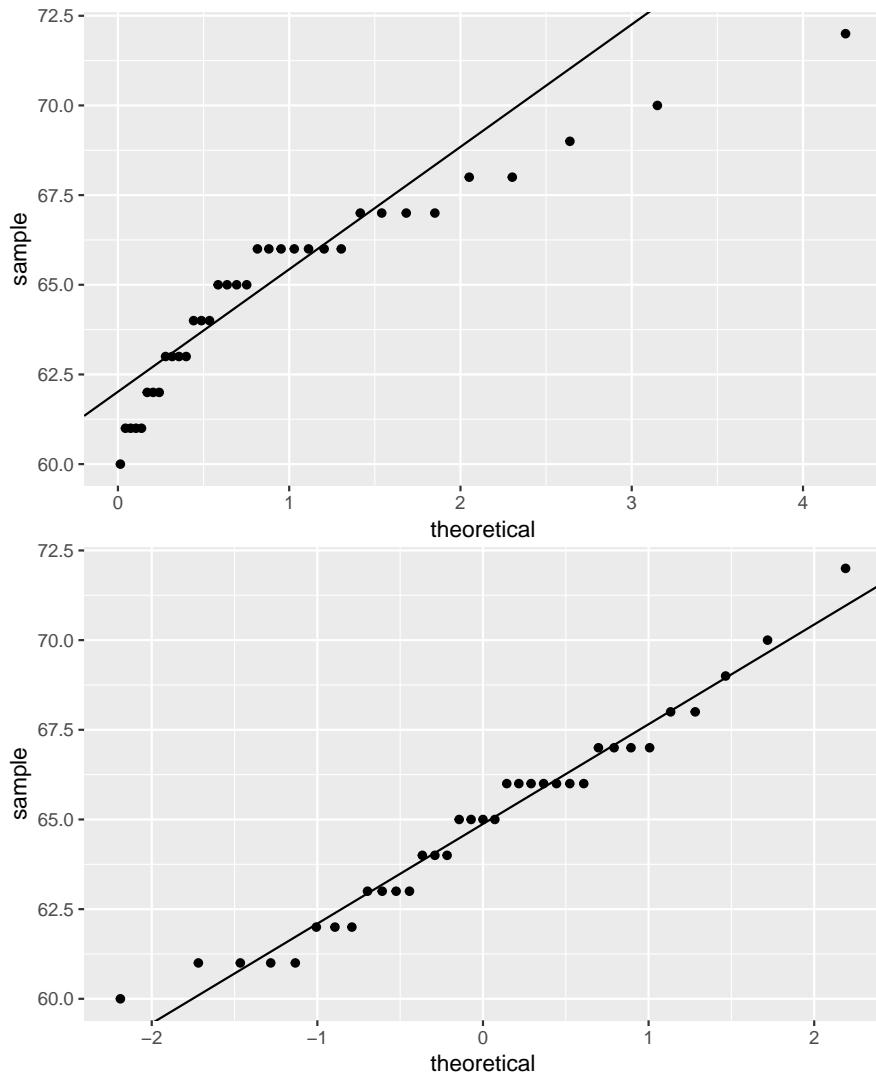
```

qq_any <- function(var, f) {
  # Find the slope and intercept of the line that passes through the 1st and 3rd
  # quartile of the normal q-q plot

  y      <- quantile(var, c(0.25, 0.75), type=5) # Find the 1st and 3rd quartiles
  x      <- f( c(0.25, 0.75))                  # Find the matching normal values x-axis
  slope <- diff(y) / diff(x)                    # Compute the line slope
  int   <- y[1] - slope * x[1]                  # Compute the line intercept
  ggplot() + aes(sample = var) +
  stat_qq(distribution = f) +
  geom_abline(intercept=int, slope=slope)
}

# two function only, for the moment
qq_any(alto, qexp)
qq_any(alto, qnorm)

```



We can, of course, make use of ggplot's faceting function to generate trellised plots. For example, the following plot replicates Cleveland's figure 2.11 (except for the layout which we'll setup as a single row of plots instead). But first, we will need to compute the slopes for each singer group. We'll use dplyr's piping operations to create a new dataframe with singer group name, slope and intercept.

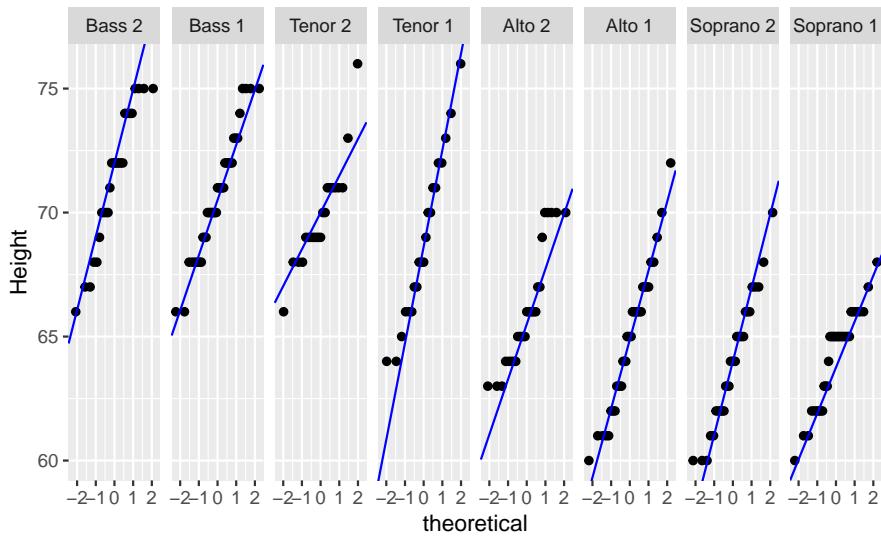
```
library(dplyr)

intsl <- df %>%
  group_by(voice.part) %>%
  summarize(q25      = quantile(height, 0.25, type=5),
            q75      = quantile(height, 0.75, type=5),
            norm25   = qnorm( 0.25),
            norm75   = qnorm( 0.75),
            slope    = (q25 - q75) / (norm25 - norm75),
            int     = q25 - slope * norm25) %>%
  select(voice.part, slope, int) %>%
  print
#> # A tibble: 8 x 3
#>   voice.part slope   int
#>   <fct>     <dbl> <dbl>
```

```
#> 1 Bass 2      2.97  72
#> 2 Bass 1      2.22  70.5
#> 3 Tenor 2     1.48   70
#> 4 Tenor 1     3.89  68.6
#> 5 Alto 2      2.22  65.5
#> 6 Alto 1      2.78  64.9
#> # ... with 2 more rows
```

It's important that the `voice.part` names match those in `df` letter-for-letter so that when `ggplot` is called, it will know which facet to assign the slope and intercept values to via `geom_abline`.

```
ggplot(df, aes(sample = height)) +
  stat_qq(distribution = qnorm) +
  geom_abline(data=intsl, aes(intercept=int, slope=slope), col="blue") +
  facet_wrap(~voice.part, nrow=1) +
  ylab("Height")
```



Chapter 33

QQ and PP Plots

<https://homepage.divms.uiowa.edu/~luke/classes/STAT4580/qqpp.html>

33.1 QQ Plot

One way to assess how well a particular theoretical model describes a data distribution is to plot data quantiles against theoretical quantiles.

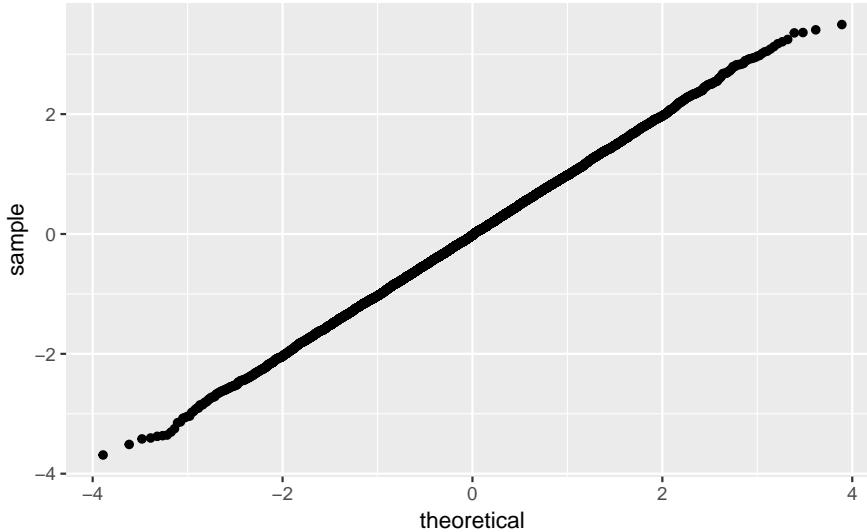
Base graphics provides `qqnorm`, lattice has `qqmath`, and ggplot2 has `geom_qq`.

The default theoretical distribution used in these is a standard normal, but, except for `qqnorm`, these allow you to specify an alternative.

For a large sample from the theoretical distribution the plot should be a straight line through the origin with slope 1:

```
library(ggplot2)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures    rlang
#>   c.quosures    rlang
#>   print.quosures rlang

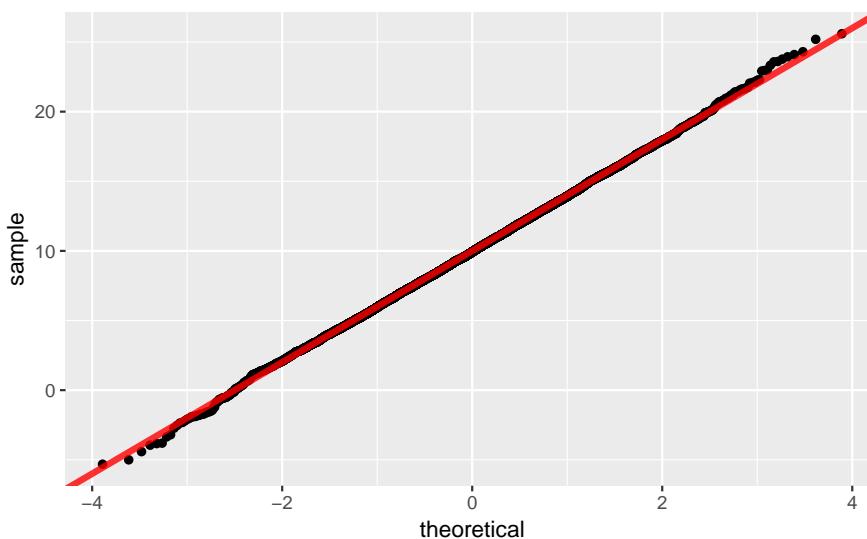
n <- 10000
ggplot() + geom_qq(aes(sample = rnorm(n)))
```



If the plot is a straight line with a different slope or intercept, then the data distribution corresponds to a location-scale transformation of the theoretical distribution.

The slope is the scale and the intercept is the location:

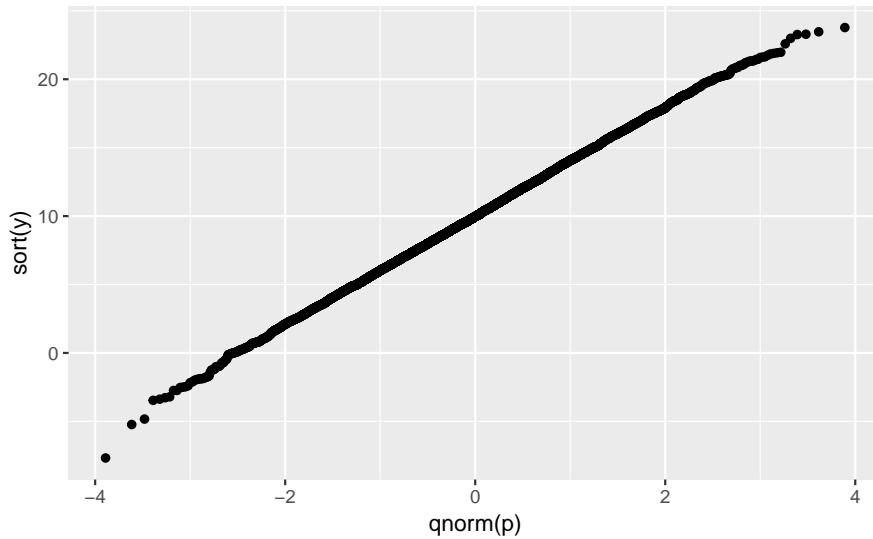
```
ggplot() +
  geom_qq(aes(sample = rnorm(n, 10, 4))) +
  geom_abline(intercept = 10, slope = 4,
              color = "red", size = 1.5, alpha = 0.8)
```



The QQ plot can be constructed directly as a scatterplot of the sorted sample $i = 1, \dots, n$ against quantiles for

$$p_i = \frac{i}{n} - \frac{1}{2n}$$

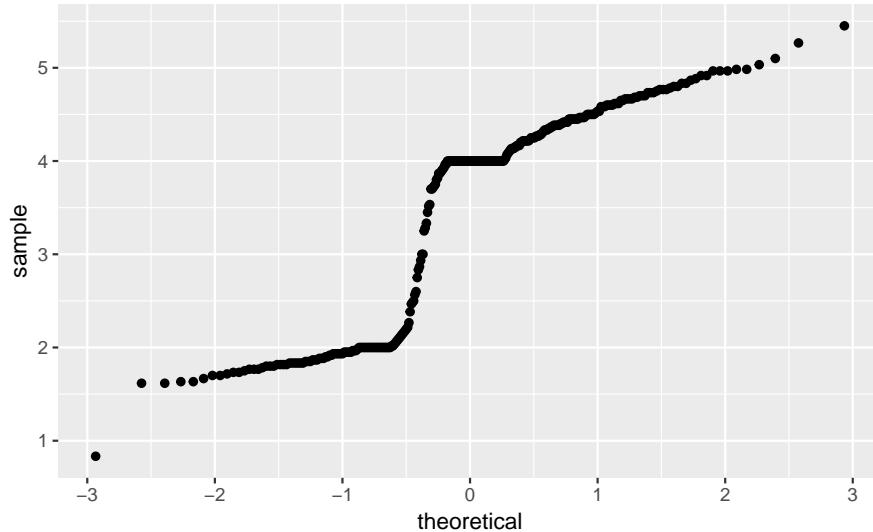
```
p <- (1 : n) / n - 0.5 / n
y <- rnorm(n, 10, 4)
ggplot() + geom_point(aes(x = qnorm(p), y = sort(y)))
```



33.2 Some Examples

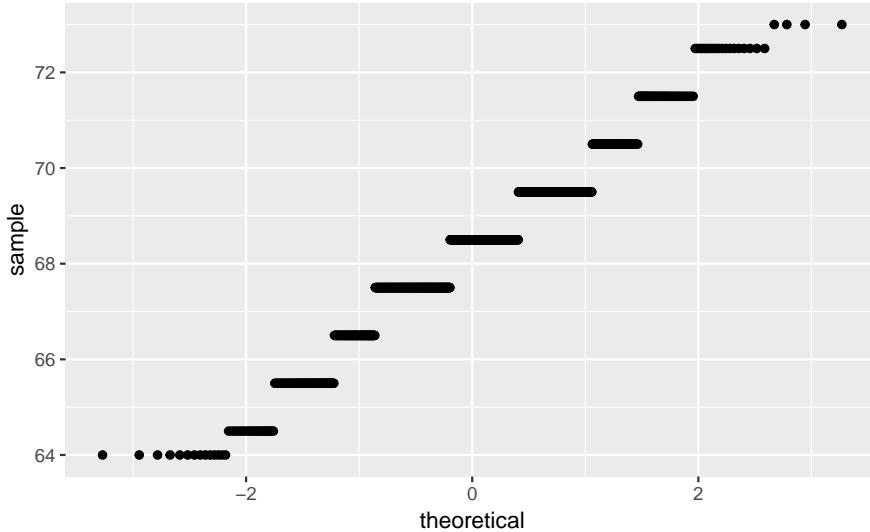
The histograms and density estimates for the duration variable in the `geyser` data set showed that the distribution is far from a normal distribution, and the normal QQ plot shows this as well:

```
library(MASS)
ggplot(geyser) + geom_qq(aes(sample = duration))
```



Except for rounding the parent heights in the Galton data seemed not too fat from normally distributed:

```
library(psych)
ggplot(galton) + geom_qq(aes(sample = parent))
```

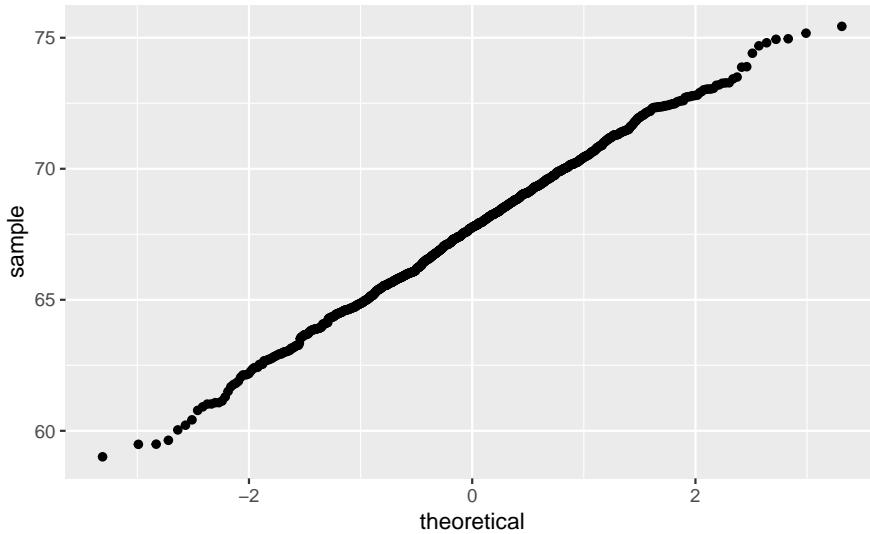


Rounding interferes more with this visualization than with a histogram or a density plot.

Rounding is more visible with this visualization than with a histogram or a density plot.

Another Gatlton dataset available in the UsingR package with less rounding is father.son:

```
library(UsingR)
ggplot(father.son) + geom_qq(aes(sample = fheight))
```



The middle seems to be fairly straight, but the ends are somewhat wiggly.

How can you calibrate your judgment?

33.3 Calibrating the Variability

One approach is to use simulation, sometimes called a graphical bootstrap.

The `nboot` function will simulate R samples from a normal distribution that match a variable `x` on sample size, sample mean, and sample SD.

The result is returned in a data frame suitable for plotting:

```

nsim <- function(n, m = 0, s = 1) {
  z <- rnorm(n)
  m + s * ((z - mean(z)) / sd(z))
}

nboot <- function(x, R) {
  n <- length(x)
  m <- mean(x)
  s <- sd(x)
  do.call(rbind,
    lapply(1 : R,
      function(i) {
        xx <- sort(nsim(n, m, s))
        p <- seq_along(xx) / n - 0.5 / n
        data.frame(x = xx, p = p, sim = i)
      }))
}

```

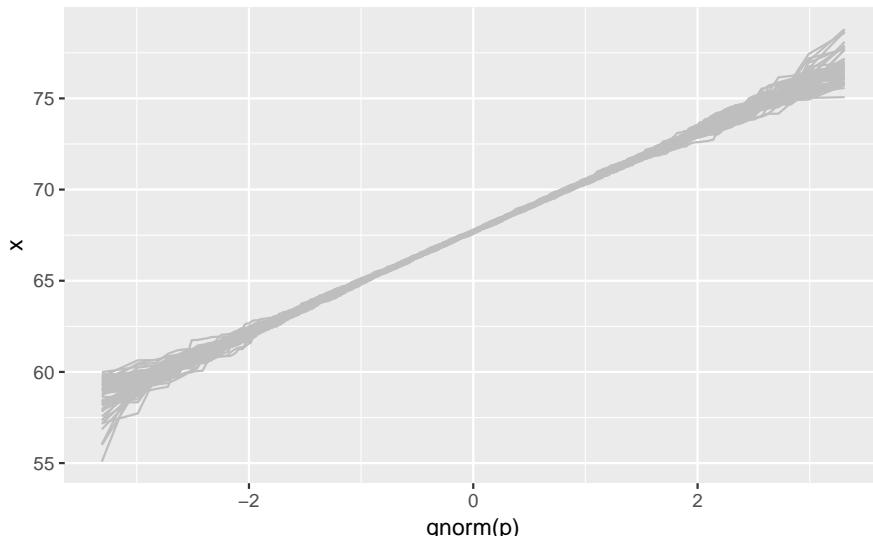
Plotting these as lines shows the variability in shapes we can expect when sampling from the theoretical normal distribution:

```

gb <- nboot(father.son$fheight, 50)
tibble::as_tibble(gb)
#> # A tibble: 53,900 x 3
#>   x     p   sim
#>   <dbl> <dbl> <int>
#> 1 59.8 0.000464     1
#> 2 59.9 0.00139     1
#> 3 59.9 0.00232     1
#> 4 60.8 0.00325     1
#> 5 60.8 0.00417     1
#> 6 60.9 0.00510     1
#> # ... with 5.389e+04 more rows

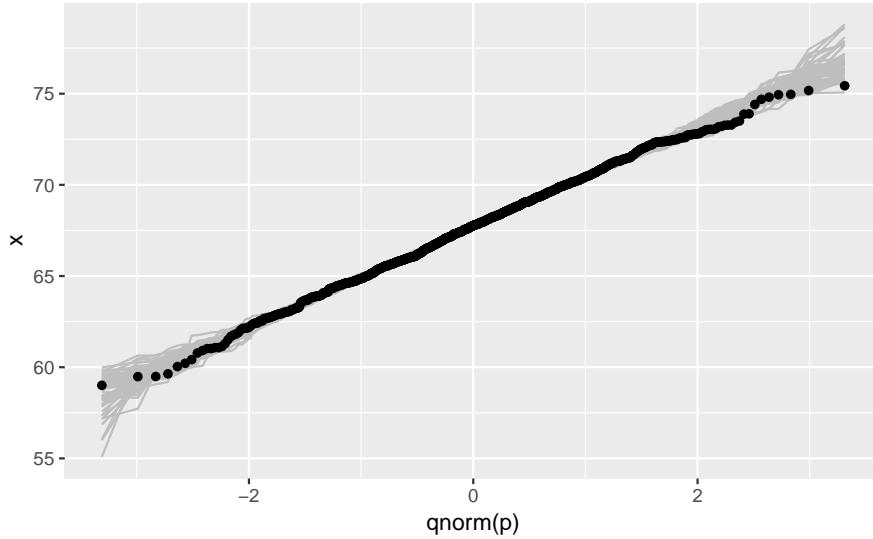
ggplot() +
  geom_line(aes(x = qnorm(p), y = x, group = sim),
            color = "gray", data = gb)

```



We can then insert this simulation behind our data to help calibrate the visualization:

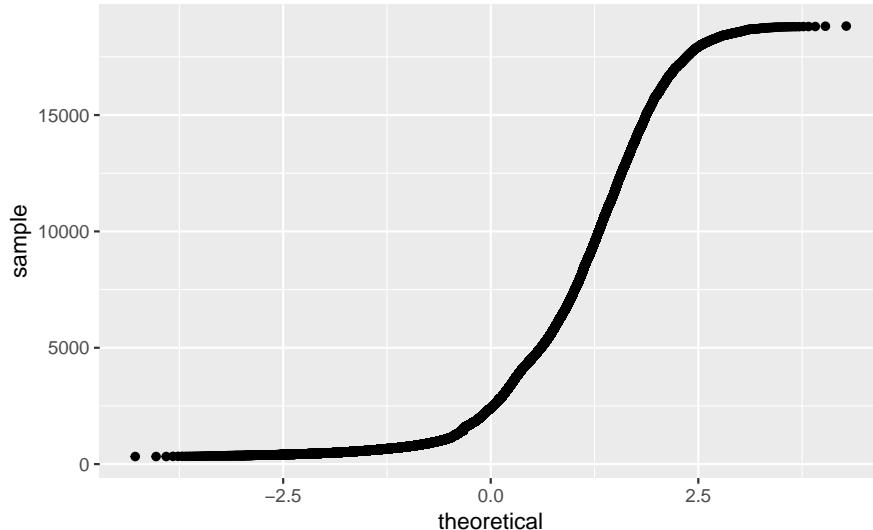
```
ggplot(father.son) +
  geom_line(aes(x = qnorm(p), y = x, group = sim),
            color = "gray", data = gb) +
  geom_qq(aes(sample = fheight))
```



33.4 Scalability

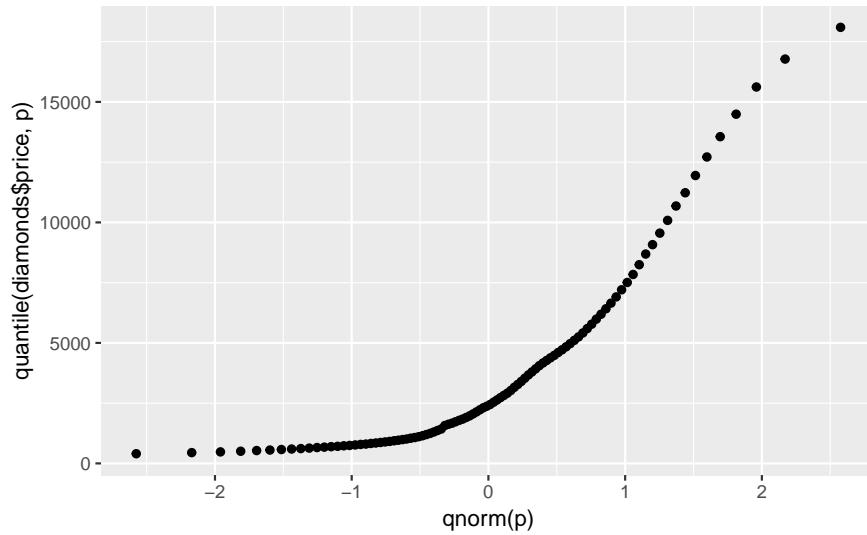
For large sample sizes overplotting will occur:

```
ggplot(diamonds) + geom_qq(aes(sample = price))
```



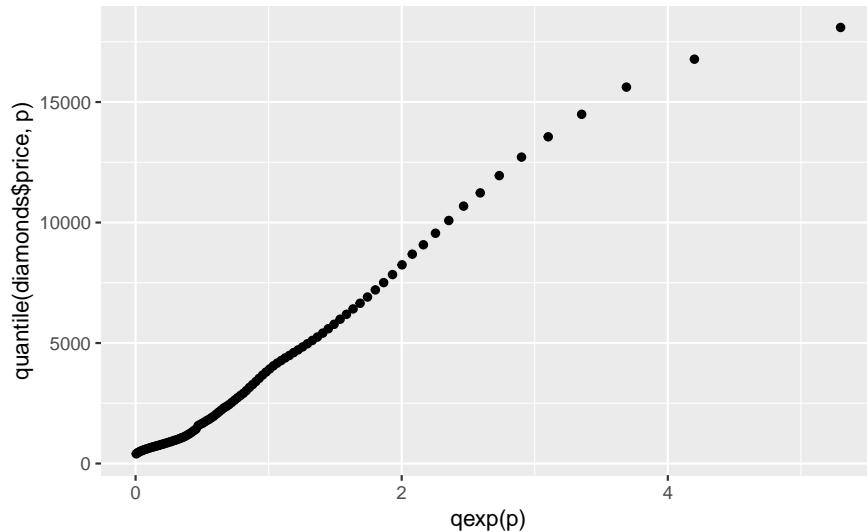
This can be alleviated by using a grid of quantiles:

```
nq <- 100
p <- (1 : nq) / nq - 0.5 / nq
ggplot() + geom_point(aes(x = qnorm(p), y = quantile(diamonds$price, p)))
```



A more reasonable model might be an exponential distribution:

```
ggplot() + geom_point(aes(x = qexp(p), y = quantile(diamonds$price, p)))
```



33.5 Comparing Two Distributions

The QQ plot can also be used to compare two distributions based on a sample from each.

If the samples are the same size then this is just a plot of the ordered sample values against each other.

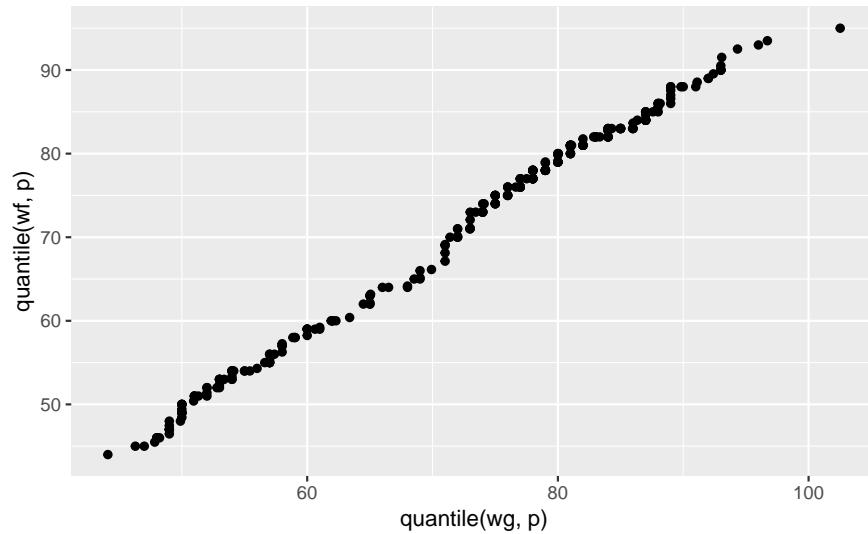
Choosing a fixed set of quantiles allows samples of unequal size to be compared.

Using a small set of quantiles we can compare the distributions of waiting times between eruptions of Old Faithful from the two different data sets we have looked at:

```
nq <- 31 # user defined
nq <- min(length(geyser$waiting), length(faithful$waiting)) # or take the minimum
p <- (1 : nq) / nq - 0.5 / nq

wg <- geyser$waiting
```

```
wf <- faithful$waiting
ggplot() + geom_point(aes(x = quantile(wg, p), y = quantile(wf, p)))
```



33.6 PP Plots

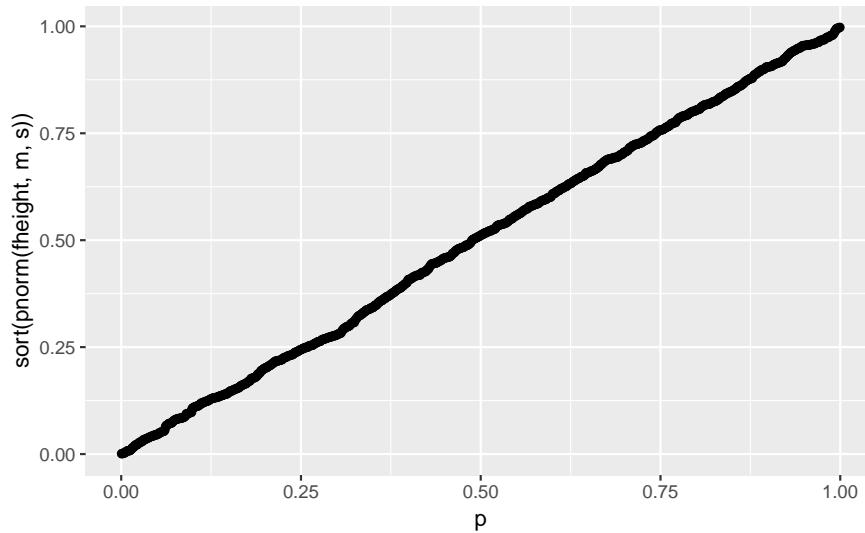
The PP plot for comparing a sample to a theoretical model plots the theoretical proportion less than or equal to each observed value against the actual proportion.

For a theoretical cumulative distribution function F this means plotting

$$F(x(i))pi$$

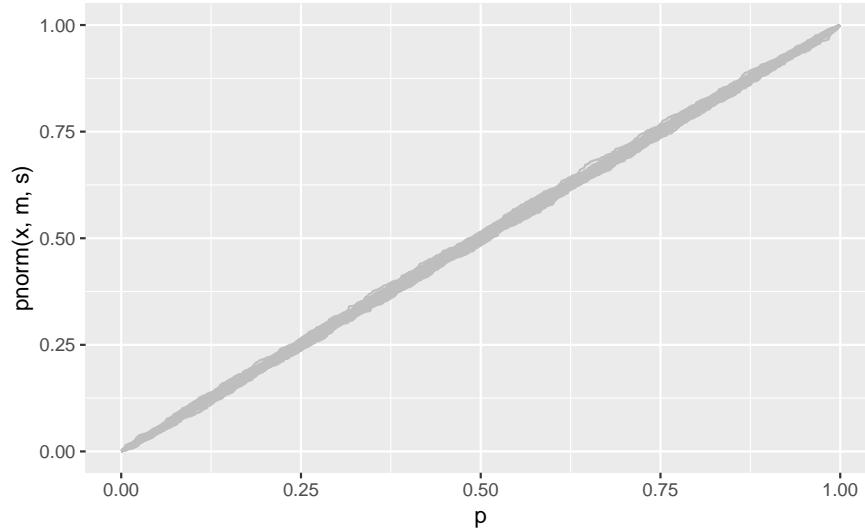
For the `fheight` variable in the `father.son` data:

```
m <- mean(father.son$fheight)
s <- sd(father.son$fheight)
n <- nrow(father.son)
p <- (1 : n) / n - 0.5 / n
ggplot(father.son) + geom_point(aes(x = p, y = sort(pnorm(fheight, m, s))))
```



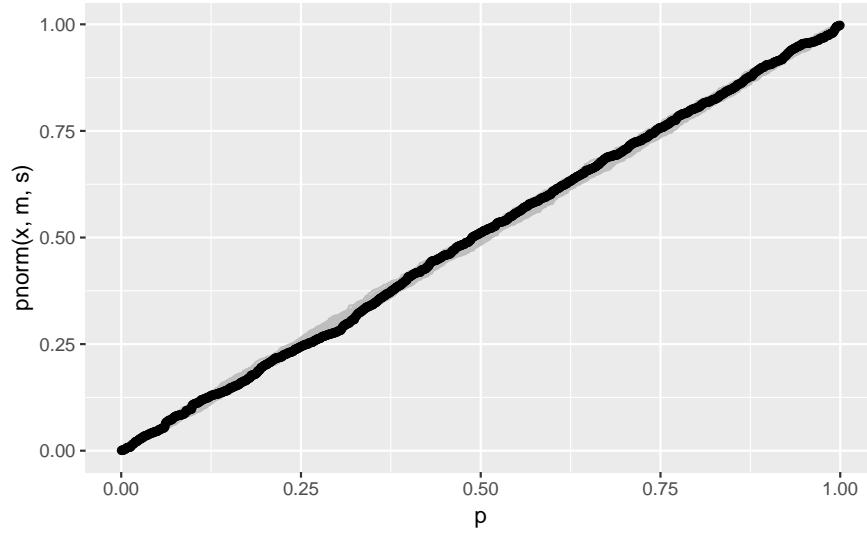
- The values on the vertical axis are the probability integral transform of the data for the theoretical distribution.
- If the data are a sample from the theoretical distribution then these transforms would be uniformly distributed on [0,1].
- The PP plot is a QQ plot of these transformed values against a uniform distribution.
- The PP plot goes through the points (0,0) and (1,1) and so is much less variable in the tails:

```
pp <- ggplot() +
  geom_line(aes(x = p, y = pnorm(x, m, s), group = sim),
            color = "gray", data = gb)
pp
```



Adding the data:

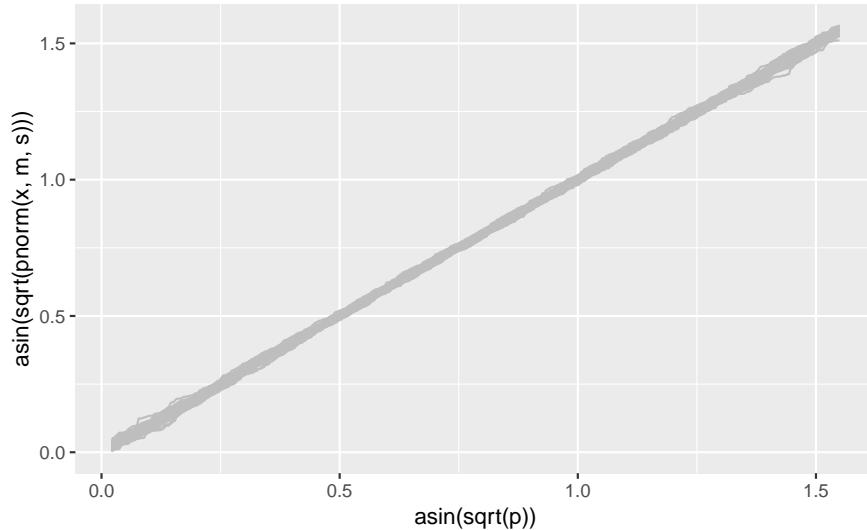
```
pp +
  geom_point(aes(x = p, y = sort(pnorm(fheight, m, s))), data = (father.son))
```



The PP plot is also less sensitive to deviations in the tails.

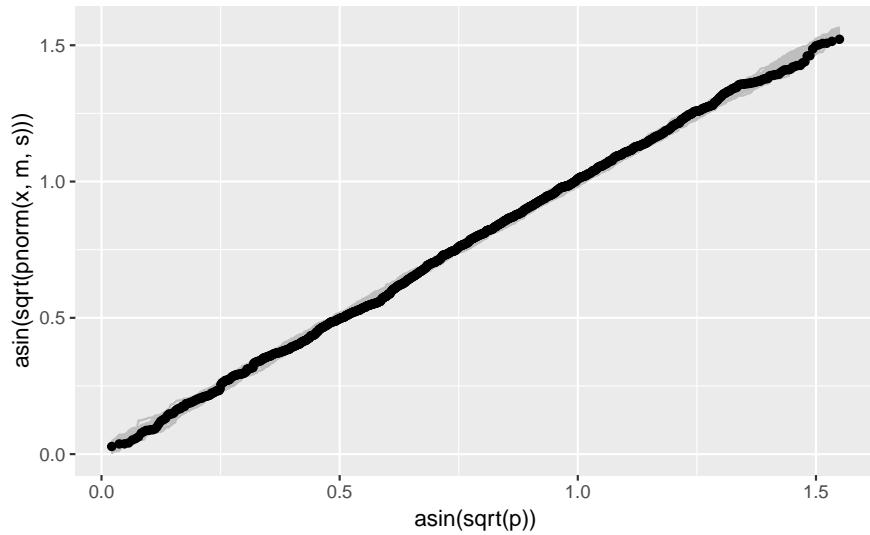
A compromise between the QQ and PP plots uses the arcsine square root variance-stabilizing transformation, which makes the variability approximately constant across the range of the plot:

```
vpp <- ggplot() +
  geom_line(aes(x = asin(sqrt(p)), y = asin(sqrt(pnorm(x, m, s)))), group = sim),
  color = "gray", data = gb)
vpp
```



Adding the data:

```
vpp +
  geom_point(aes(x = asin(sqrt(p)), y = sort(asin(sqrt(pnorm(fheight, m, s))))),
  data = (father.son))
```



33.7 Plots For Assessing Model Fit

- Both QQ and PP plots can be used to assess how well a theoretical family of models fits your data, or your residuals.
- To use a PP plot you have to estimate the parameters first.
- For a location-scale family, like the normal distribution family, you can use a QQ plot with a standard member of the family.
- Some other families can use other transformations that lead to straight lines for family members:

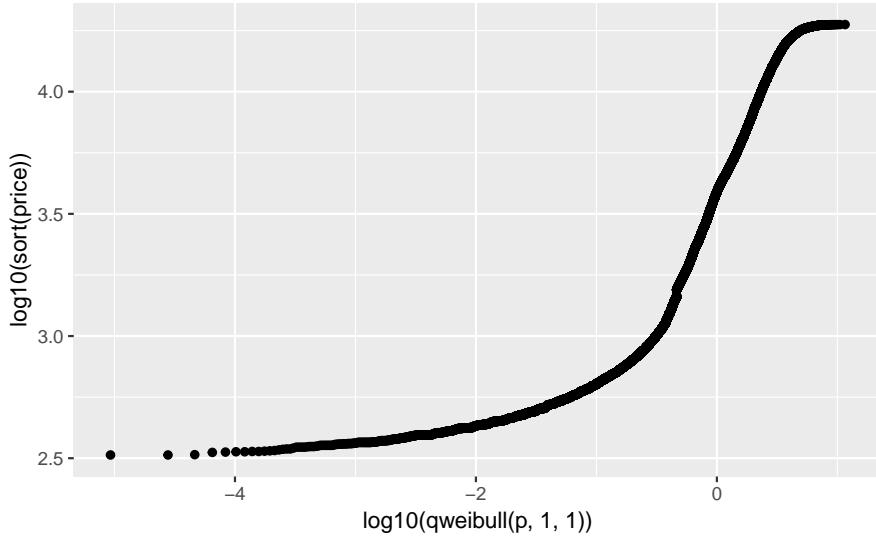
The Weibull family is widely used in reliability modeling; its CDF is

$$F(t) = 1 - \exp \left\{ - \left(\frac{t}{b} \right)^a \right\}$$

- The logarithms of Weibull random variables form a location-scale family.
- Special paper used to be available for Weibull probability plots.

A Weibull QQ plot for price in the diamonds data:

```
n <- nrow(diamonds)
p <- (1 : n) / n - 0.5 / n
ggplot(diamonds) +
  geom_point(aes(x = log10(qweibull(p, 1, 1)), y = log10(sort(price))))
```



- The lower tail does not match a Weibull distribution.
- Is this important?
- In engineering applications it often is.
- In selecting a reasonable model to capture the shape of this distribution it may not be.
- QQ plots are helpful for understanding departures from a theoretical model.
- No data will fit a theoretical model perfectly.
- Case-specific judgment is needed to decide whether departures are important.
- George Box: All models are wrong but some are useful.

Chapter 34

Data Visualization: Working with models

34.1 Introduction

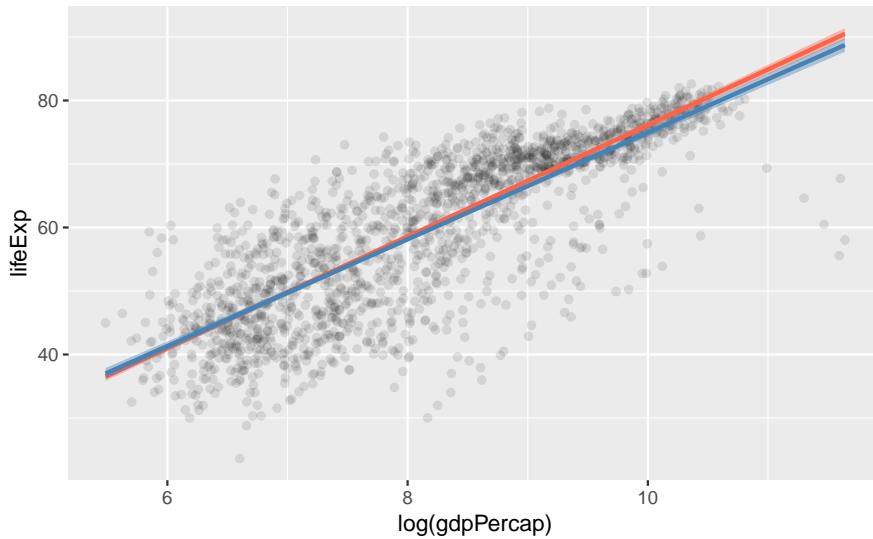
Source: <https://socviz.co/modeling.html>

Data visualization is about more than generating figures that display the raw numbers from a table of data. Right from the beginning, it involves summarizing or transforming parts of the data, and then plotting the results. Statistical models are a central part of that process. In this Chapter, we will begin by looking briefly at how `ggplot` can use various modeling techniques directly within geoms. Then we will see how to use the `broom` and `margins` libraries to tidily extract and plot estimates from models that we fit ourselves.

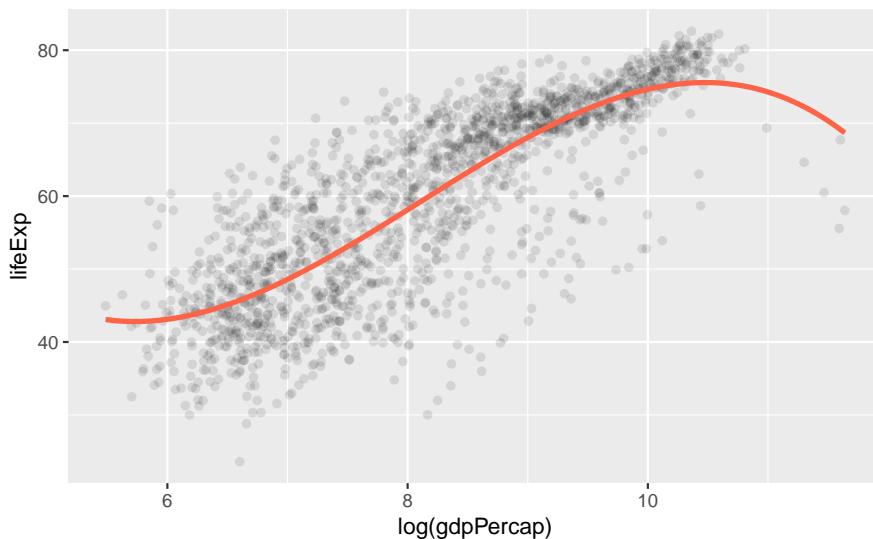
```
# load libraries
library(ggplot2)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures    rlang
#>   c.quosures    rlang
#>   print.quosures rlang
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>   filter, lag
#> The following objects are masked from 'package:base':
#>
#>   intersect, setdiff, setequal, union
library(tidyr)
library(purrr)
library(socviz)      # devtools::install_github("kjhealy/socviz")
library(gapminder)

# plot two lines
p <- ggplot(data = gapminder,
             mapping = aes(x = log(gdpPercap), y = lifeExp))
```

```
p + geom_point(alpha=0.1) +
  geom_smooth(color = "tomato", fill="tomato", method = MASS::rlm) +
  geom_smooth(color = "steelblue", fill="steelblue", method = "lm")
```

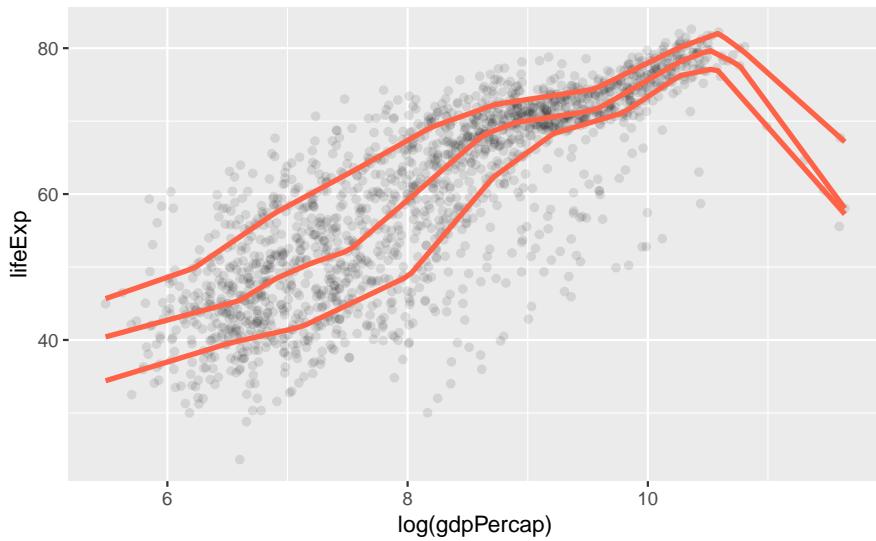


```
# plot spline
p + geom_point(alpha=0.1) +
  geom_smooth(color = "tomato", method = "lm", size = 1.2,
              formula = y ~ splines::bs(x, 3), se = FALSE)
```



```
p + geom_point(alpha=0.1) +
  geom_quantile(color = "tomato", size = 1.2, method = "rqss",
                lambda = 1, quantiles = c(0.20, 0.5, 0.85))
#> Loading required package: SparseM
#>
#> Attaching package: 'SparseM'
#> The following object is masked from 'package:base':
#>
#>     backsolve
#> Smoothing formula not specified. Using: y ~ qss(x, lambda = 1)
```

```
#> Warning in rq.fit.sfn(x, y, tau = tau, rhs = rhs, control = control, ...): tiny diagonals replaced w
```



Histograms, density plots, boxplots, and other geoms compute either single numbers or new variables before plotting them. As we saw in Section 4.4, these calculations are done by `stat_` functions, each of which works hand-in-hand with its default `geom_` function, and vice versa. Moreover, from the smoothing lines we drew from almost the very first plots we made, we have seen that `stat_` functions can do a fair amount of calculation and even model estimation on the fly. The `geom_smooth()` function can take a range of method arguments to fit LOESS, OLS, and robust regression lines, amongst others.

Both the `geom_smooth()` and `geom_quantile()` functions can also be instructed to use different formulas to produce their fits. In the top panel of Figure 6.1, we access the MASS library's `r1m` function to fit a robust regression line. In the second panel, the `bs` function is invoked directly from the `splines` library in the same way, to fit a polynomial curve to the data. This is the same approach to directly accessing functions without loading a whole library that we have already used several times when using functions from the `scales` library. The `geom_quantile()` function, meanwhile, is like a specialized version of `geom_smooth()` that can fit quantile regression lines using a variety of methods. The `quantiles` argument takes a vector specifying the quantiles at which to fit the lines.

34.2 Show several fits at once, with a legend

As we just saw in the first panel of Figure 6.1, where we plotted both an OLS and a robust regression line, we can look at several fits at once on the same plot by layering on new smoothers with `geom_smooth()`. As long as we set the color and fill aesthetics to different values for each fit, we can easily distinguish them visually. However, `ggplot` will not draw a legend that guides us about which fit is which. This is because the smoothers are not logically connected to one another. They exist as separate layers. What if we are comparing several different fits and want a legend describing them?

As it turns out, `geom_smooth()` can do this via the slightly unusual route of mapping the color and fill aesthetics to a string describing the model we are fitting, and then using `scale_color_manual()` and `scale_fill_manual()` to create the legend. First we use `brewer.pal()` from the `RColorBrewer` library to extract three qualitatively different colors from a larger palette. The colors are represented as hex values. As before use the `::` convention to use the function without loading the whole library:

```
model_colors <- RColorBrewer::brewer.pal(3, "Set1")
model_colors
```

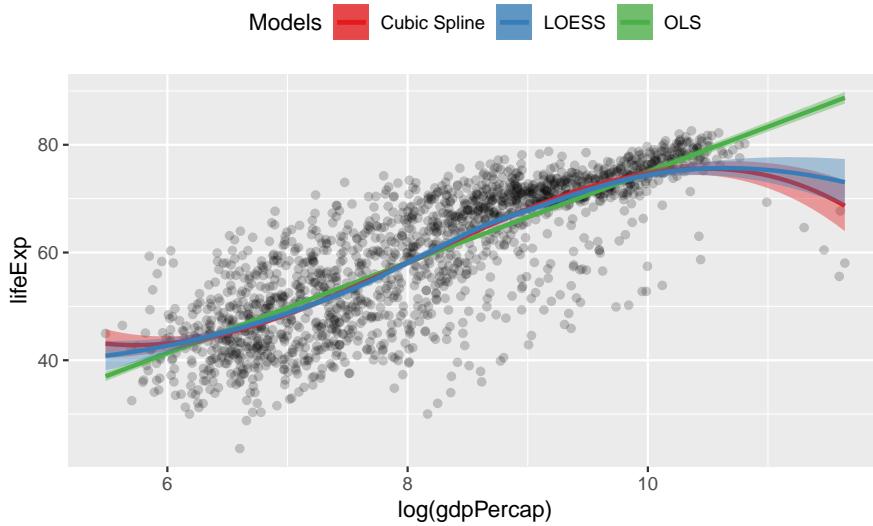
```
#> [1] "#E41A1C" "#377EB8" "#4DAF4A"
```

Then we create a plot with three different smoothers, mapping the color and fill within the `aes()` function as the name of the smoother:

```
p0 <- ggplot(data = gapminder,
               mapping = aes(x = log(gdpPercap), y = lifeExp))

p1 <- p0 + geom_point(alpha = 0.2) +
  geom_smooth(method = "lm", aes(color = "OLS", fill = "OLS")) +
  geom_smooth(method = "lm", formula = y ~ splines::bs(x, df = 3),
              aes(color = "Cubic Spline", fill = "Cubic Spline")) +
  geom_smooth(method = "loess",
              aes(color = "LOESS", fill = "LOESS"))

p1 + scale_color_manual(name = "Models", values = model_colors) +
  scale_fill_manual(name = "Models", values = model_colors) +
  theme(legend.position = "top")
```



In a way we have cheated a little here to make the plot work. Until now, we have always mapped aesthetics to the names of variables, not to strings like “OLS” or “Cubic Splines”. In Chapter 3, when we discussed mapping versus setting aesthetics, we saw what happened when we tried to change the color of the points in a scatterplot by setting them to “purple” inside the `aes()` function. The result was that the points turned red instead, as `ggplot` in effect created a new variable and labeled it with the word “purple”. We learned there that the `aes()` function was for mapping variables to aesthetics.

Here we take advantage of that behavior, creating a new single-value variable for the name of each of our models. Ggplot will properly construct the relevant guide if we call `scale_color_manual()` and `scale_fill_manual()`. Remember that we have to call two scale functions because we have two mappings. The result is a single plot containing not just our three smoothers, but also an appropriate legend to guide the reader.

These model-fitting features make `ggplot` very useful for exploratory work, and make it straightforward to generate and compare model-based trends and other summaries as part of the process of descriptive data visualization. The various `stat_` functions are a flexible way to add summary estimates of various kinds to plots. But we will also want more than this, including presenting results from models we fit ourselves.

34.3 Look inside model objects

Covering the details of fitting statistical models in R is beyond the scope of this book. For a comprehensive, modern introduction to that topic you should work your way through (Gelman & Hill, 2018). (Harrell, 2016) is also very good on the many practical connections between modeling and graphing data. Similarly, (Gelman, 2004) provides a detailed discussion of the use of graphics as a tool in model-checking and validation. Here we will discuss some ways to take the models that you fit and extract information that is easy to work with in `ggplot`. Our goal, as always, is to get from however the object is stored to a tidy table of numbers that we can plot. Most classes of statistical model in R will contain the information we need, or will have a special set of functions, or methods, designed to extract it.

We can start by learning a little more about how the output of models is stored in R. Remember, we are always working with objects, and objects have an internal structure consisting of named pieces. Sometimes these are single numbers, sometimes vectors, and sometimes lists of things like vectors, matrices, or formulas.

We have been working extensively with tibbles and data frames. These store tables of data with named columns, perhaps consisting of different classes of variable, such as integers, characters, dates, or factors. Model objects are a little more complicated again.

```
gapminder
#> # A tibble: 1,704 x 6
#>   country     continent    year lifeExp      pop gdpPercap
#>   <fct>       <fct>     <int>   <dbl>     <int>     <dbl>
#> 1 Afghanistan Asia      1952    28.8  8425333    779.
#> 2 Afghanistan Asia      1957    30.3  9240934   821.
#> 3 Afghanistan Asia      1962    32.0  10267083   853.
#> 4 Afghanistan Asia      1967    34.0  11537966   836.
#> 5 Afghanistan Asia      1972    36.1  13079460   740.
#> 6 Afghanistan Asia      1977    38.4  14880372   786.
#> # ... with 1,698 more rows
```

Remember, we can use the `str()` function to learn more about the internal structure of any object. For example, we can get some information on what class (or classes) of object `gapminder` is, how large it is, and what components it has. The output from `str(gapminder)` is somewhat dense:

```
str(gapminder)
#> Classes 'tbl_df', 'tbl' and 'data.frame': 1704 obs. of 6 variables:
#> $ country : Factor w/ 142 levels "Afghanistan",...: 1 1 1 1 1 1 1 1 1 ...
#> $ continent: Factor w/ 5 levels "Africa", "Americas", ...: 3 3 3 3 3 3 3 3 3 ...
#> $ year     : int 1952 1957 1962 1967 1972 1977 1982 1987 1992 1997 ...
#> $ lifeExp  : num 28.8 30.3 32 34 36.1 ...
#> $ pop      : int 8425333 9240934 10267083 11537966 13079460 14880372 12881816 13867957 16317921 22...
```

There is a lot of information here about the object as a whole and each variable in it. In the same way, statistical models in R have an internal structure. But because models are more complex entities than data tables, their structure is correspondingly more complicated. There are more pieces of information, and more kinds of information, that we might want to use. All of this information is generally stored in or is computable from parts of a model object.

We can create a linear model, an ordinary OLS regression, using the `gapminder` data. This dataset has a country-year structure that makes an OLS specification like this the wrong one to use. But never mind that for now. We use the `lm()` function to run the model, and store it in an object called `out`:

```
out <- lm(formula = lifeExp ~ gdpPercap + pop + continent,
          data = gapminder)
```

The first argument is the formula for the model. `lifeExp` is the dependent variable and the tilde `~` operator is used to designate the left- and right-hand sides of a model (including in cases, as we saw with `facet_wrap()` where the model just has a right-hand side.)

Let's look at the results by asking R to print a summary of the model.

```
summary(out)
#>
#> Call:
#> lm(formula = lifeExp ~ gdpPercap + pop + continent, data = gapminder)
#>
#> Residuals:
#>   Min     1Q Median     3Q    Max
#> -49.16 -4.49  0.30  5.11 25.17
#>
#> Coefficients:
#>              Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 4.78e+01 3.40e-01 140.82 <2e-16 ***
#> gdpPercap   4.50e-04 2.35e-05 19.16 <2e-16 ***
#> pop         6.57e-09 1.98e-09  3.33  9e-04 ***
#> continentAmericas 1.35e+01 6.00e-01 22.46 <2e-16 ***
#> continentAsia    8.19e+00 5.71e-01 14.34 <2e-16 ***
#> continentEurope   1.75e+01 6.25e-01 27.97 <2e-16 ***
#> continentOceania  1.81e+01 1.78e+00 10.15 <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 8.37 on 1697 degrees of freedom
#> Multiple R-squared:  0.582, Adjusted R-squared:  0.581
#> F-statistic: 394 on 6 and 1697 DF,  p-value: <2e-16
```

When we use the `summary()` function on `out`, we are not getting a simple feed of what's in the model object. Instead, like any function, `summary()` takes its input, performs some actions, and produces output. In this case, what is printed to the console is partly information that is stored inside the model object, and partly information that the `summary()` function has calculated and formatted for display on the screen. Behind the scenes, `summary()` gets help from other functions. Objects of different classes have default methods associated with them, so that when the generic `summary()` function is applied to a linear model object, the function knows to pass the work on to a more specialized function that does a bunch of calculations and formatting appropriate to a linear model object. We use the same generic `summary()` function on data frames, as in `summary(gapminder)`, but in that case a different default method is applied.

Schematic view of a linear model object. Figure 6.3: Schematic view of a linear model object.

The output from `summary()` gives a precis of the model, but we can't really do any further analysis with it directly. For example, what if we want to plot something from the model? The information necessary to make plots is inside the `out` object, but it is not obvious how to use it.

If we take a look at the structure of the model object with `str(out)` we will find that there is a lot of information in there. Like most complex objects in R, `out` is organized as a list of components or elements. Several of these elements are themselves lists. Figure 6.3 gives you a schematic view of the contents of a linear model object. In this list of items, elements are single values, some are data frames, and some are additional lists of simpler items. Again, remember our earlier discussion where we said objects could be

thought of as being organized like a filing system: cabinets contain drawers, and drawer may contain which may contain pages of information, whole documents, or groups of folders with more documents inside. As an alternative analogy, and sticking with the image of a list, you can think of a master to-do list for a project, where the top-level headings lead to contain additional lists of tasks of different kinds.

The `out` object created by `lm` contains several different named elements. Some, like the residual degrees of freedom in the model, are just a single number. Try `out$df.residual` at the console. Others are much larger entities, such as the data frame used to fit the model, which is retained by default. Try `out$model`, but be prepared for a lot of stuff to be printed at the console. Other elements have been computed by R and then stored, such as the coefficients of the model and other quantities. You can try `out$coefficients`, `out$residuals`, and `out$fitted.values`, for instance. Others are lists themselves (like `qr`). So you can see that the `summary()` function is selecting and printing only a small amount of core information, in comparison to what is stored in the model object.

Just like the tables of data we saw earlier in Section A.1.3, the output of `summary()` is presented in a way that is compact and efficient in terms of getting information across, but also untidy when considered from the point of view of further manipulation. There is a table of coefficients, but the variable names are in the rows. The column names are awkward, and some information (e.g. at the bottom of the output) has been calculated and printed out, but is not stored in the model object.

34.4 Get model-based graphics right

Figures based on statistical models face all the ordinary challenges of effective data visualization, and then some. This is because model results usually carry a considerable extra burden of interpretation and necessary background knowledge. The more complex the model, the trickier it becomes to convey this information effectively, and the easier it becomes to lead one's audience or oneself into error. Within the social sciences, our ability to clearly and honestly present model-based graphics has greatly improved over the past ten or fifteen years. Over the same period, it has become clearer that some kinds of models are quite tricky to understand, even ones that had previously been seen as straightforward elements of the modeling toolkit (Ai & Norton, 2003; Brambor, Clark, & Golder, 2006).

Plotting model estimates is closely connected to properly estimating models in the first place. This means there is no substitute for learning the statistics. You should not use graphical methods as a substitute for understanding the model used to produce them. While this book cannot teach you that material, we can make a few general points about what good model-based graphics look like, and work through some examples of how `ggplot` and some additional libraries can make it easier to get good results.

34.4.1 Present your findings in substantive terms

Useful model-based plots show results in ways that are substantively meaningful and directly interpretable with respect to the questions the analysis is trying to answer. This means showing results in a context where other variables in the analysis are held at sensible values, such as their means or medians. With continuous variables, it can often be useful to generate predicted values that cover some substantively meaningful move across the distribution, such as from the **25th to the 75th percentile**, rather than a single-unit increment in the variable of interest. For unordered categorical variables, predicted values might be presented with respect to the modal category in the data, or for a particular category of theoretical interest. Presenting substantively interpretable findings often also means using (and sometimes converting to) a scale that readers can easily understand. If your model reports results in log-odds, for example, converting the estimates to predicted probabilities will make it easier to interpret. All of this advice is quite general. Each of these points applies equally well to the presentation of summary results in a table rather than a graph. There is nothing distinctively graphical about putting the focus on the substantive meaning of your findings.

34.4.2 Show your degree of confidence

Much the same applies to presenting the degree of uncertainty or confidence you have in your results. Model estimates come with various measures of precision, confidence, credence, or significance. Presenting and interpreting these measures is notoriously prone to misinterpretation, or over-interpretation, as researchers and audiences both demand more from things like confidence intervals and p-values than these statistics can deliver. At a minimum, having decided on an appropriate measure of model fit or the right assessment of confidence, you should show their range when you present your results. A family of related `ggplot` geoms allow you to show a range or interval defined by position on the x-axis and then a `ymin` and `ymax` range on the y-axis. These geoms include `geom_pointrange()` and `geom_errorbar()`, which we will see in action shortly. A related geom, `geom_ribbon()` uses the same arguments to draw filled areas, and is useful for plotting ranges of y-axis values along some continuously varying x-axis.

34.4.3 Show your data when you can

Plotting the results from a multivariate model generally means one of two things. First, we can show what is in effect a table of coefficients with associated measures of confidence, perhaps organizing the coefficients into meaningful groups, or by the size of the predicted association, or both. Second, we can show the predicted values of some variables (rather than just a model's coefficients) across some range of interest. The latter approach lets us show the original data points if we wish. The way `ggplot` builds graphics layer by layer allows us to easily combine model estimates (e.g. a regression line and an associated range) and the underlying data. In effect these are manually-constructed versions of the automatically-generated plots that we have been producing with `geom_smooth()` since the beginning of this book.

34.5 Generate predictions to graph

Having fitted a model, then, we might want to get a picture of the estimates it produces over the range of some particular variable, holding other covariates constant at some sensible values. The `predict()` function is a generic way of using model objects to produce this kind of prediction. In R, “generic” functions take their inputs and pass them along to more specific functions behind the scenes, ones that are suited to working with the particular kind of model object we have. The details of getting predicted values from a **OLS** model, for instance, will be somewhat different from getting predictions out of a logistic regression. But in each case we can use the same `predict()` function, taking care to check the documentation to see what form the results are returned in for the kind of model we are working with. Many of the most commonly-used functions in R are generic in this way. The `summary()` function, for example, works on objects of many different classes, from vectors to data frames and statistical models, producing appropriate output in each case by way of a class-specific function in the background.

For `predict()` to calculate the new values for us, it needs some new data to fit the model to. We will generate a new data frame whose columns have the same names as the variables in the model's original data, but where the rows have new values. A very useful function called `expand.grid()` will help us do this. We will give it a list of variables, specifying the range of values we want each variable to take. Then `expand.grid()` will generate the will multiply out the full range of values for all combinations of the values we give it, thus creating a new data frame with the new data we need.

In the following bit of code, we use `min()` and `max()` to get the minimum and maximum values for per capita GDP, and then create a vector with one hundred evenly-spaced elements between the minimum and the maximum. We hold population constant at its median, and we let continent take all of its five available values.

```
min_gdp <- min(gapminder$gdpPercap)
max_gdp <- max(gapminder$gdpPercap)
med_pop <- median(gapminder$pop)
```

```

pred_df <- expand.grid(gdpPercap = seq(from = min_gdp,
                                         to = max_gdp,
                                         length.out = 100),
                        pop = med_pop,
                        continent = c("Africa", "Americas",
                                      "Asia", "Europe", "Oceania"))

dim(pred_df)
#> [1] 500   3

head(pred_df)
#>   gdpPercap    pop continent
#> 1     241 7023596    Africa
#> 2     1385 7023596    Africa
#> 3     2530 7023596    Africa
#> 4     3674 7023596    Africa
#> 5     4818 7023596    Africa
#> 6     5962 7023596    Africa

```

Now we can use `predict()`. If we give the function our new data and model, without any further argument, it will calculate the fitted values for every row in the data frame. If we specify `interval = 'predict'` as an argument, it will calculate 95% prediction intervals in addition to the point estimate.

```

pred_out <- predict(object = out,
                     newdata = pred_df,
                     interval = "predict")
head(pred_out)
#>   fit  lwr  upr
#> 1 48.0 31.5 64.4
#> 2 48.5 32.1 64.9
#> 3 49.0 32.6 65.4
#> 4 49.5 33.1 65.9
#> 5 50.0 33.6 66.4
#> 6 50.5 34.1 67.0

```

Because we know that, by construction, the cases in `pred_df` and `pred_out` correspond row for row, we can bind the two data frames together by column. This method of joining or merging tables is definitely not recommended when you are dealing with data.

```

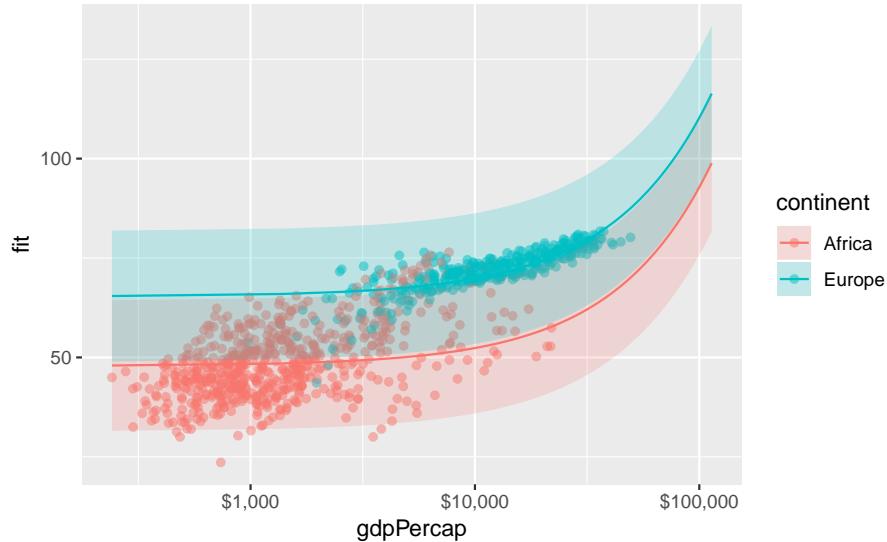
pred_df <- cbind(pred_df, pred_out)
head(pred_df)
#>   gdpPercap    pop continent   fit  lwr  upr
#> 1     241 7023596    Africa 48.0 31.5 64.4
#> 2     1385 7023596    Africa 48.5 32.1 64.9
#> 3     2530 7023596    Africa 49.0 32.6 65.4
#> 4     3674 7023596    Africa 49.5 33.1 65.9
#> 5     4818 7023596    Africa 50.0 33.6 66.4
#> 6     5962 7023596    Africa 50.5 34.1 67.0

```

The end result is a tidy data frame, containing the predicted values from the model for the range of values we specified. Now we can plot the results. Because we produced a full range of predicted values, we can decide whether or not to use all of them. Here we further subset the predictions to just those for Europe and Africa.

```
p <- ggplot(data = subset(pred_df, continent %in% c("Europe", "Africa")),
  aes(x = gdpPercap,
      y = fit, ymin = lwr, ymax = upr,
      color = continent,
      fill = continent,
      group = continent))

p + geom_point(data = subset(gapminder,
                             continent %in% c("Europe", "Africa")),
  aes(x = gdpPercap, y = lifeExp,
      color = continent),
  alpha = 0.5,
  inherit.aes = FALSE) +
  geom_line() +
  geom_ribbon(alpha = 0.2, color = FALSE) +
  scale_x_log10(labels = scales::dollar)
```



We use a new geom here to draw the area covered by the prediction intervals: `geom_ribbon()`. It takes an `x` argument like a line, but a `ymin` and `ymax` argument as specified in the `ggplot()` aesthetic mapping. This defines the lower and upper limits of the prediction interval.

In practice, you may not use `predict()` directly all that often. Instead, you might write code using additional libraries that encapsulate the process of producing predictions and plots from models. These are especially useful when your model is a little more complex and the interpretation of coefficients becomes trickier. This happens, for instance, when you have a binary outcome variable and need to convert the results of a logistic regression into predicted probabilities, or when you have interaction terms amongst your predictions. We will discuss some of these helper libraries in the next few sections. However, bear in mind that `predict()` and its ability to work safely with different classes of model underpins many of those libraries. So it's useful to see it in action first hand in order to understand what it is doing.

34.6 Tidy model objects with broom

The `predict` method is very useful, but there are a lot of other things we might want to do with our model output. We will use David Robinson's `broom` package to help us out. It is a library of functions that help us get from the model results that R generates to numbers that we can plot. It will take model objects and

turn pieces of them into data frames that you can use easily with `ggplot`.

```
library(broom)
```

Broom takes ggplot's approach to tidy data and extends it to the model objects that R produces. Its methods can tidily extract three kinds of information. First, we can see component-level information about aspects of the model itself, such as coefficients and t-statistics. Second, we can obtain observation-level information about the model's connection to the underlying data. This includes the fitted values and residuals for each observation in the data. And finally we can get model-level information that summarizes the fit as a whole, such as an F-statistic, the model deviance, or the r-squared. There is a `broom` function for each of these tasks.

34.6.1 Get component-level statistics with `tidy()`

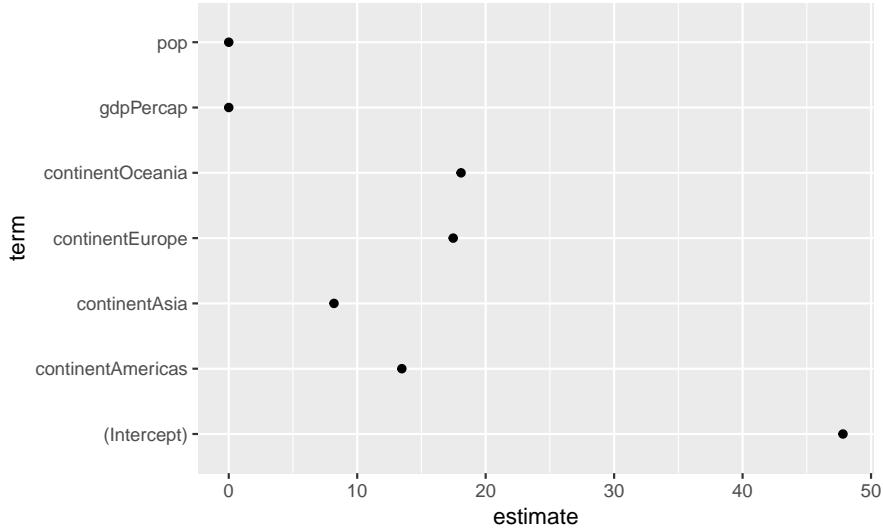
The `tidy()` function takes a model object and returns a data frame of component-level information. We can work with this to make plots in a familiar way, and much more easily than fishing inside the model object to extract the various terms. Here is an example, using the default results as just returned. For a more convenient display of the results, we will pipe the object we create with `tidy()` through a function that rounds the numeric columns of the data frame to two decimal places. This doesn't change anything about the object itself, of course.

```
out_comp <- tidy(out)
out_comp %>% round_df()
#> # A tibble: 7 x 5
#>   term      estimate std.error statistic p.value
#>   <chr>     <dbl>    <dbl>     <dbl>    <dbl>
#> 1 (Intercept) 47.8     0.34     141.      0
#> 2 gdpPerCap    0        0        19.2      0
#> 3 pop          0        0        3.33     0
#> 4 continentAmericas 13.5     0.6      22.5      0
#> 5 continentAsia   8.19    0.570    14.3      0
#> 6 continentEurope 17.5     0.62     28.0      0
#> # ... with 1 more row
```

We are now able to treat this dataframe just like all the other data that we have seen so far.

```
p <- ggplot(out_comp, mapping = aes(x = term,
                                      y = estimate))

p + geom_point() + coord_flip()
```



We can extend and clean up this plot in a variety of ways. For example, we can tell `tidy()` to calculate confidence intervals for the estimates, using R’s `confint()` function.

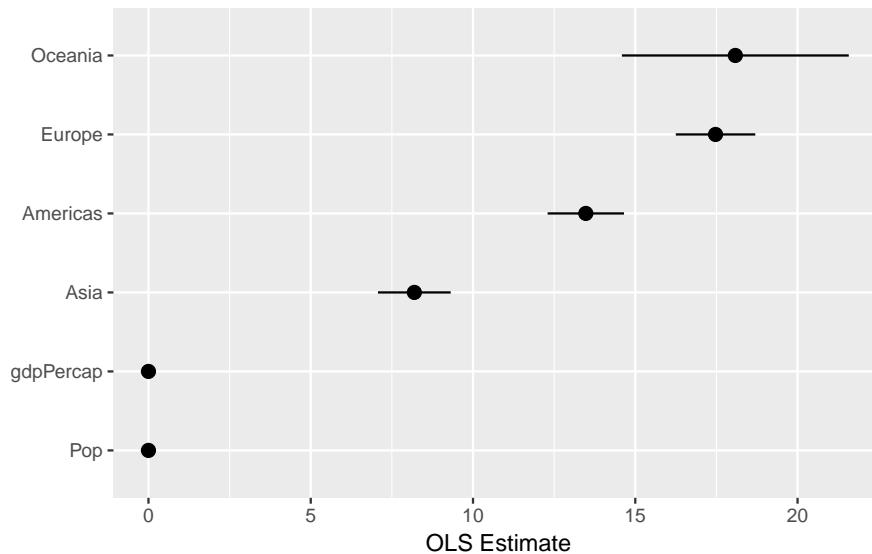
```
out_conf <- tidy(out, conf.int = TRUE)
out_conf %>% round_df()
#> # A tibble: 7 x 7
#>   term      estimate std.error statistic p.value conf.low conf.high
#>   <chr>     <dbl>    <dbl>     <dbl>    <dbl>    <dbl>    <dbl>
#> 1 (Intercept) 47.8     0.34     141.       0     47.2     48.5
#> 2 gdpPercap    0        0        19.2      0        0        0
#> 3 pop          0        0        3.33     0        0        0
#> 4 continentAmericas 13.5     0.6      22.5      0      12.3     14.6
#> 5 continentAsia  8.19    0.570     14.3      0      7.07     9.31
#> 6 continentEurope 17.5     0.62     28.0      0      16.2     18.7
#> # ... with 1 more row
```

The convenience “not in” operator `%nin%` is available via the `socviz` library. It does the opposite of `%in%` and selects only the items in a first vector of characters that are not in the second. We’ll use it to drop the intercept term from the table. We also want to something about the labels. When fitting a model with categorical variables, R will create coefficient names based on the variable name and the category name, like `continentAmericas`. Normally we like to clean these up before plotting. Most commonly, we just want to strip away the variable name at the beginning of the coefficient label. For this we can use `prefix_strip()`, a convenience function in the `socviz` library. We tell it which prefixes to drop, using it to create a new column variable in `out_conf` that corresponds to the `terms` column, but that has nicer labels.

```
out_conf <- subset(out_conf, term %nin% "(Intercept)")
out_conf$nicelabs <- prefix_strip(out_conf$term, "continent")
```

Now we can use `geom_pointrange()` to make a figure that displays some information about our confidence in the variable estimates, as opposed to just the coefficients. As with the boxplots earlier, we use `reorder()` to sort the names of the model’s terms by the `estimate` variable, thus arranging our plot of effects from largest to smallest in magnitude.

```
p <- ggplot(out_conf, mapping = aes(x = reorder(nicelabs, estimate),
                                      y = estimate, ymin = conf.low, ymax = conf.high))
p + geom_pointrange() + coord_flip() + labs(x="", y="OLS Estimate")
```



Dotplots of this kind can be very compact. The vertical axis can often be compressed quite a bit, with no loss in comprehension. In fact, they are often easier to read with much less room between the rows than given by a default square shape.

34.6.2 Get observation-level statistics with `augment()`

The values returned by `augment()` are all statistics calculated at the level of the original observations. As such, they can be added on to the data frame that the model is based on. Working from a call to `augment()` will return a data frame with all the original observations used in the estimation of the model, together with columns like the following:

- `.fitted` — The fitted values of the model.
- `.se.fit` — The standard errors of the fitted values.
- `.resid` — The residuals.
- `.hat` — The diagonal of the hat matrix.
- `.sigma` — An estimate of residual standard deviation when the corresponding observation is dropped from the model.
- `.cooksdist` — Cook's distance, a common regression diagnostic; and
- `.std.resid` — The standardized residuals.

Each of these variables is named with a leading `.`, for example `.hat` rather than `hat`, and so on. This is to guard against accidentally confusing it with (or accidentally overwriting) an existing variable in your data with this name. The columns of values return will differ slightly depending on the class of model being fitted.

```
out_aug <- augment(out)
head(out_aug) %>% round_df()
#> # A tibble: 6 x 11
#>   lifeExp gdpPercap    pop continent .fitted .se.fit .resid .hat .sigma
#>   <dbl>     <dbl>    <dbl> <fct>      <dbl>    <dbl>    <dbl> <dbl>    <dbl>
#> 1  28.8      779. 8.43e6 Asia       56.4     0.47   -27.6     0   8.34
#> 2  30.3      821. 9.24e6 Asia       56.4     0.47   -26.1     0   8.34
#> 3   32        853. 1.03e7 Asia      56.5     0.47   -24.5     0   8.35
#> 4  34.0      836. 1.15e7 Asia      56.5     0.47   -22.4     0   8.35
#> 5  36.1      740. 1.31e7 Asia      56.4     0.47   -20.3     0   8.35
#> 6  38.4      786. 1.49e7 Asia      56.5     0.47   -18.0     0   8.36
```

```
#> # ... with 2 more variables: .cooksrd <dbl>, .std.resid <dbl>
```

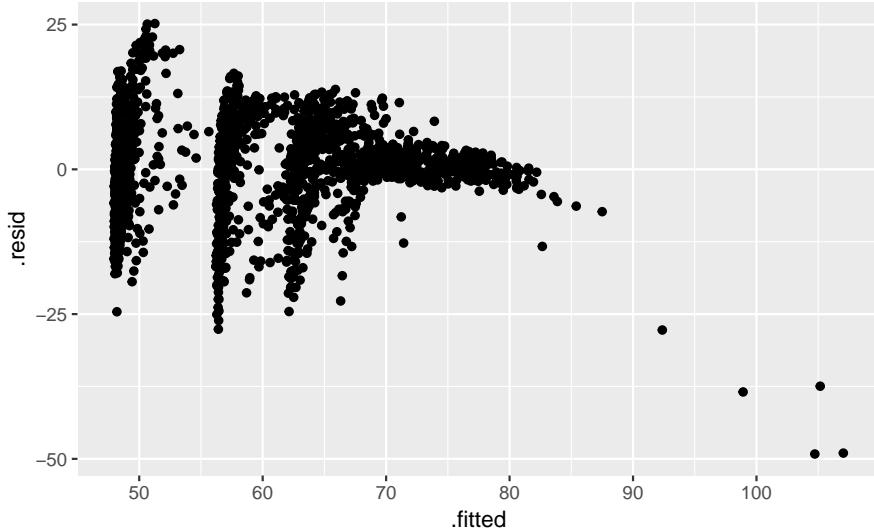
By default, `augment()` will extract the available data from the model object. This will usually include the variables used in the model itself, but not any additional ones contained in the original data frame. Sometimes it is useful to have these. We can add them by specifying the `data` argument:

```
out_aug <- augment(out, data = gapminder)
head(out_aug) %>% round_df()
#> # A tibble: 6 x 13
#>   country continent year lifeExp     pop gdpPercap .fitted .se.fit .resid
#>   <fct>    <fct>   <dbl>   <dbl>   <dbl>      <dbl>   <dbl>   <dbl>
#> 1 Afghan~ Asia     1952     28.8 8.43e6     779.    56.4    0.47  -27.6
#> 2 Afghan~ Asia     1957     30.3 9.24e6     821.    56.4    0.47  -26.1
#> 3 Afghan~ Asia     1962      32  1.03e7     853.    56.5    0.47  -24.5
#> 4 Afghan~ Asia     1967     34.0 1.15e7     836.    56.5    0.47  -22.4
#> 5 Afghan~ Asia     1972     36.1 1.31e7     740.    56.4    0.47  -20.3
#> 6 Afghan~ Asia     1977     38.4 1.49e7     786.    56.5    0.47  -18.0
#> # ... with 4 more variables: .hat <dbl>, .sigma <dbl>, .cooksrd <dbl>,
#> #   .std.resid <dbl>
```

If some rows containing missing data were dropped to fit the model, then these will not be carried over to the augmented dataframe.

The new columns created by `augment()` can be used to create some standard regression plots. For example, we can plot the residuals versus the fitted values. Figure 6.7 suggests, unsurprisingly, that our country-year data has rather more structure than is captured by our OLS model.

```
p <- ggplot(data = out_aug,
             mapping = aes(x = .fitted, y = .resid))
p + geom_point()
```



34.6.3 Get model-level statistics with `glance()`

This function organizes the information typically presented at the bottom of a model's `summary()` output. By itself, it usually just returns a table with a single row in it. But as we shall see in a moment, the real

power of broom's approach is the way that it can scale up to cases where we are grouping or subsampling our data.

```
glance(out) %>% round_df()
#> # A tibble: 1 x 11
#>   r.squared adj.r.squared sigma statistic p.value    df logLik     AIC
#>       <dbl>        <dbl> <dbl>    <dbl> <dbl> <dbl> <dbl>
#> 1      0.580        0.580  8.37    394.     0     7 -6034. 12084.
#> # ... with 3 more variables: BIC <dbl>, deviance <dbl>, df.residual <dbl>
```

Broom is able to `tidy` (and `augment`, and `glance` at) a wide range of model types. Not all functions are available for all classes of model. Consult broom's documentation for more details on what is available. For example, here is a plot created from the tidied output of an event-history analysis. First we generate a Cox proportional hazards model of some `survival` data.

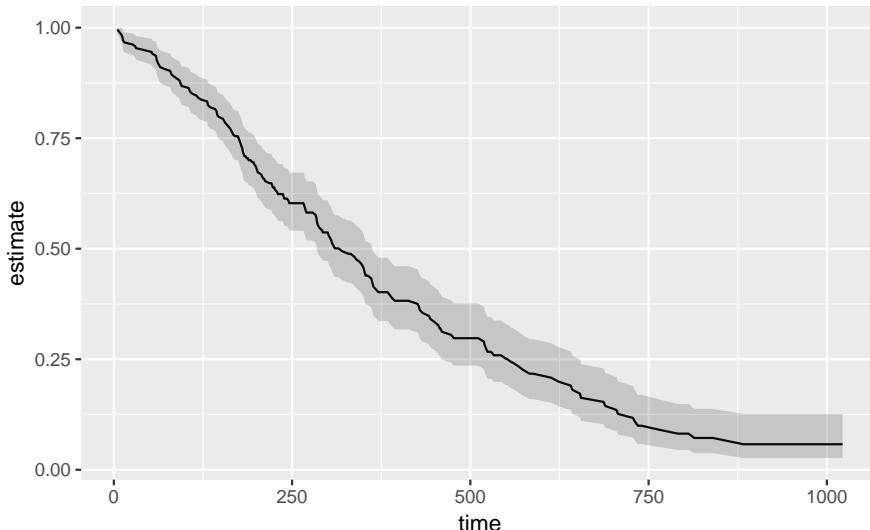
```
library(survival)
#>
#> Attaching package: 'survival'
#> The following object is masked from 'package:quantreg':
#>
#>     untangle.specials

out_cph <- coxph(Surv(time, status) ~ age + sex, data = lung)
out_surv <- survfit(out_cph)
```

The details of the fit are not important here, but in the first step the `Surv()` function creates the response or outcome variable for the proportional hazards model that is then fitted by the `coxph()` function. Then the `survfit()` function creates the survival curve from the model, much like we used `predict()` to generate predicted values earlier. Try `summary(out_cph)` to see the model, and `summary(out_surv)` to see the table of predicted values that will form the basis for our plot. Next we tidy `out_surv` to get a data frame, and plot it.

```
# Figure 6.8: A Kaplan-Meier plot.
out_tidy <- tidy(out_surv)

p <- ggplot(data = out_tidy, mapping = aes(time, estimate))
p + geom_line() +
  geom_ribbon(mapping = aes(ymin = conf.low, ymax = conf.high), alpha = .2)
```



34.7 Grouped analysis and list-columns

Broom makes it possible to quickly fit models to different subsets of your data and get consistent and usable tables of results out the other end. For example, let's say we wanted to look at the gapminder data by examining the relationship between life expectancy and GDP by continent, for each year in the data.

The `gapminder` data is at bottom organized by country-years. That is the unit of observation in the rows. If we wanted, we could take a slice of the data manually, such as “all countries observed in Asia, in 1962” or “all in Africa, 2002”. Here is “Europe, 1977”:

```
eu77 <- gapminder %>% filter(continent == "Europe", year == 1977)
```

We could then see what the relationship between life expectancy and GDP looked like for that continent-year group:

```
fit <- lm(lifeExp ~ log(gdpPercap), data = eu77)
summary(fit)

#>
#> Call:
#> lm(formula = lifeExp ~ log(gdpPercap), data = eu77)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -7.496 -1.031  0.093  1.176  3.712
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 29.489     7.161   4.12  0.00031 ***
#> log(gdpPercap) 4.488     0.756   5.94  2.2e-06 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 2.11 on 28 degrees of freedom
#> Multiple R-squared:  0.557, Adjusted R-squared:  0.541
#> F-statistic: 35.2 on 1 and 28 DF,  p-value: 2.17e-06
```

With `dplyr` and `broom` we can do this for every continent-year slice of the data in a compact and tidy way. We start with our table of data, and then (`%>%`) group the countries by continent and year using the `group_by()` function. We introduced this grouping operation in Chapter 4. Our data is reorganized first by continent, and within continent by year. Here we will take one further step and nest the data that make up each group:

```
out_le <- gapminder %>%
  group_by(continent, year) %>%
  nest()

out_le
#> # A tibble: 60 x 3
#>   continent  year   data
#>   <fct>     <int> <list>
#> 1 Asia       1952 <tibble [33 x 4]>
#> 2 Asia       1957 <tibble [33 x 4]>
#> 3 Asia       1962 <tibble [33 x 4]>
#> 4 Asia       1967 <tibble [33 x 4]>
```

```
#> 5 Asia      1972 <tibble [33 x 4]>
#> 6 Asia      1977 <tibble [33 x 4]>
#> # ... with 54 more rows
```

Think of what `nest()` does as a more intensive version what `group_by()` does. The resulting object is has the tabular form we expect (it is a tibble) but it looks a little unusual. The first two columns are the familiar continent and year. But we now also have a new column, data, that contains a small table of data corresponding to each continent-year group. This is a list-column, something we have not seen before. It turns out to be very useful for bundling together complex objects (structured, in this case, as a list of tibbles, each being a 33x4 table of data) within the rows of our data (which remains tabular). Our “Europe 1977” fit is in there. We can look at it, if we like, by filtering the data and then unnesting the list column.

```
out_le %>% filter(continent == "Europe" & year == 1977) %>% unnest()
#> # A tibble: 30 x 6
#>   continent year country           lifeExp     pop gdpPercap
#>   <fct>     <int> <fct>          <dbl>    <int>    <dbl>
#> 1 Europe     1977 Albania        68.9  2509048   3533.
#> 2 Europe     1977 Austria       72.2  7568430  19749.
#> 3 Europe     1977 Belgium       72.8  9821800  19118.
#> 4 Europe     1977 Bosnia and Herzegovina 69.9  4086000  3528.
#> 5 Europe     1977 Bulgaria      70.8  8797022  7612.
#> 6 Europe     1977 Croatia       70.6  4318673 11305.
#> # ... with 24 more rows
```

List-columns are useful because we can act on them in a compact and tidy way. In particular, we can pass functions along to each row of the list-column and make something happen. For example, a moment ago we ran a regression of life expectancy and logged GDP for European countries in 1977. We can do that for every continent-year combination in the data. We first create a convenience function called `fit_ols()` that takes a single argument, `df` (for data frame) and that fits the linear model we are interested in. Then we map that function to each of our list-column rows in turn. Recall from Chapter 4 that `mutate` creates new variables or columns on the fly within a pipeline.

The `map` action is an important idea in functional programming. If you have written code in other, more imperative languages you can think of it as a compact alternative to writing `for ... next` loops. You can of course write loops like this in R. Computationally they are often not any less efficient than their functional alternatives. But mapping functions to arrays is more easily integrated into a sequence of data transformations.

```
fit_ols <- function(df) {
  lm(lifeExp ~ log(gdpPercap), data = df)
}

out_le <- gapminder %>%
  group_by(continent, year) %>%
  nest() %>%
  mutate(model = map(data, fit_ols))

out_le
#> # A tibble: 60 x 4
#>   continent year data           model
#>   <fct>     <int> <list>         <list>
#> 1 Asia       1952 <tibble [33 x 4]> <lm>
#> 2 Asia       1957 <tibble [33 x 4]> <lm>
#> 3 Asia       1962 <tibble [33 x 4]> <lm>
#> 4 Asia       1967 <tibble [33 x 4]> <lm>
```

```
#> 5 Asia      1972 <tibble [33 x 4]> <lm>
#> 6 Asia      1977 <tibble [33 x 4]> <lm>
#> # ... with 54 more rows
```

Before starting the pipeline we create a new function: It is a convenience function whose only job is to estimate a particular OLS model on some data. Like almost everything in R, functions are a kind of object. To make a new one, we use the slightly special `function()` function. (Nerds love that sort of thing.) There is a little more detail on creating functions in the Appendix. To see what `fit_ols()` looks like once it is created, type `fit_ols` without parentheses at the Console. To see what it does, try `fit_ols(df = gapminder)`, or `summary(fit_ols(gapminder))`.

Now we have two list-columns: `data`, and `model`. The latter was created by mapping the `fit_ols()` function to each row of data. Inside each element of `model` is a linear model for that continent-year. So we now have sixty OLS fits, one for every continent-year grouping. Having the models inside the list column is not much use to us in and of itself. But we can extract the information we want while keeping things in a tidy tabular form. For clarity we will run the pipeline from the beginning again, this time adding a few new steps.

First we extract summary statistics from each model by mapping the `tidy()` function from `broom` to the model list column. Then we unnest the result, dropping the other columns in the process. Finally, we filter out all the Intercept terms, and also drop all observations from Oceania. In the case of the Intercepts we do this just out of convenience. Oceania we drop just because there are so few observations. We put the results in an object called `out_tidy`.

```

fit_ols <- function(df) {
  lm(lifeExp ~ log(gdpPercap), data = df)
}

out_tidy <- gapminder %>%
  group_by(continent, year) %>%
  nest() %>%
  mutate(model = map(data, fit_ols),
         tidied = map(model, tidy)) %>%
  unnest(tidied, .drop = TRUE) %>%
  filter(term %in% "(Intercept)" &
         continent %in% "Oceania"))

out_tidy %>% sample_n(5)
#> # A tibble: 5 x 7
#>   continent  year term      estimate std.error statistic    p.value
#>   <fct>     <int> <chr>        <dbl>     <dbl>      <dbl>      <dbl>
#> 1 Americas    1992 log(gdpPercap)  6.06      0.895     6.77 0.000000664
#> 2 Europe      2002 log(gdpPercap)  3.74      0.445     8.40 0.00000000391
#> 3 Asia         2007 log(gdpPercap)  5.16      0.694     7.43 0.0000000226
#> 4 Americas    1952 log(gdpPercap) 10.4       2.72      3.84 0.000827
#> 5 Americas    1957 log(gdpPercap) 10.3       2.40      4.31 0.000261

```

We now have tidy regression output with an estimate of the association between log GDP per capita and life expectancy for each year, within continents. We can plot these estimates in a way that takes advantage of their groupiness.

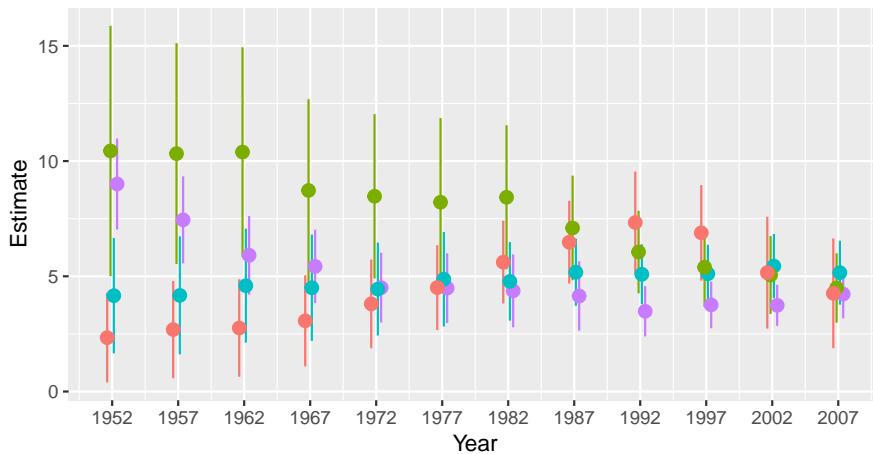
```

    ymax = estimate + 2*std.error,
    group = continent, color = continent))

p + geom_pointrange(position = position_dodge(width = 1)) +
  scale_x_continuous(breaks = unique(gapminder$year)) +
  theme(legend.position = "top") +
  labs(x = "Year", y = "Estimate", color = "Continent")

```

Continent Africa Americas Asia Europe



The call to `position_dodge()` within `geom_pointrange()` allows the point ranges for each continent to be near each other within years, instead of being plotted right on top of one another. We could have faceted the results by continent, but doing it this way lets us see differences in the yearly estimates much more easily. This technique is very useful not just for cases like this, but also when you want to compare the coefficients given by different kinds of statistical model. This sometimes happens when we're interested in seeing how, say, OLS performs against some other model specification.

34.8 Plot marginal effects

Our earlier discussion of `predict()` was about obtaining estimates of the average effect of some coefficient, net of the other terms in the model. Over the past decade, estimating and plotting partial or marginal effects from a model has become an increasingly common way of presenting accurate and interpretively useful predictions. Interest in marginal effects plots was stimulated by the realization that the interpretation of terms in logistic regression models, in particular, was trickier than it seemed—especially when there were interaction terms in the model (Ai & Norton, 2003). Thomas Leeper's `margins` package can make these plots for us.

```
library(margins)
```

To see it in action, we'll take another look at the General Social Survey data in `gss_sm`, this time focusing on the binary variable, `obama`. As is common with retrospective questions on elections, rather more people claim to have voted for Obama than is consistent with the vote share he received in the election. It is coded 1 if the respondent said they voted for Barack Obama in the 2012 presidential election, and 0 otherwise. In this case, mostly for convenience here, the zero code includes all other answers to the question, including those who said they voted for Mitt Romney, those who said they did not vote, those who refused to answer, and those who said they didn't know who they voted for. We will fit a logistic regression on `obama`, with `age`, `polviews`, `race`, and `sex` as the predictors. The `age` variable is the respondent's age in years. The `sex` variable is coded as "Male" or "Female" with "Male" as the reference category. The `race` variable is coded

as “White”, “Black”, or “Other” with “White” as the reference category. The polviews measure is a self-reported scale of the respondent’s political orientation from “Extremely Conservative” through “Extremely Liberal”, with “Moderate” in the middle. We take `polviews` and create a new variable, `polviews_m`, using the `relevel()` function to recode “Moderate” to be the reference category. We fit the model with the `glm()` function, and specify an interaction between race and sex.

```
gss_sm$polviews_m <- relevel(gss_sm$polviews, ref = "Moderate")

out_bo <- glm(obama ~ polviews_m + sex*race,
               family = "binomial", data = gss_sm)
summary(out_bo)
#>
#> Call:
#> glm(formula = obama ~ polviews_m + sex * race, family = "binomial",
#>      data = gss_sm)
#>
#> Deviance Residuals:
#>    Min      1Q  Median      3Q     Max
#> -2.905  -0.554   0.177   0.542   2.244
#>
#> Coefficients:
#>                               Estimate Std. Error z value Pr(>|z|)
#> (Intercept)                 0.29649   0.13409   2.21   0.0270 *
#> polviews_mExtremely Liberal 2.37295   0.52504   4.52   6.2e-06 ***
#> polviews_mLiberal           2.60003   0.35667   7.29   3.1e-13 ***
#> polviews_mSlightly Liberal  1.29317   0.24843   5.21   1.9e-07 ***
#> polviews_mSlightly Conservative -1.35528  0.18129  -7.48   7.7e-14 ***
#> polviews_mCConservative    -2.34746  0.20038  -11.71  < 2e-16 ***
#> polviews_mExtremely Conservative -2.72738  0.38721  -7.04   1.9e-12 ***
#> sexFemale                   0.25487   0.14537   1.75   0.0796 .
#> raceBlack                    3.84953   0.50132   7.68   1.6e-14 ***
#> raceOther                    -0.00214  0.43576   0.00   0.9961
#> sexFemale:raceBlack          -0.19751  0.66007  -0.30   0.7648
#> sexFemale:raceOther          1.57483   0.58766   2.68   0.0074 **
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> (Dispersion parameter for binomial family taken to be 1)
#>
#> Null deviance: 2247.9 on 1697 degrees of freedom
#> Residual deviance: 1345.9 on 1686 degrees of freedom
#> (1169 observations deleted due to missingness)
#> AIC: 1370
#>
#> Number of Fisher Scoring iterations: 6
```

The summary reports the coefficients and other information. We can now graph the data in any one of several ways. Using `margins()` we calculate the marginal effects for each variable:

```
bo_m <- margins(out_bo)
summary(bo_m)
#>                               factor      AME       SE        z      p    lower
#> polviews_mConservative -0.4119 0.0283 -14.5394 0.0000 -0.4674
#> polviews_mEExtremely Conservative -0.4538 0.0420 -10.7971 0.0000 -0.5361
#> polviews_mEExtremely Liberal    0.2681 0.0295   9.0996 0.0000  0.2103
```

```
#>          polviews_mLiberal  0.2768 0.0229  12.0736 0.0000  0.2319
#>  polviews_mSlightly Conservative -0.2658 0.0330  -8.0596 0.0000 -0.3304
#>          polviews_mSlightly Liberal  0.1933 0.0303   6.3896 0.0000  0.1340
#>          raceBlack  0.4032 0.0173  23.3568 0.0000  0.3694
#>          raceOther  0.1247 0.0386   3.2297 0.0012  0.0490
#>          sexFemale  0.0443 0.0177   2.5073 0.0122  0.0097
#>          upper
#>  -0.3564
#>  -0.3714
#>  0.3258
#>  0.3218
#>  -0.2011
#>  0.2526
#>  0.4371
#>  0.2005
#>  0.0789
```

The `margins` library comes with several plot methods of its own. If you wish, at this point you can just try `plot(bo_m)` to see a plot of the average marginal effects, produced with the general look of a Stata graphic. Other plot methods in the `margins` library include `cplot()`, which visualizes marginal effects conditional on a second variable, and `image()`, which shows predictions or marginal effects as a filled heatmap or contour plot.

Alternatively, we can take results from `margins()` and plot them ourselves. To clean up the summary a little a little, we convert it to a tibble, then use `prefix_strip()` and `prefix_replace()` to tidy the labels. We want to strip the `polviews_m` and `sex` prefixes, and (to avoid ambiguity about “Other”), adjust the `race` prefix.

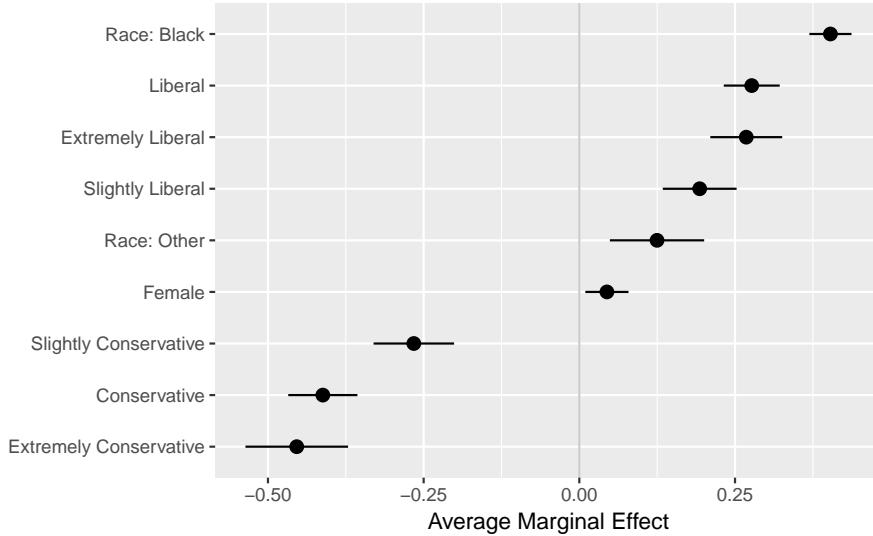
```
bo_gg <- as_tibble(summary(bo_m))
prefixes <- c("polviews_m", "sex")
bo_gg$factor <- prefix_strip(bo_gg$factor, prefixes)
bo_gg$factor <- prefix_replace(bo_gg$factor, "race", "Race: ")

bo_gg %>% select(factor, AME, lower, upper)
#> # A tibble: 9 x 4
#>   factor           AME   lower  upper
#>   <chr>          <dbl>  <dbl>  <dbl>
#> 1 Conservative    -0.412 -0.467 -0.356
#> 2 Extremely Conservative -0.454 -0.536 -0.371
#> 3 Extremely Liberal     0.268  0.210  0.326
#> 4 Liberal          0.277  0.232  0.322
#> 5 Slightly Conservative -0.266 -0.330 -0.201
#> 6 Slightly Liberal      0.193  0.134  0.253
#> # ... with 3 more rows
```

Now we have a table that we can plot as we have learned:

```
p <- ggplot(data = bo_gg, aes(x = reorder(factor, AME),
                                y = AME, ymin = lower, ymax = upper))

p + geom_hline(yintercept = 0, color = "gray80") +
  geom_pointrange() + coord_flip() +
  labs(x = NULL, y = "Average Marginal Effect")
```

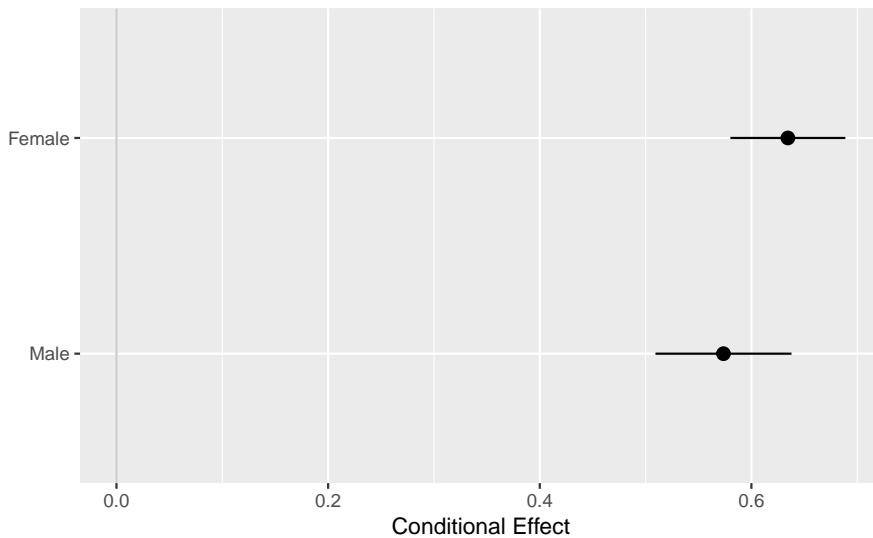


If we are just interested in getting conditional effects for a particular variable, then conveniently we can ask the plot methods in the margins library to do the work calculating effects for us but without drawing their plot. Instead, they can return the results in a format we can easily use in ggplot, and with less need for clean up, for the clean-up. For example, with `cplot()`:

```
pv_cp <- cplot(out_bo, x = "sex", draw = FALSE)
#>   xvals yvals upper lower
#> 1   Male 0.574 0.638 0.509
#> 2 Female 0.634 0.689 0.580

p <- ggplot(data = pv_cp, aes(x = reorder(xvals, yvals),
                               y = yvals, ymin = lower, ymax = upper))

p + geom_hline(yintercept = 0, color = "gray80") +
  geom_pointrange() + coord_flip() +
  labs(x = NULL, y = "Conditional Effect")
```



The `margins` package is under active development. It can do much more than described here. The vignettes that come with the package provide more extensive discussion and numerous examples.

34.9 Plots from complex surveys

Social scientists often work with data collected using a complex survey design. Survey instruments may be stratified by region or some other characteristic, contain replicate weights to make them comparable to a reference population, have a clustered structure, and so on. In Chapter 4 we learned how calculate and then plot frequency tables of categorical variables, using some data from the General Social Survey (GSS). However, if we want accurate estimates of US households from the GSS, we will need to take the survey's design into account, and use the survey weights provided in the dataset. Thomas Lumley's **survey** library provides a comprehensive set of tools for addressing these issues. The tools and the theory behind them are discussed in detail in Lumley (2010), and an overview of the package is provided in Lumley (2004). While the functions in the **survey** package are straightforward to use and return results in a generally tidy form, the package predates the tidyverse and its conventions by several years. This means we cannot use survey functions directly with **dplyr**. However, Greg Freedman Ellis has written a helper package, **srvyr**, that solves this problem for us, and lets us use the **survey** library's functions within a data analysis pipeline in a familiar way.

For example, the **gss_lon** data contains a small subset of measures from every wave of the GSS since its inception in 1972. It also contains several variables that describe the design of the survey and provide replicate weights for observations in various years. These technical details are described in the GSS documentation. Similar information is typically provided by other complex surveys. Here we will use this design information to calculate weighted estimates of the distribution of educational attainment by race, for selected survey years from 1976 to 2016.

To begin, we load the **survey** and **srvyr** libraries.

```
library(survey)
#> Loading required package: grid
#> Loading required package: Matrix
#>
#> Attaching package: 'Matrix'
#> The following object is masked from 'package:tidyverse':
#>
#>     expand
#>
#> Attaching package: 'survey'
#> The following object is masked from 'package:graphics':
#>
#>     dotchart
library(srvyr)
#>
#> Attaching package: 'srvyr'
#> The following object is masked from 'package:stats':
#>
#>     filter
```

Next, we take our **gss_lon** dataset and use the survey tools to create a new object that contains the data, as before, but with some additional information about the survey's design:

```
options(survey.lonely.psu = "adjust")
options(na.action="na.pass")

gss_wt <- subset(gss_lon, year > 1974) %>%
  mutate(stratvar = interaction(year, vstrat)) %>%
  as_survey_design(ids = vpsu,
```

```
strata = stratvar,
weights = wtssall,
nest = TRUE)
```

The two options set at the beginning provide some information to the survey library about how to behave. You should consult Lumley (2010) and the survey package documentation for details. The subsequent operations create `gss_wt`, an object with one additional column (`stratvar`), describing the yearly sampling strata. We use the `interaction()` function to do this. It multiplies the `vstrat` variable by the `year` variable to get a vector of stratum information for each year. We have to do this because of the way the GSS codes its stratum information. In the next step, we use the `as_survey_design()` function to add the key pieces of information about the survey design. It adds information about the sampling identifiers (`ids`), the strata (`strata`), and the replicate weights (`weights`). With those in place we can take advantage of a large number of specialized functions in the survey library that allow us to calculate properly weighted survey means or estimate models with the correct sampling specification. For example, we can easily calculate the distribution of education by race for a series of years from 1976 to 2016. We use `survey_mean()` to do this:

```
out_grp <- gss_wt %>%
  filter(year %in% seq(1976, 2016, by = 4)) %>%
  group_by(year, race, degree) %>%
  summarize(prop = survey_mean(na.rm = TRUE))
#> Warning: Factor `degree` contains implicit NA, consider using
#> `forcats::fct_explicit_na`

out_grp
#> # A tibble: 150 x 5
#>   year race   degree      prop    prop_se
#>   <dbl> <fct> <fct>     <dbl>    <dbl>
#> 1 1976 White Lt High School 0.328  0.0160
#> 2 1976 White High School   0.518  0.0162
#> 3 1976 White Junior College 0.0129 0.00298
#> 4 1976 White Bachelor     0.101  0.00960
#> 5 1976 White Graduate     0.0393 0.00644
#> 6 1976 Black Lt High School 0.562  0.0611
#> # ... with 144 more rows
```

The results returned in `out_grp` include standard errors. We can also ask `survey_mean()` to calculate confidence intervals for us, if we wish.

Grouping with `group_by()` lets us calculate counts or means for the innermost variable, grouped by the next variable “up” or “out”, in this case, degree by race, such that the proportions for degree will sum to one for each group in race, and this will be done separately for each value of year. If we want the marginal frequencies, such that the values for all combinations of race and degree sum to one within each year, we first have to interact the variables we are cross-classifying. Then we group by the new interacted variable and do the calculation as before:

```
out_mrg <- gss_wt %>%
  filter(year %in% seq(1976, 2016, by = 4)) %>%
  mutate(racedeg = interaction(race, degree)) %>%
  group_by(year, racedeg) %>%
  summarize(prop = survey_mean(na.rm = TRUE))
#> Warning: Factor `racedeg` contains implicit NA, consider using
#> `forcats::fct_explicit_na`

out_mrg
```

```
#> # A tibble: 150 x 4
#>   year racedeg      prop  prop_se
#>   <dbl> <fct>        <dbl>    <dbl>
#> 1 1976 White.Lt High School 0.298  0.0146
#> 2 1976 Black.Lt High School 0.0471 0.00840
#> 3 1976 Other.Lt High School 0.00195 0.00138
#> 4 1976 White.High School 0.471  0.0160
#> 5 1976 Black.High School 0.0283 0.00594
#> 6 1976 Other.High School 0.00325 0.00166
#> # ... with 144 more rows
```

This gives us the numbers that we want and returns them in a tidy data frame. The `interaction()` function produces variable labels that are a compound of the two variables we interacted, with each combination of categories separated by a period, (such as White.Graduate. However, perhaps we would like to see these categories as two separate columns, one for race and one for education, as before. Because the variable labels are organized in a predictable way, we can use one of the convenient functions in the tidyverse’s `tidyverse` library to separate the single variable into two columns while correctly preserving the row values. Appropriately, this function is called `separate()`.

```

out_mrg <- gss_wt %>%
  filter(year %in% seq(1976, 2016, by = 4)) %>%
  mutate(racedeg = interaction(race, degree)) %>%
  group_by(year, racedeg) %>%
  summarize(prop = survey_mean(na.rm = TRUE)) %>%
  separate(racedeg, sep = "\\.", into = c("race", "degree"))
#> Warning: Factor `racedeg` contains implicit NA, consider using
#> `forcats::fct_explicit_na`

out_mrg
#> # A tibble: 150 x 5
#>   year race  degree          prop    prop_se
#>   <dbl> <chr> <chr>        <dbl>    <dbl>
#> 1 1976 White Lt High School 0.298    0.0146
#> 2 1976 Black Lt High School 0.0471   0.00840
#> 3 1976 Other Lt High School 0.00195  0.00138
#> 4 1976 White High School   0.471    0.0160
#> 5 1976 Black High School   0.0283   0.00594
#> 6 1976 Other High School  0.00325  0.00166
#> # ... with 144 more rows

```

The call to `separate()` says to take the `racedeg` column, split each value when it sees a period, and reorganize the results into two columns, `race` and `degree`. This gives us a tidy table much like `out_grp`, but for the marginal frequencies.

Reasonable people can disagree over how best to plot a small multiple of a frequency table while faceting by year, especially when there is some measure of uncertainty attached. A barplot is the obvious approach for a single case, but when there are many years it can become difficult to compare bars across panels. This is especially the case when standard errors or confidence intervals are used in conjunction with bars. Sometimes it may be preferable to show that the underlying variable is categorical, as a bar chart makes clear, and not continuous, as a line graph suggests. Here the trade-off is in favor of the line graphs as the bars are very hard to compare across facets. This is sometimes called a “dynamite plot”, not because it looks amazing but because the t-shaped error bars on the tops of the columns make them look like cartoon dynamite plungers. An alternative is to use a line graph to join up the time observations, faceting on educational categories instead of year. Figure 6.12 shows the results for our GSS data in *dynamite-plot* form, where the error bars are defined as twice the standard error in either direction around the point estimate.

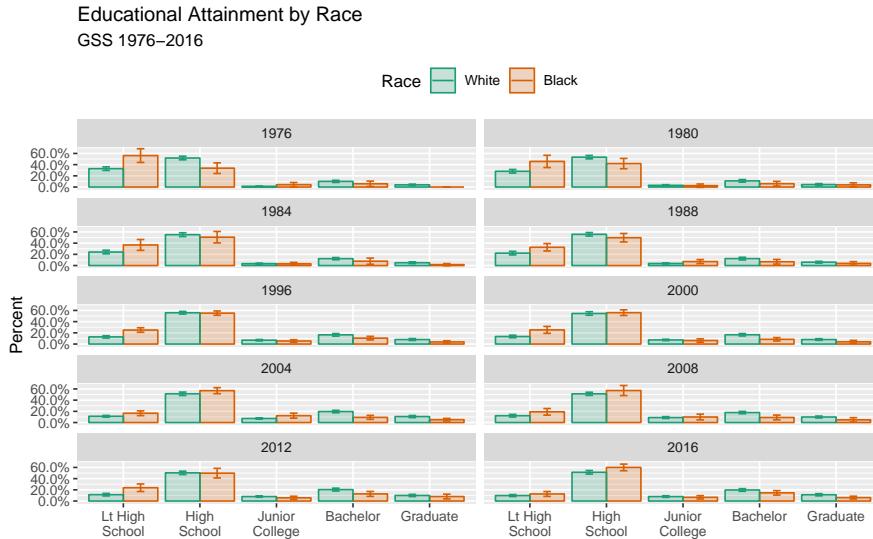
```

p <- ggplot(data = subset(out_grp, race %in% "Other"),
             mapping = aes(x = degree, y = prop,
                           ymin = prop - 2*prop_se,
                           ymax = prop + 2*prop_se,
                           fill = race,
                           color = race,
                           group = race))

dodge <- position_dodge(width=0.9)

p + geom_col(position = dodge, alpha = 0.2) +
  geom_errorbar(position = dodge, width = 0.2) +
  scale_x_discrete(labels = scales::wrap_format(10)) +
  scale_y_continuous(labels = scales::percent) +
  scale_color_brewer(type = "qual", palette = "Dark2") +
  scale_fill_brewer(type = "qual", palette = "Dark2") +
  labs(title = "Educational Attainment by Race",
       subtitle = "GSS 1976–2016",
       fill = "Race",
       color = "Race",
       x = NULL, y = "Percent") +
  facet_wrap(~ year, ncol = 2) +
  theme(legend.position = "top")

```



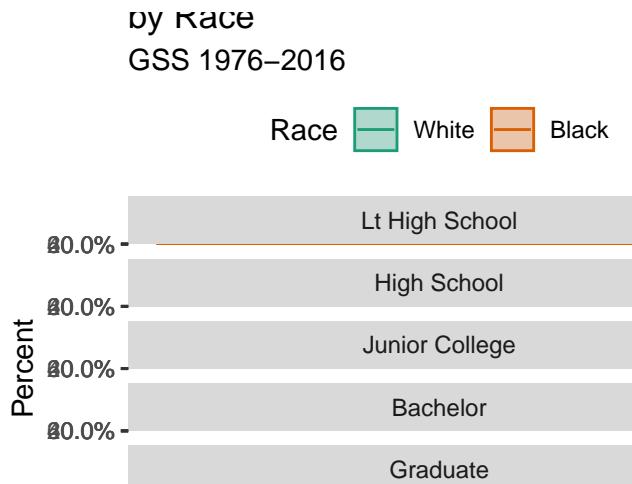
This plot has a few cosmetic details and adjustments that we will learn more about in Chapter 8. As before, I encourage you to peel back the plot from the bottom, one instruction at a time, to see what changes. One useful adjustment to notice is the new call to the `scales` library to adjust the labels on the x-axis. The adjustment on the y-axis is familiar, `scales::percent` to convert the proportion to a percentage. On the x-axis, the issue is that several of the labels are rather long. If we do not adjust them they will print over one another. The `scales::wrap_format()` function will break long labels into lines. It takes a single numerical argument (here 10) that is the maximum length a string can be before it is wrapped onto a new line.

Faceting by education instead. Figure 6.13: Faceting by education instead. A graph like this is true to the categorical nature of the data, while showing the breakdown of groups within each year. But you should experiment with some alternatives. For example, we might decide that it is better to facet by degree category instead, and put the year on the x-axis within each panel. If we do that, then we can use

`geom_line()` to show a time trend, which is more natural, and `geom_ribbon()` to show the error range. This is perhaps a better way to show the data, especially as it brings out the time trends within each degree category, and allows us to see the similarities and differences by racial classification at the same time.

```
p <- ggplot(data = subset(out_grp, race %in% "Other"),
             mapping = aes(x = year, y = prop, ymin = prop - 2*prop_se,
                           ymax = prop + 2*prop_se, fill = race, color = race,
                           group = race))

p + geom_ribbon(alpha = 0.3, aes(color = NULL)) +
  geom_line() +
  facet_wrap(~ degree, ncol = 1) +
  scale_y_continuous(labels = scales::percent) +
  scale_color_brewer(type = "qual", palette = "Dark2") +
  scale_fill_brewer(type = "qual", palette = "Dark2") +
  labs(title = "Educational Attainment\nby Race",
       subtitle = "GSS 1976–2016", fill = "Race",
       color = "Race", x = NULL, y = "Percent") +
  theme(legend.position = "top")
```



34.10 Where to go next

In general, when you estimate models and want to plot the results, the difficult step is not the plotting but rather calculating and extracting the right numbers. Generating predicted values and measures of confidence or uncertainty from models requires that you understand the model you are fitting, and the function you use to fit it, especially when it involves interactions, cross-level effects, or transformations of the predictor or response scales. The details can vary substantially from model type to model type, and also with the goals of any particular analysis. It is unwise to approach them mechanically. That said, several tools exist to help you work with model objects and produce a default set of plots from them.

34.10.1 Default plots for models

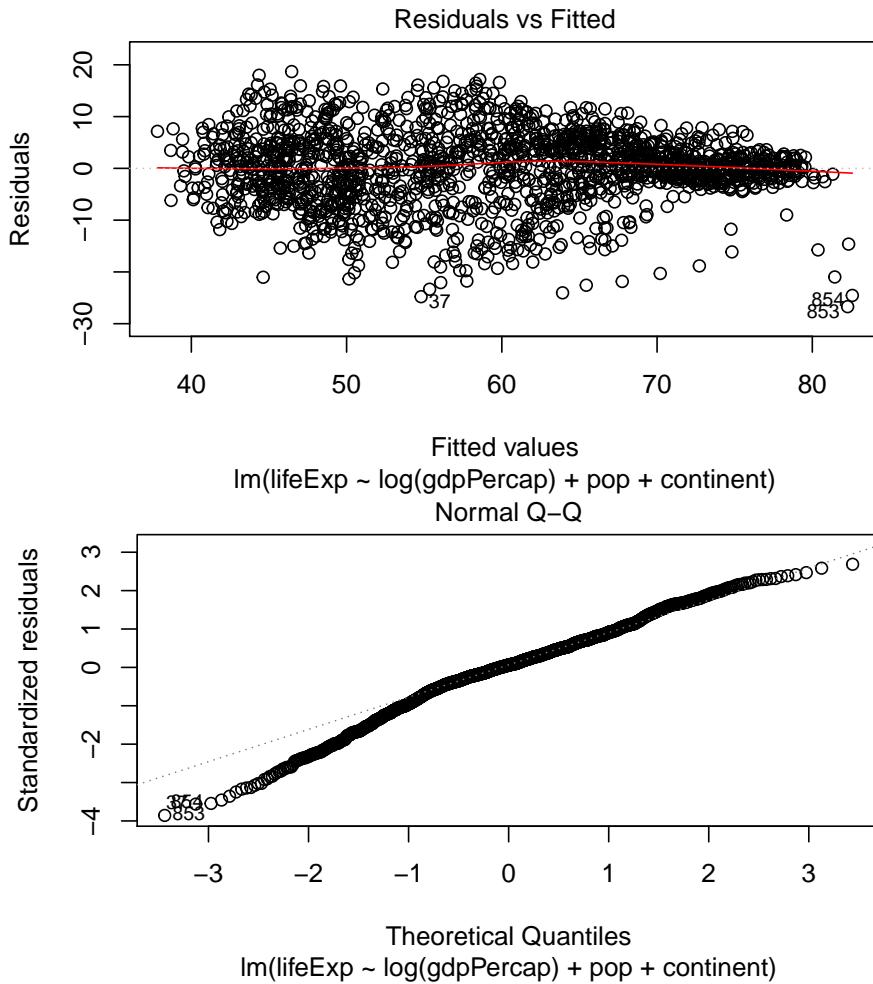
Just as model objects in R usually have a default `summary()` method, printing out an overview tailored to the type of model it is, they will usually have a default `plot()` method, too. Figures produced by `plot()` are typically not generated via `ggplot`, but it is usually worth exploring them. They typically make use of

either R's base graphics or the `lattice` library (Sarkar, 2008). These are two plotting systems that we do not cover in this book. Default plot methods are easy to examine. Let's take a look again at our simple OLS model.

```
out <- lm(formula = lifeExp ~ log(gdpPercap) + pop + continent, data = gapminder)
```

To look at some of R's default plots for this model, use the `plot()` function.

```
# Plot not shown
plot(out, which = c(1,2), ask=FALSE)
```

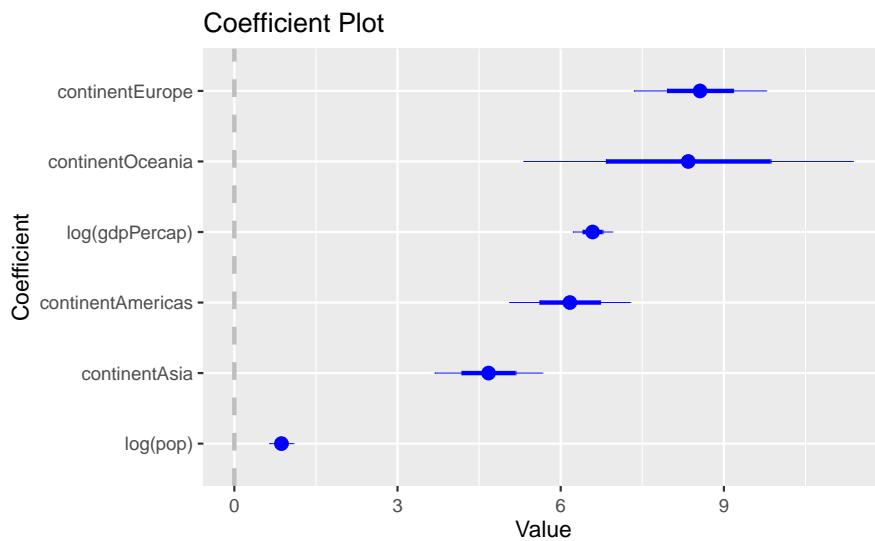


The `which()` statement here selects the first two of four default plots for this kind of model. If you want to easily reproduce base R's default model graphics using `ggplot`, the `ggfortify` library is worth examining. It is in some ways similar to `broom`, in that it tidies the output of model objects, but it focuses on producing a standard plot (or group of plots) for a wide variety of model types. It does this by defining a function called `autoplot()`. The idea is to be able to use `autoplot()` with the output of many different kinds of model.

A second option worth looking at is the `coefplot` library. It provides a quick way to produce good-quality plots of point estimates and confidence intervals. It has the advantage of managing the estimation of interaction effects and other occasionally tricky calculations.

```
library(coefplot)
out <- lm(formula = lifeExp ~ log(gdpPercap) + log(pop) + continent, data = gapminder)

coefplot(out, sort = "magnitude", intercept = FALSE)
```



34.10.2 Tools in development

Tidyverse tools for modeling and model exploration are being actively developed. The `broom` and `margins` libraries continue to get more and more useful. There are also other projects worth paying attention to. The `infer` package infer.netlify.com is in its early stages but can already do useful things in a pipeline-friendly way. You can install it from CRAN with `install.packages("infer")`.

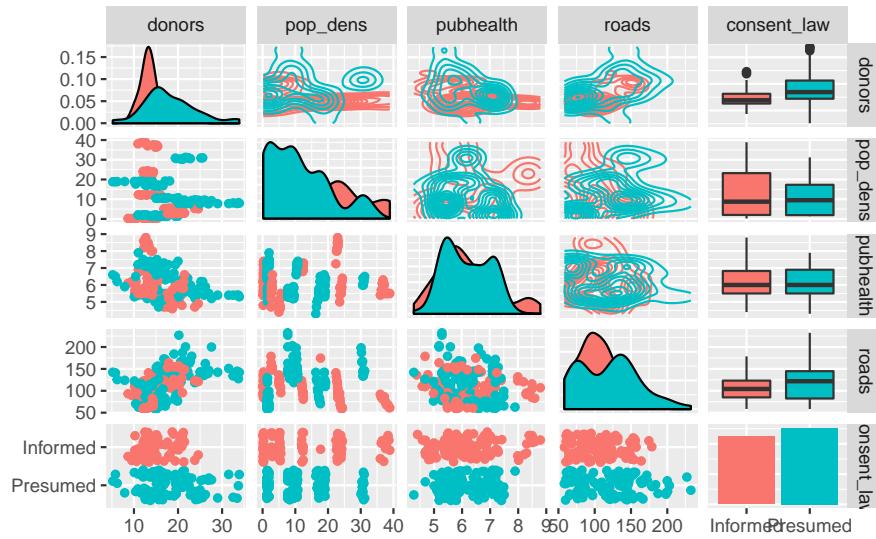
34.10.3 Extensions to ggplot

The `GGally` package provides a suite of functions designed to make producing standard but somewhat complex plots a little easier. For instance, it can produce generalized pairs plots, a useful way of quickly examining possible relationships between several different variables at once. This sort of plot is like the visual version of a correlation matrix. It shows a bivariate plot for all pairs of variables in the data. This is relatively straightforward when all the variables are continuous measures. Things get more complex when, as is often the case in the social sciences, some or all variables are categorical or otherwise limited in the range of values they can take. A generalized pairs plot can handle these cases. For example, Figure ?? shows a generalized pairs plot for five variables from the `organdata` dataset.

```
library(GGally)

organdata_sm <- organdata %>%
  select(donors, pop_dens, pubhealth,
         roads, consent_law)

ggpairs(data = organdata_sm,
        mapping = aes(color = consent_law),
        upper = list(continuous = wrap("density"), combo = "box_no_facet"),
        lower = list(continuous = wrap("points"), combo = wrap("dot_no_facet")))
```



Multi-panel plots like this are intrinsically very rich in information. When combined with several within-panel types of representation, or any more than a modest number of variables, they can become quite complex. They should be used less for the presentation of finished work, although it is possible. More often they are a useful tool for the working researcher to quickly investigate aspects of a dataset. The goal is not to pithily summarize a single point one already knows, but to open things up for further exploration.

Chapter 35

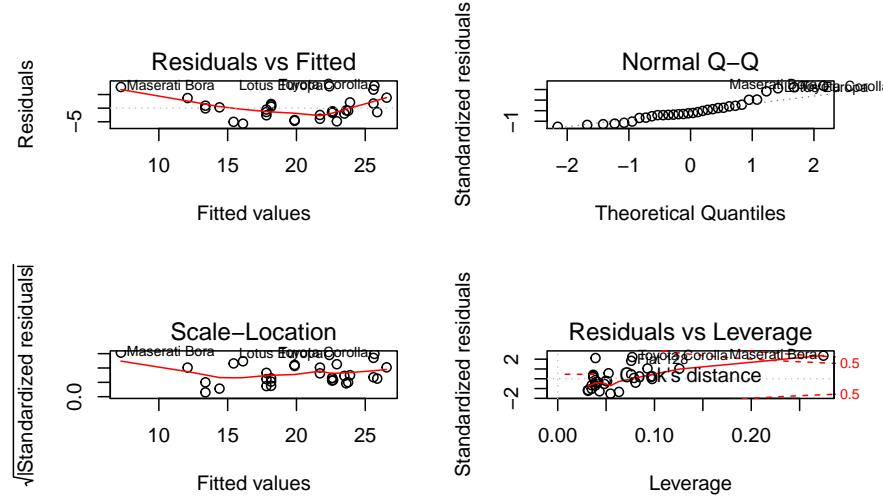
Visualizing residuals

Source: <https://www.r-bloggers.com/visualising-residuals/>

```
fit <- lm(mpg ~ hp, data = mtcars) # Fit the model
summary(fit) # Report the results
#>
#> Call:
#> lm(formula = mpg ~ hp, data = mtcars)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -5.712 -2.112 -0.885  1.582  8.236
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 30.0989   1.6339  18.42 < 2e-16 ***
#> hp          -0.0682   0.0101  -6.74 1.8e-07 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 3.86 on 30 degrees of freedom
#> Multiple R-squared:  0.602, Adjusted R-squared:  0.589
#> F-statistic: 45.5 on 1 and 30 DF,  p-value: 1.79e-07

par(mfrow = c(2, 2)) # Split the plotting panel into a 2 x 2 grid
plot(fit) # Plot the model information

par(mfrow = c(1, 1)) # Return plotting panel to 1 section
```



35.1 Simple Linear Regression

```
d <- mtcars
fit <- lm(mpg ~ hp, data = d)

d$predicted <- predict(fit) # Save the predicted values
d$residuals <- residuals(fit) # Save the residual values

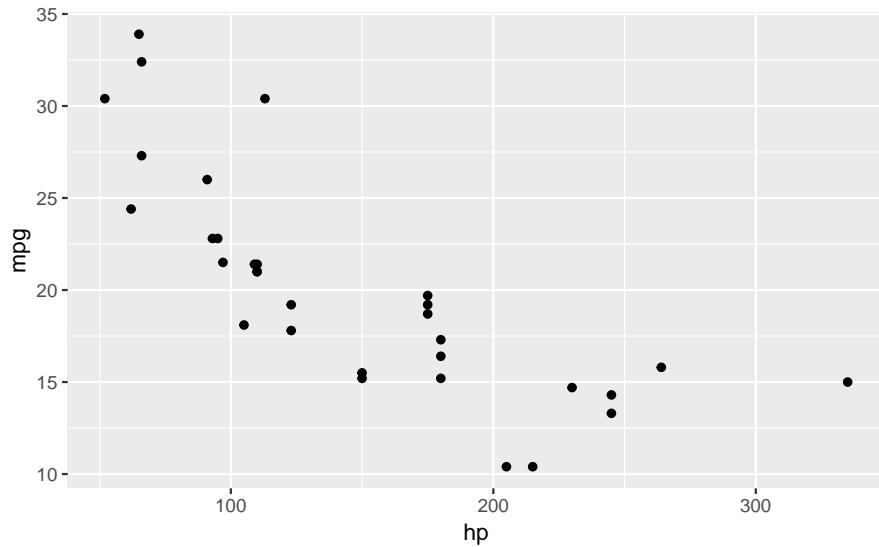
# Quick look at the actual, predicted, and residual values
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union
d %>% select(mpg, predicted, residuals) %>% head()
#>          mpg    predicted   residuals
#> Mazda RX4     21.0      22.6 -1.594
#> Mazda RX4 Wag 21.0      22.6 -1.594
#> Datsun 710    22.8      23.8 -0.954
#> Hornet 4 Drive 21.4      22.6 -1.194
#> Hornet Sportabout 18.7     18.2  0.541
#> Valiant       18.1      22.9 -4.835
```

35.1.1 Step 3: plot the actual and predicted values

plot first the actual data

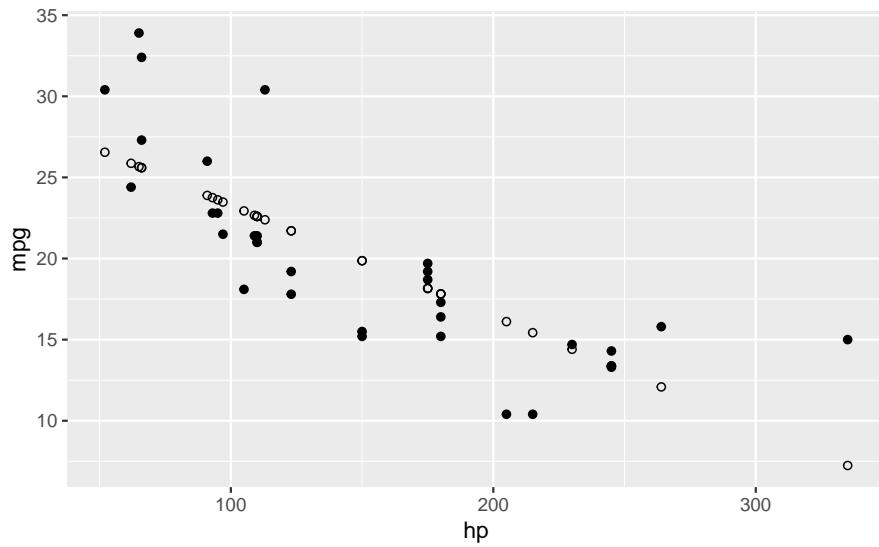
```
library(ggplot2)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method        from
#>   [.quosures     rlang
```

```
#>   c_quosures     rlang
#>   print_quosures rlang
ggplot(d, aes(x = hp, y = mpg)) + # Set up canvas with outcome variable on y-axis
  geom_point() # Plot the actual points
```



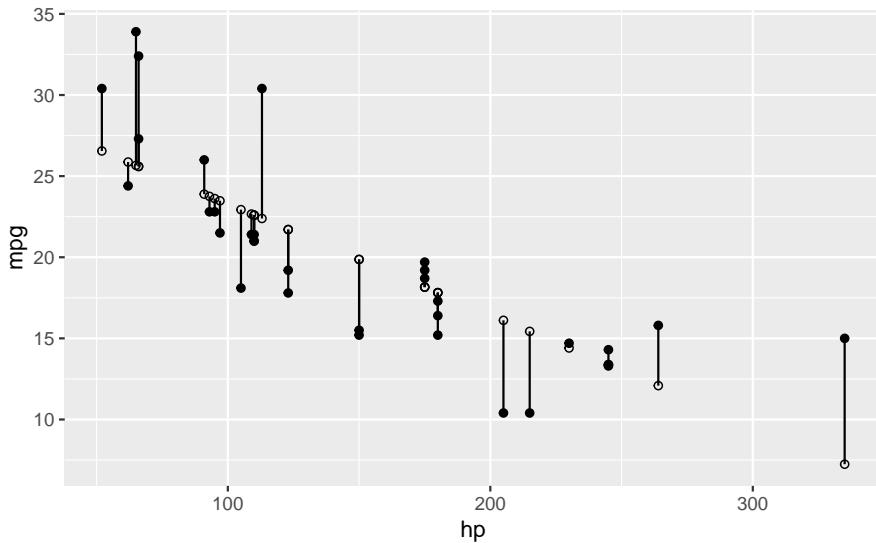
Next, we plot the predicted values in a way that they're distinguishable from the actual values. For example, let's change their shape:

```
ggplot(d, aes(x = hp, y = mpg)) +
  geom_point() +
  geom_point(aes(y = predicted), shape = 1) # Add the predicted values
```



This is on track, but it's difficult to see how our actual and predicted values are related. Let's connect the actual data points with their corresponding predicted value using `geom_segment()`:

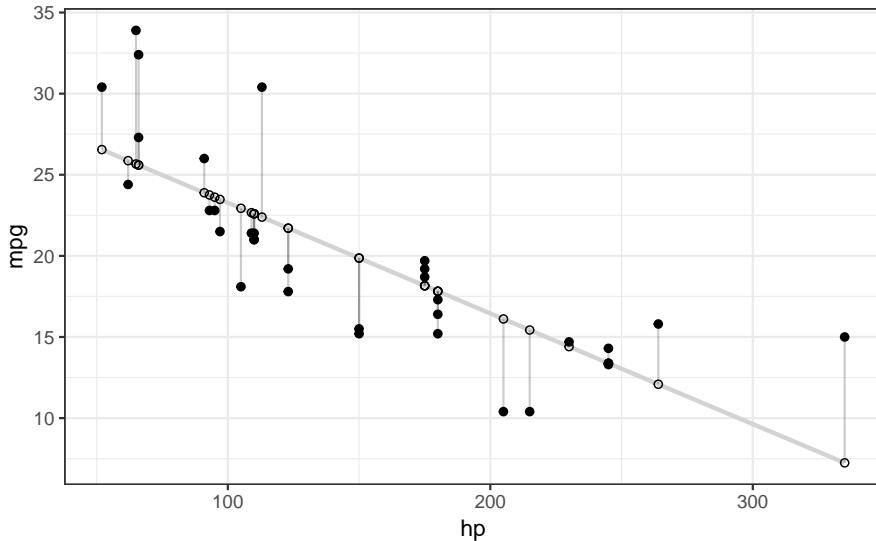
```
ggplot(d, aes(x = hp, y = mpg)) +
  geom_segment(aes(xend = hp, yend = predicted)) +
  geom_point() +
  geom_point(aes(y = predicted), shape = 1)
```



We'll make a few final adjustments:

- * Clean up the overall look with `theme_bw()`.
- * Fade out connection lines by adjusting their alpha.
- * Add the regression slope with `geom_smooth()`:

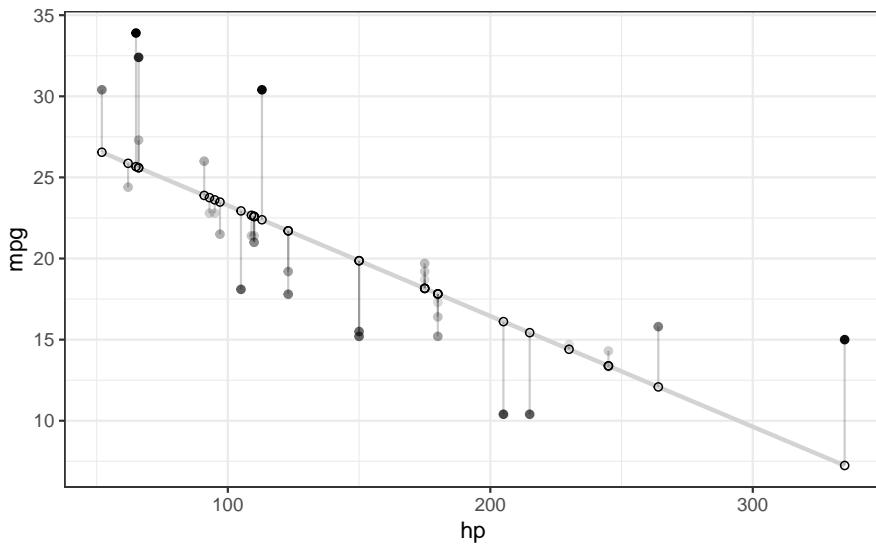
```
library(ggplot2)
ggplot(d, aes(x = hp, y = mpg)) +
  geom_smooth(method = "lm", se = FALSE, color = "lightgrey") + # Plot regression slope
  geom_segment(aes(xend = hp, yend = predicted), alpha = .2) + # alpha to fade lines
  geom_point() +
  geom_point(aes(y = predicted), shape = 1) +
  theme_bw() # Add theme for cleaner look
```



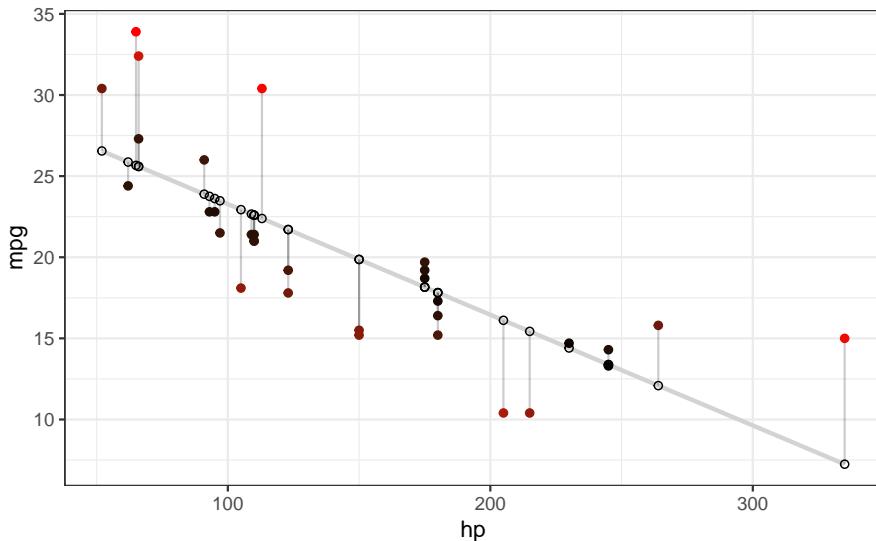
35.2 Step 4: use residuals to adjust

Finally, we want to make an adjustment to highlight the size of the residual. There are MANY options. To make comparisons easy, I'll make adjustments to the actual values, but you could just as easily apply these, or other changes, to the predicted values. Here are a few examples building on the previous plot:

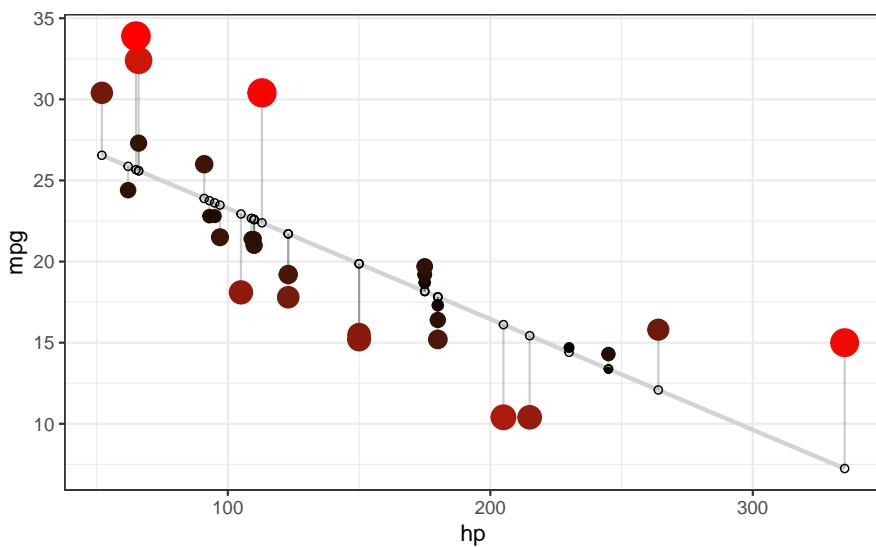
```
# ALPHA
# Changing alpha of actual values based on absolute value of residuals
ggplot(d, aes(x = hp, y = mpg)) +
  geom_smooth(method = "lm", se = FALSE, color = "lightgrey") +
  geom_segment(aes(xend = hp, yend = predicted), alpha = .2) +
  # > Alpha adjustments made here...
  geom_point(aes(alpha = abs(residuals))) + # Alpha mapped to abs(residuals)
  guides(alpha = FALSE) + # Alpha legend removed
  # <
  geom_point(aes(y = predicted), shape = 1) +
  theme_bw()
```



```
# COLOR
# High residuals (in absolute terms) made more red on actual values.
ggplot(d, aes(x = hp, y = mpg)) +
  geom_smooth(method = "lm", se = FALSE, color = "lightgrey") +
  geom_segment(aes(xend = hp, yend = predicted), alpha = .2) +
  # > Color adjustments made here...
  geom_point(aes(color = abs(residuals))) + # Color mapped to abs(residuals)
  scale_color_continuous(low = "black", high = "red") + # Colors to use here
  guides(color = FALSE) + # Color legend removed
  # <
  geom_point(aes(y = predicted), shape = 1) +
  theme_bw()
```



```
# SIZE AND COLOR
# Same coloring as above, size corresponding as well
ggplot(d, aes(x = hp, y = mpg)) +
  geom_smooth(method = "lm", se = FALSE, color = "lightgrey") +
  geom_segment(aes(xend = hp, yend = predicted), alpha = .2) +
  # > Color AND size adjustments made here...
  geom_point(aes(color = abs(residuals), size = abs(residuals))) + # size also mapped
  scale_color_continuous(low = "black", high = "red") +
  guides(color = FALSE, size = FALSE) + # Size legend also removed
  # <
  geom_point(aes(y = predicted), shape = 1) +
  theme_bw()
```



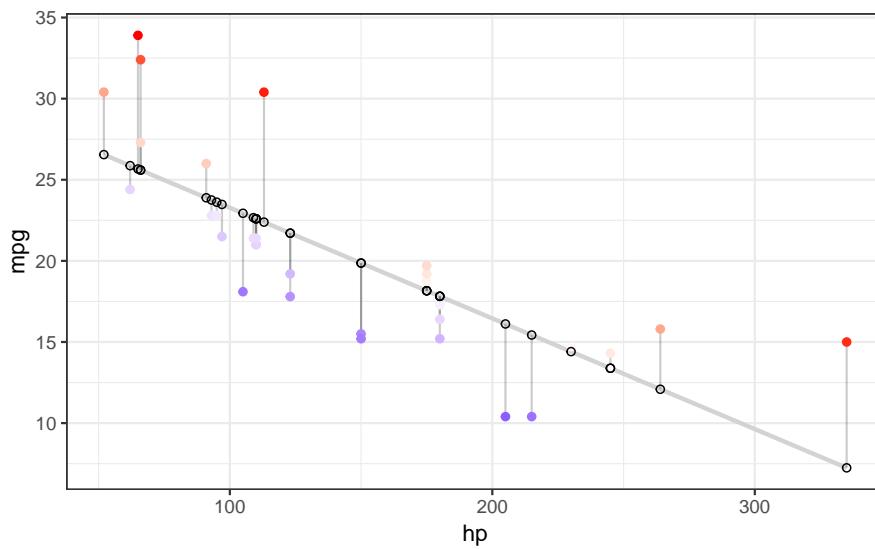
```
# COLOR UNDER/OVER
# Color mapped to residual with sign taken into account.
# i.e., whether actual value is greater or less than predicted
ggplot(d, aes(x = hp, y = mpg)) +
```

```

geom_smooth(method = "lm", se = FALSE, color = "lightgrey") +
  geom_segment(aes(xend = hp, yend = predicted), alpha = .2) +
  # > Color adjustments made here...
  geom_point(aes(color = residuals)) + # Color mapped here
  scale_color_gradient2(low = "blue", mid = "white", high = "red") + # Colors to use here
  guides(color = FALSE) +
  # <

  geom_point(aes(y = predicted), shape = 1) +
  theme_bw()

```



I particularly like this last example, because the colours nicely help to identify non-linearity in the data. For example, we can see that there is more red for extreme values of hp where the actual values are greater than what is being predicted. There is more blue in the centre, however, indicating that the actual values are less than what is being predicted. Together, this suggests that the relationship between the variables is non-linear, and might be better modelled by including a quadratic term in the regression equation.

Part V

Neural Networks

Chapter 36

Building deep neural nets with h2o that predict arrhythmia of the heart

36.1 Introduction

27 February 2017

This week, I am showing how to build feed-forward deep neural networks or multilayer perceptrons. The models in this example are built to classify ECG data into being either from healthy hearts or from someone suffering from arrhythmia. I will show how to prepare a dataset for modeling, setting weights and other modeling parameters, and finally, how to evaluate model performance with the **h2o** package.

36.1.1 Deep learning with neural networks

Deep learning with neural networks is arguably one of the most rapidly growing applications of machine learning and AI today. They allow building complex models that consist of multiple hidden layers within artificial networks and are able to find non-linear patterns in unstructured data. Deep neural networks are usually feed-forward, which means that each layer feeds its output to subsequent layers, but recurrent or feed-back neural networks can also be built. Feed-forward neural networks are also called multilayer perceptrons (MLPs).

36.1.2 H2O

The R package **h2o** provides a convenient interface to **H2O**, which is an open-source machine learning and deep learning platform. H2O distributes a wide range of common machine learning algorithms for classification, regression and deep learning.

36.1.3 Preparing the R session

First, we need to load the packages.

```
library(dplyr)
library(h2o)
library(ggplot2)
library(ggrepel)
library(h2o)
```

```

h2o.init()
#>
#> H2O is not running yet, starting it now...
#>
#> Note: In case of errors look at the following log files:
#>       /tmp/RtmpQvWp1Q/h2o_datascience_started_from_r.out
#>       /tmp/RtmpQvWp1Q/h2o_datascience_started_from_r.err
#>
#>
#> Starting H2O JVM and connecting: . Connection successful!
#>
#> R is connected to the H2O cluster:
#>   H2O cluster uptime:      1 seconds 256 milliseconds
#>   H2O cluster timezone:    America/Chicago
#>   H2O data parsing timezone: UTC
#>   H2O cluster version:     3.22.1.1
#>   H2O cluster version age: 8 months and 23 days !!!
#>   H2O cluster name:        H2O_started_from_R_datascience_mwl453
#>   H2O cluster total nodes: 1
#>   H2O cluster total memory: 6.96 GB
#>   H2O cluster total cores:  8
#>   H2O cluster allowed cores: 8
#>   H2O cluster healthy:     TRUE
#>   H2O Connection ip:       localhost
#>   H2O Connection port:     54321
#>   H2O Connection proxy:    NA
#>   H2O Internal Security:   FALSE
#>   H2O API Extensions:     XGBoost, Algos, AutoML, Core V3, Core V4
#>   R Version:               R version 3.6.0 (2019-04-26)
#> Warning in h2o.clusterInfo():
#> Your H2O cluster version is too old (8 months and 23 days) !
#> Please download and install the latest version from http://h2o.ai/download/

my_theme <- function(base_size = 12, base_family = "sans"){
  theme_minimal(base_size = base_size, base_family = base_family) +
  theme(
    axis.text = element_text(size = 12),
    axis.title = element_text(size = 14),
    panel.grid.major = element_line(color = "grey"),
    panel.grid.minor = element_blank(),
    panel.background = element_rect(fill = "aliceblue"),
    strip.background = element_rect(fill = "darkgrey", color = "grey", size = 1),
    strip.text = element_text(face = "bold", size = 12, color = "white"),
    legend.position = "right",
    legend.justification = "top",
    panel.border = element_rect(color = "grey", fill = NA, size = 0.5)
  )
}

```

36.2 Arrhythmia data

The data I am using to demonstrate the building of neural nets is the arrhythmia dataset from UC Irvine's machine learning database. It contains 279 features from ECG heart rhythm diagnostics and one output column. I am not going to rename the feature columns because they are too many and the descriptions are too complex. Also, we don't need to know specifically which features we are looking at for building the models.

For a description of each feature, see <https://archive.ics.uci.edu/ml/machine-learning-databases/arrhythmia/arrhythmia.names>.

The output column defines 16 classes: *class 1* samples are from healthy ECGs, the remaining classes belong to different types of arrhythmia, with *class 16* being all remaining arrhythmia cases that didn't fit into distinct classes.

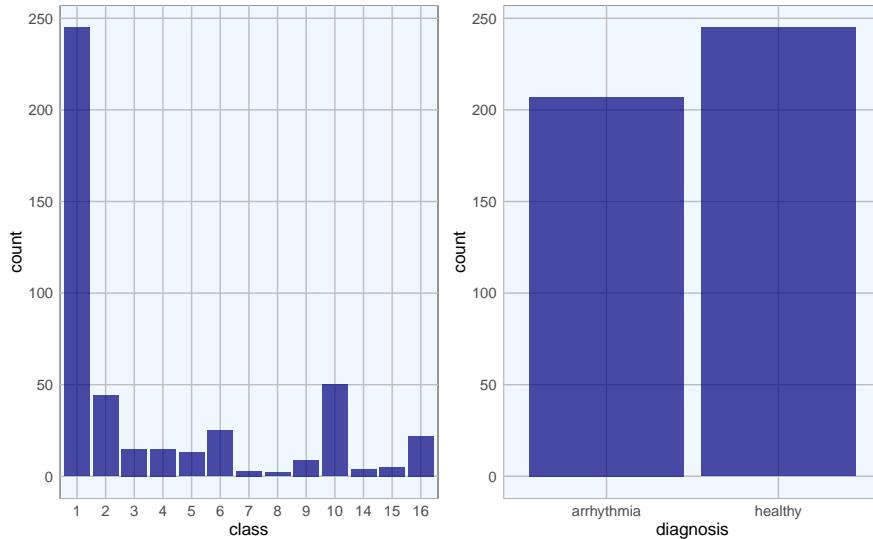
```
arrhythmia <- read.table(file.path(data_raw_dir, "arrhythmia.data.txt"), sep = ",")  
arrhythmia[arrhythmia == "?"] <- NA  
  
# making sure, that all feature columns are numeric  
arrhythmia[-280] <- lapply(arrhythmia[-280], as.character)  
arrhythmia[-280] <- lapply(arrhythmia[-280], as.numeric)  
  
# renaming output column and converting to factor  
colnames(arrhythmia)[280] <- "class"  
arrhythmia$class <- as.factor(arrhythmia$class)
```

As usual, I want to get acquainted with the data and explore its properties before I am building any model. So, I am first going to look at the distribution of classes and of healthy and arrhythmia samples.

```
p1 <- ggplot(arrhythmia, aes(x = class)) +  
  geom_bar(fill = "navy", alpha = 0.7) +  
  my_theme()
```

Because I am interested in distinguishing healthy from arrhythmia ECGs, I am converting the output to binary format by combining all arrhythmia cases into one class.

```
# all arrhythmia cases into one class  
arrhythmia$diagnosis <- ifelse(arrhythmia$class == 1, "healthy", "arrhythmia")  
arrhythmia$diagnosis <- as.factor(arrhythmia$diagnosis)  
  
p2 <- ggplot(arrhythmia, aes(x = diagnosis)) +  
  geom_bar(fill = "navy", alpha = 0.7) +  
  my_theme()  
  
library(gridExtra)  
#>  
## Attaching package: 'gridExtra'  
## The following object is masked from 'package:dplyr':  
##  
##     combine  
library(grid)  
  
grid.arrange(p1, p2, ncol = 2)
```



With binary classification, we have almost the same numbers of healthy and arrhythmia cases in our dataset.

I am also interested in how much the normal and arrhythmia cases cluster in a Principal Component Analysis (PCA). I am first preparing the PCA plotting function and then run it on the feature data.

```
library(pcaGoPromoter)
#> Warning: replacing previous import 'BiocGenerics::boxplot' by
#> 'graphics::boxplot' when loading 'Biostrings'
#> Warning: replacing previous import 'BiocGenerics::image' by
#> 'graphics::image' when loading 'Biostrings'
#> Warning: replacing previous import 'S4Vectors::na.exclude' by
#> 'stats::na.exclude' when loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::smoothEnds' by
#> 'stats::smoothEnds' when loading 'Biostrings'
#> Warning: replacing previous import 'BiocGenerics::density' by
#> 'stats::density' when loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::mad' by 'stats::mad' when
#> loading 'Biostrings'
#> Warning: replacing previous import 'S4Vectors::na.omit' by 'stats::na.omit'
#> when loading 'Biostrings'
#> Warning: replacing previous import 'S4Vectors::complete.cases' by
#> 'stats::complete.cases' when loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::runmed' by 'stats::runmed'
#> when loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::start' by 'stats::start' when
#> loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::window<-' by 'stats::window<-' 
#> when loading 'Biostrings'
#> Warning: replacing previous import 'S4Vectors::window' by 'stats::window'
#> when loading 'Biostrings'
#> Warning: replacing previous import 'S4Vectors::aggregate' by
#> 'stats::aggregate' when loading 'Biostrings'
#> Warning: replacing previous import 'BiocGenerics::weights' by
#> 'stats::weights' when loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::cor' by 'stats::cor' when
#> loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::cov' by 'stats::cov' when
```

```

#> loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::quantile' by 'stats::quantile'
#> when loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::end' by 'stats::end' when
#> loading 'Biostrings'
#> Warning: replacing previous import 'BiocGenerics::residuals' by
#> 'stats::residuals' when loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::median' by 'stats::median'
#> when loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::sd' by 'stats::sd' when
#> loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::var' by 'stats::var' when
#> loading 'Biostrings'
#> Warning: replacing previous import 'S4Vectors::xtabs' by 'stats::xtabs'
#> when loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::IQR' by 'stats::IQR' when
#> loading 'Biostrings'
#> Warning: replacing previous import 'S4Vectors::tail' by 'utils::tail' when
#> loading 'Biostrings'
#> Warning: replacing previous import 'IRanges::stack' by 'utils::stack' when
#> loading 'Biostrings'
#> Warning: replacing previous import 'XVector::relist' by 'utils::relist'
#> when loading 'Biostrings'
#> Warning: replacing previous import 'S4Vectors::head' by 'utils::head' when
#> loading 'Biostrings'

pca_func <- function(pcaOutput2, group_name){
  centroids <- aggregate(cbind(PC1, PC2) ~ groups, pcaOutput2, mean)
  conf.rgn <- do.call(rbind, lapply(unique(pcaOutput2$groups), function(t)
    data.frame(groups = as.character(t),
               ellipse(cov(pcaOutput2[pcaOutput2$groups == t, 1:2]),
                        centre = as.matrix(centroids[centroids$groups == t, 2:3]),
                        level = 0.95),
               stringsAsFactors = FALSE)))
  plot <- ggplot(data = pcaOutput2, aes(x = PC1, y = PC2, group = groups,
                                         color = groups)) +
    geom_polygon(data = conf.rgn, aes(fill = groups), alpha = 0.2) +
    geom_point(size = 2, alpha = 0.5) +
    labs(color = paste(group_name),
         fill = paste(group_name),
         x = paste0("PC1: ", round(pcaOutput$pov[1], digits = 2) * 100, "% variance"),
         y = paste0("PC2: ", round(pcaOutput$pov[2], digits = 2) * 100, "% variance")) +
    my_theme()

  return(plot)
}

# Find what columns have NAs and the quantity
for (col in names(arrhythmia)) {
  n_nas <- length(which(is.na(arrhythmia[, col])))
  if (n_nas > 0) cat(col, n_nas, "\n")
}
#> V11 8

```

```
#> V12 22
#> V13 1
#> V14 376
#> V15 1

# Replace NAs with zeros
arrhythmia[is.na(arrhythmia)] <- 0
```

Find and plot the PCAs.

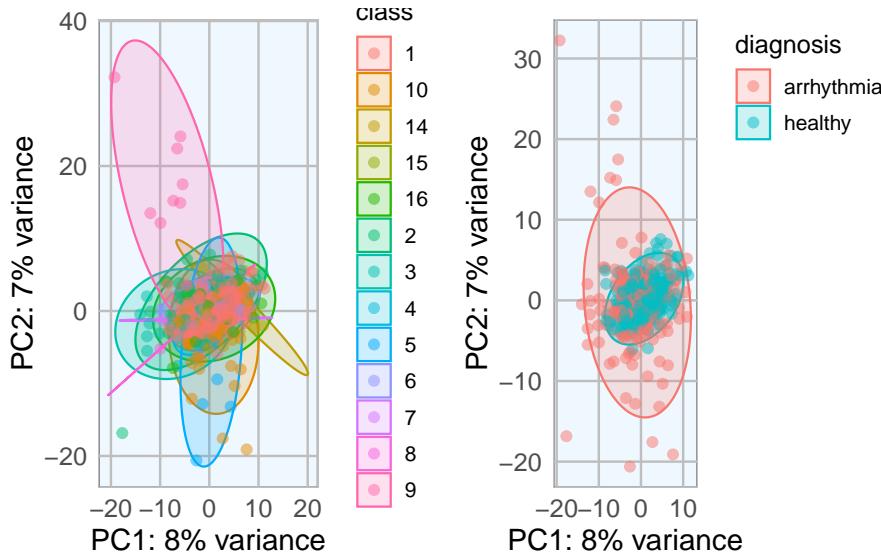
```
pcaOutput <- pca(t(arrhythmia[-c(280, 281)]), printDropped=FALSE,
                     scale=TRUE,
                     center = TRUE)

pcaOutput2 <- as.data.frame(pcaOutput$scores)

pcaOutput2$groups <- arrhythmia$class
p1 <- pca_func(pcaOutput2, group_name = "class")

pcaOutput2$groups <- arrhythmia$diagnosis
p2 <- pca_func(pcaOutput2, group_name = "diagnosis")

grid.arrange(p1, p2, ncol = 2)
```



The PCA shows that there is a big overlap between healthy and arrhythmia samples, i.e. there does not seem to be major global differences in all features. The class that is most distinct from all others seems to be class 9.

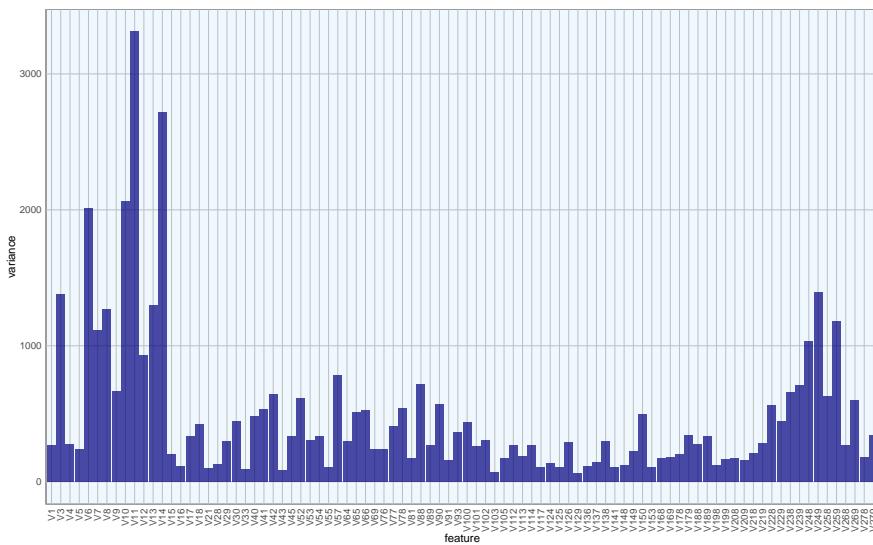
I want to give the arrhythmia cases that are very different from the rest a stronger weight in the neural network, so I define a **weight column** where every sample outside the central PCA cluster will get a “2”, they will in effect be used twice in the model.

```
weights <- ifelse(pcaOutput2$PC1 < -5 & abs(pcaOutput2$PC2) > 10, 2, 1)
```

I also want to know what the variance is within features.

```
library(matrixStats)
#>
#> Attaching package: 'matrixStats'
```

```
#> The following object is masked from 'package:dplyr':  
#>  
#>     count  
  
colvars <- data.frame(feature = colnames(arrhythmia[-c(280, 281)]),  
                      variance = colVars(as.matrix(arrhythmia[-c(280, 281)])))  
  
subset(colvars, variance > 50) %>%  
  mutate(feature = factor(feature, levels = colnames(arrhythmia[-c(280, 281)]))) %>%  
  ggplot(aes(x = feature, y = variance)) +  
    geom_bar(stat = "identity", fill = "navy", alpha = 0.7) +  
    my_theme() +  
    theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust = 1))
```



Features with low variance are less likely to strongly contribute to a differentiation between healthy and arrhythmia cases, so I am going to remove them. I am also concatenating the weights column:

```
arrhythmia_subset <- cbind(weights,
                           arrhythmia[, c(281, 280, which(colvars$variance > 50))])
```

36.3 Converting the dataframe to a h2o object

Now that I have my final data frame for modeling, for working with h2o functions, the data needs to be converted from a **DataFrame** to an **H2O Frame**. This is done with the `as_h2o_frame()` function.

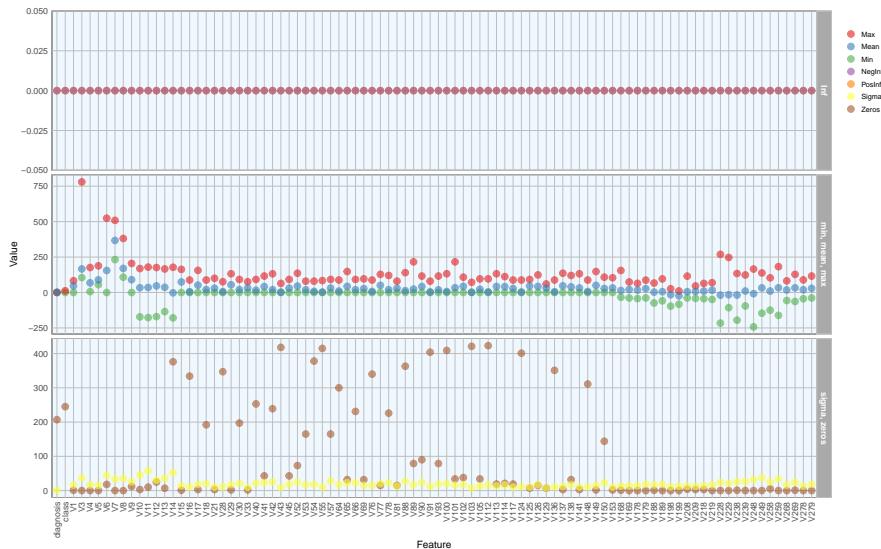
```
#as_h2o_frame(arrhythmia_subset)
arrhythmia_hf <- as.h2o(arrhythmia_subset, key="arrhythmia.hex")
#>
| |
| |
| |
|=====| 0%
|=====| 100%
```

We can now access all functions from the `h2o` package that are built to work on `h2o` Frames. A useful such function is `h2o.describe()`. It is similar to base R's `summary()` function but outputs many more descriptive measures for our data. To get a good overview about these measures, I am going to plot them.

```

library(tidyr) # for gathering
#>
#> Attaching package: 'tidyr'
#> The following object is masked from 'package:S4Vectors':
#>
#>     expand
h2o.describe(arrhythmia_hf[, -1]) %>% # excluding the weights column
  gather(x, y, Zeros:Sigma) %>%
  mutate(group = ifelse(
    x %in% c("Min", "Max", "Mean"), "min, mean, max",
    ifelse(x %in% c("NegInf", "PosInf"), "Inf", "sigma, zeros"))) %>%
  # separating them into facets makes them easier to see
  mutate(Label = factor(Label, levels = colnames(arrhythmia_hf[, -1]))) %>%
  ggplot(aes(x = Label, y = as.numeric(y), color = x)) +
  geom_point(size = 4, alpha = 0.6) +
  scale_color_brewer(palette = "Set1") +
  my_theme() +
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust = 1)) +
  facet_grid(group ~ ., scales = "free") +
  labs(x = "Feature",
       y = "Value",
       color = "")
#> Warning: Removed 2 rows containing missing values (geom_point).

```



I am also interested in the correlation between features and the output. We can use the `h2o.cor()` function to calculate the correlation matrix. It is again much easier to understand the data when we visualize it, so I am going to create another plot.

```

library(reshape2) # for melting
#>
#> Attaching package: 'reshape2'
#> The following object is masked from 'package:tidyr':
#>
#>     smiths

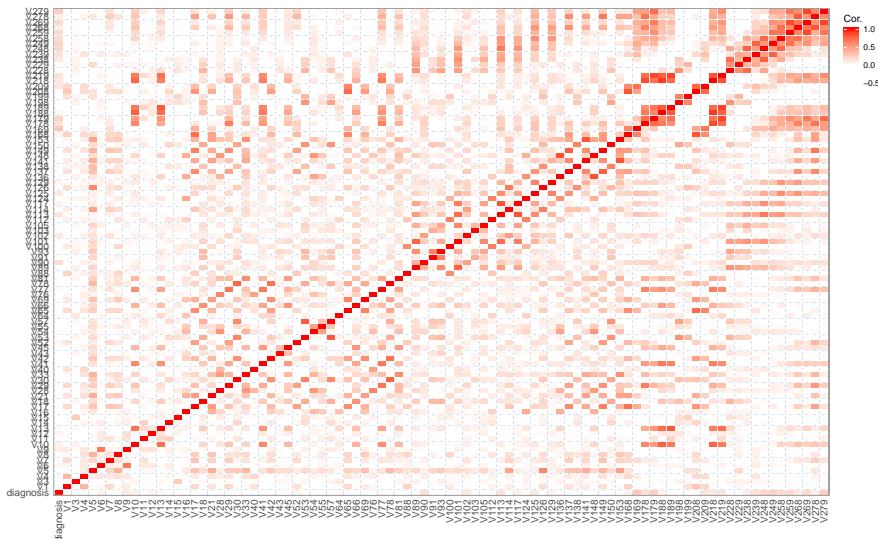
```

```
# diagnosis is now a character column and we need to convert it again
arrhythmia_hf[, 2] <- h2o.asfactor(arrhythmia_hf[, 2])
arrhythmia_hf[, 3] <- h2o.asfactor(arrhythmia_hf[, 3]) # same for class

cor <- h2o.cor(arrhythmia_hf[, -c(1, 3)])
rownames(cor) <- colnames(cor)

melt(cor) %>%
  mutate(Var2 = rep(rownames(cor), nrow(cor))) %>%
  mutate(Var2 = factor(Var2, levels = colnames(cor))) %>%
  mutate(variable = factor(variable, levels = colnames(cor))) %>%
  ggplot(aes(x = variable, y = Var2, fill = value)) +
  geom_tile(width = 0.9, height = 0.9) +
  scale_fill_gradient2(low = "white", high = "red", name = "Cor.") +
  my_theme() +
  theme(axis.text.x = element_text(angle = 90, vjust = 0.5, hjust = 1)) +
  labs(x = "",
       y = "")

##> No id variables; using all as measure variables
```



36.4 Training, test and validation data

Now we can use the `h2o.splitFrame()` function to split the data into training, validation and test data.

Here, I am using 70% for training and 15% each for validation and testing. We could also just split the data into two sections, a training and test set but when we have sufficient samples, it is a good idea to evaluate model performance on an independent test set on top of training with a validation set. Because we can easily overfit a model, we want to get an idea about how generalizable it is - this we can only assess by looking at how well it works on previously unknown data.

I am also defining `response`, `features` and `weights` column names now.

```
splits <- h2o.splitFrame(arrhythmia_hf,
                           ratios = c(0.7, 0.15),
                           seed = 1)
```

```

train <- splits[[1]]
valid <- splits[[2]]
test <- splits[[3]]

response <- "diagnosis"
weights <- "weights"
features <- setdiff(colnames(train), c(response, weights, "class"))

summary(train$diagnosis, exact_quantiles = TRUE)
#> diagnosis
#> healthy    :163
#> arrhythmia:155

summary(valid$diagnosis, exact_quantiles = TRUE)
#> diagnosis
#> healthy    :43
#> arrhythmia:25

summary(test$diagnosis, exact_quantiles = TRUE)
#> diagnosis
#> healthy    :39
#> arrhythmia:27

```

If we had more categorical features, we could use the `h2o.interaction()` function to define interaction terms, but since we only have numeric features here, we don't need this.

We can also run a PCA on the training data, using the `h2o.prcomp()` function to calculate the singular value decomposition of the Gram matrix with the power method.

```

pca <- h2o.prcomp(training_frame = train,
                    x = features,
                    validation_frame = valid,
                    transform = "NORMALIZE",
                    k = 3,
                    seed = 42)

#>
#> |                                         0%
#> |-----| 20%
#> |=====| 100%
#> # Warning in doTryCatch(return(expr), name, parentenv, handler): _train:
#> # Dataset used may contain fewer number of rows due to removal of rows with
#> # NA/missing values. If this is not desirable, set impute_missing argument in
#> # pca call to TRUE/True/true/... depending on the client language.

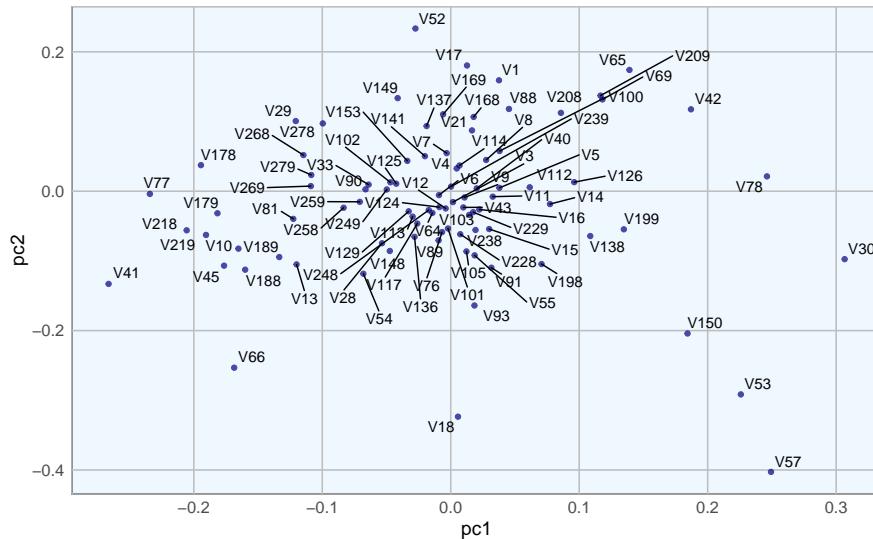
pca
#> Model Details:
#> =====
#>
#> H2ODimReductionModel: pca
#> Model ID: PCA_model_R_1569010079835_1
#> Importance of components:
#>          pc1      pc2      pc3

```

```
#> Standard deviation      0.582620 0.507796 0.421869
#> Proportion of Variance 0.164697 0.125110 0.086351
#> Cumulative Proportion   0.164697 0.289808 0.376159
#>
#>
#> H2ODimReductionMetrics: pca
#>
#> No model metrics available for PCA
#> H2ODimReductionMetrics: pca
#>
#> No model metrics available for PCA
```

```
eigenvec <- as.data.frame(pca@model$eigenvectors)
eigenvec$label <- features

ggplot(eigenvec, aes(x = pc1, y = pc2, label = label)) +
  geom_point(color = "navy", alpha = 0.7) +
  geom_text_repel() +
  my_theme()
```



36.5 Modeling

Now, we can build a deep neural network model. We can specify quite a few parameters, like

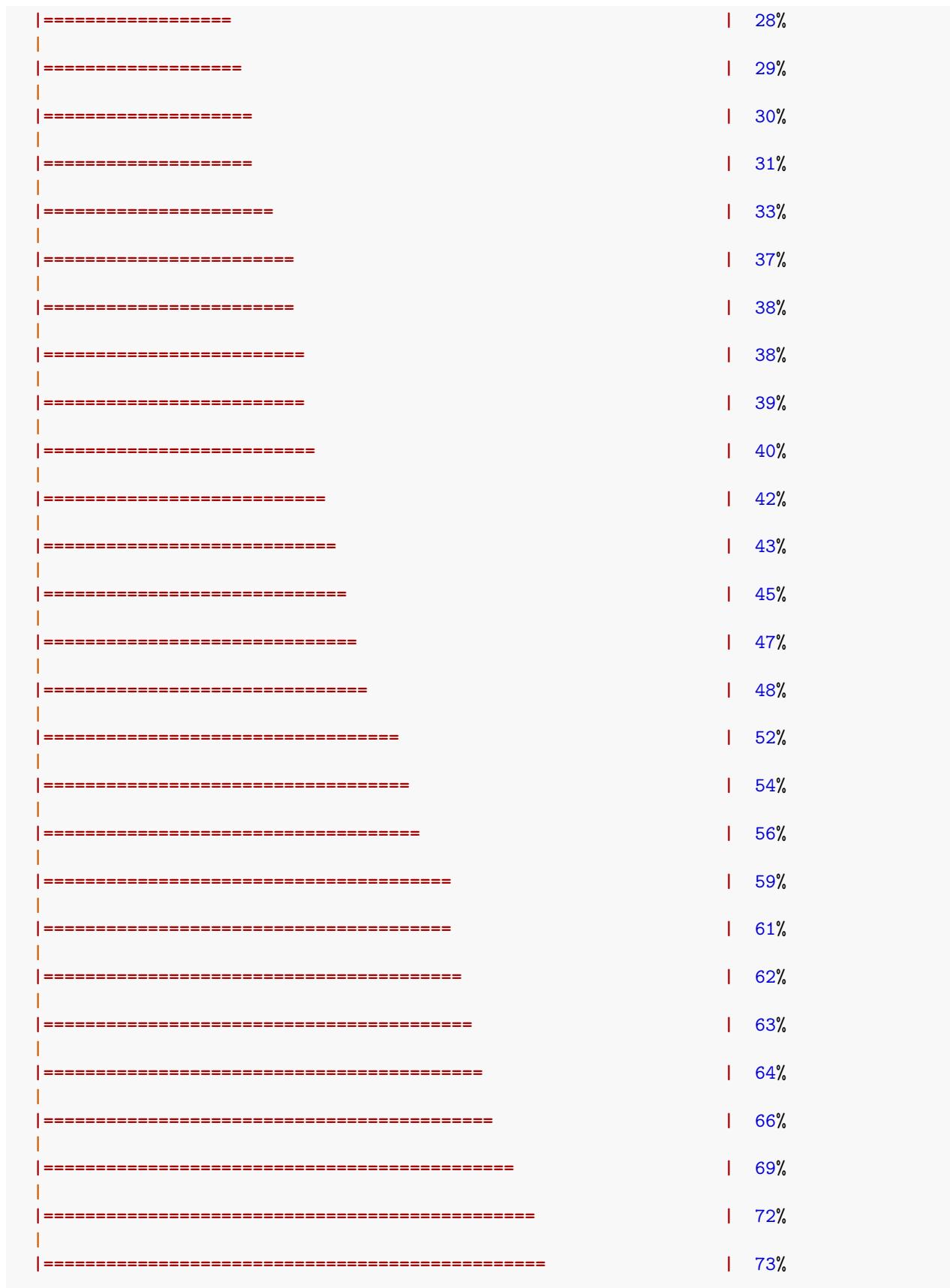
- **Cross-validation:** Cross validation can tell us the training and validation errors for each model. The final model will be overwritten with the best model, if we don't specify otherwise.
- **Adaptive learning rate:** For deep learning with h2o, we by default use stochastic gradient descent optimization with an adaptive learning rate. The two corresponding parameters rho and epsilon help us find global (or near enough) optima.
- **Activation function:** The activation function defines the node output relative to a given set of inputs. We want our activation function to be non-linear and continuously differentiable.
- **Hidden nodes:** Defines the number of hidden layers and the number of nodes per layer.

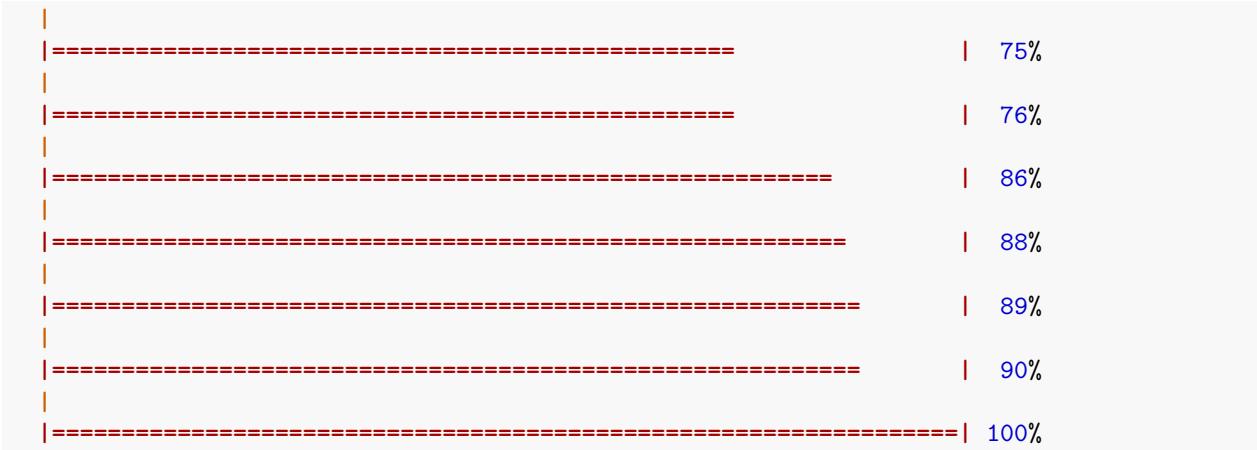
- **Epochs:** Increasing the number of epochs (one full training cycle on all training samples) can increase model performance, but we also run the risk of overfitting. To determine the optimal number of epochs, we need to use early stopping.
- **Early stopping:** By default, early stopping is enabled. This means that training will be stopped when we reach a certain validation error to prevent overfitting.

Of course, you need quite a bit of experience and intuition to hit on a good combination of parameters. That's why it usually makes sense to do a grid search for hyper-parameter tuning. Here, I want to focus on building and evaluating deep learning models, though. I will cover grid search in next week's post.

```
# this will take some time and all CPUs
dl_model <- h2o.deeplearning(x = features,
                             y = response,
                             weights_column = weights,
                             model_id = "dl_model",
                             training_frame = train,
                             validation_frame = valid,
                             nfolds = 15,                                # 10x cross validation
                             keep_cross_validation_fold_assignment = TRUE,
                             fold_assignment = "Stratified",
                             activation = "RectifierWithDropout",
                             score_each_iteration = TRUE,
                             hidden = c(200, 200, 200, 200, 200),          # 5 hidden layers, each of 200 neurons
                             epochs = 100,
                             variable_importances = TRUE,
                             export_weights_and_biases = TRUE,
                             seed = 42)

#>
|=====
|=                                         0%
|-----
|-----                                     2%
|-----
|-----                                     9%
|-----
|-----                                    10%
|-----
|-----                                    11%
|-----
|-----                                    16%
|-----
|-----                                    17%
|-----
|-----                                    19%
|-----
|-----                                    20%
|-----
|-----                                    21%
|-----
|-----                                    22%
|-----
|-----                                    26%
```





Because training can take a while, depending on how many samples, features, nodes and hidden layers you are training on, it is a good idea to save your model.

```
# if file exists, overwrite it
h2o.saveModel(dl_model, path = file.path(data_out_dir, "dl_model"), force = TRUE)
#> [1] "/home/datascience/repos/machine-learning-rsuite/export/dl_model/dl_model"
```

We can then re-load the model again any time to check the model quality and make predictions on new data.

```
dl_model <- h2o.loadModel(file.path(data_out_dir, "dl_model/dl_model"))
```

36.6 Model performance

We now want to know how our model performed on the validation data. The summary() function will give us a detailed overview of our model. I am not showing the output here, because it is quite extensive.

```
sum_model <- summary(dl_model)
#> Model Details:
#> =====
#>
#> H2OBinomialModel: deeplearning
#> Model Key: dl_model
#> Status of Neuron Layers: predicting diagnosis, 2-class classification, bernoulli distribution, Cross-
#> layer units          type dropout      l1      l2 mean_rate
#> 1     1    90      Input  0.00 %     NA     NA     NA
#> 2     2   200 RectifierDropout 50.00 % 0.000000 0.000000 0.003946
#> 3     3   200 RectifierDropout 50.00 % 0.000000 0.000000 0.006433
#> 4     4   200 RectifierDropout 50.00 % 0.000000 0.000000 0.009130
#> 5     5   200 RectifierDropout 50.00 % 0.000000 0.000000 0.007841
#> 6     6   200 RectifierDropout 50.00 % 0.000000 0.000000 0.023682
#> 7     7     2      Softmax     NA 0.000000 0.000000 0.002243
#> rate_rms momentum mean_weight weight_rms mean_bias bias_rms
#> 1     NA     NA      NA      NA     NA     NA
#> 2 0.003634 0.000000 0.002825 0.096114 0.429671 0.054778
#> 3 0.003863 0.000000 -0.008409 0.074646 0.949195 0.053149
#> 4 0.004991 0.000000 -0.007090 0.072220 0.966390 0.029603
#> 5 0.003807 0.000000 -0.006317 0.071143 0.973594 0.036020
#> 6 0.056406 0.000000 -0.009330 0.070480 0.952516 0.032818
#> 7 0.001142 0.000000 -0.041027 0.377082 0.000224 0.053261
```

```

#>
#> H2OBinomialMetrics: deeplearning
#> ** Reported on training data. **
#> ** Metrics reported on full training frame **
#>
#> MSE: 0.0321
#> RMSE: 0.179
#> LogLoss: 0.117
#> Mean Per-Class Error: 0.0336
#> AUC: 0.984
#> pr_auc: 0.964
#> Gini: 0.967
#>
#> Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
#>           arrhythmia healthy   Error    Rate
#> arrhythmia          155      9 0.054878 =9/164
#> healthy              2     161 0.012270 =2/163
#> Totals              157     170 0.033639 =11/327
#>
#> Maximum Metrics: Maximum metrics at their respective thresholds
#>           metric threshold   value idx
#> 1             max f1 0.684745 0.966967 169
#> 2             max f2 0.684745 0.979319 169
#> 3             max f0point5 0.922844 0.972046 155
#> 4             max accuracy 0.845474 0.966361 163
#> 5             max precision 0.991717 1.000000 0
#> 6             max recall 0.137131 1.000000 181
#> 7             max specificity 0.991717 1.000000 0
#> 8             max absolute_mcc 0.684745 0.933586 169
#> 9 max min_per_class_accuracy 0.845474 0.963415 163
#> 10 max mean_per_class_accuracy 0.684745 0.966426 169
#>
#> Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T>)
#> H2OBinomialMetrics: deeplearning
#> ** Reported on validation data. **
#> ** Metrics reported on full validation frame **
#>
#> MSE: 0.182
#> RMSE: 0.427
#> LogLoss: 0.976
#> Mean Per-Class Error: 0.192
#> AUC: 0.884
#> pr_auc: 0.902
#> Gini: 0.767
#>
#> Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
#>           arrhythmia healthy   Error    Rate
#> arrhythmia          16      9 0.360000 =9/25
#> healthy              1     42 0.023256 =1/43
#> Totals              17     51 0.147059 =10/68
#>
#> Maximum Metrics: Maximum metrics at their respective thresholds
#>           metric threshold   value idx

```

```

#> 1           max f1  0.000686 0.893617  50
#> 2           max f2  0.000686 0.941704  50
#> 3           max f0point5 0.786678 0.871795  37
#> 4           max accuracy 0.000686 0.852941  50
#> 5           max precision 0.992411 1.000000  0
#> 6           max recall  0.000001 1.000000  57
#> 7           max specificity 0.992411 1.000000  0
#> 8           max absolute_mcc 0.000686 0.686752  50
#> 9   max min_per_class_accuracy 0.786678 0.790698  37
#> 10  max mean_per_class_accuracy 0.786678 0.815349  37
#>
#> Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T/`
#> H2OBinomialMetrics: deeplearning
#> ** Reported on cross-validation data. **
#> ** 15-fold cross-validation on training data (Metrics computed for combined holdout predictions) **
#>
#> MSE: 0.168
#> RMSE: 0.41
#> LogLoss: 0.557
#> Mean Per-Class Error: 0.208
#> AUC: 0.848
#> pr_auc: 0.81
#> Gini: 0.697
#>
#> Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
#>
#> arrhythmia healthy    Error      Rate
#> arrhythmia        120     44 0.268293  =44/164
#> healthy            24    139 0.147239  =24/163
#> Totals             144    183 0.207951  =68/327
#>
#> Maximum Metrics: Maximum metrics at their respective thresholds
#>
#> metric threshold      value idx
#> 1           max f1  0.442932 0.803468 182
#> 2           max f2  0.012515 0.884956 251
#> 3           max f0point5 0.624456 0.792453 157
#> 4           max accuracy 0.467967 0.792049 178
#> 5           max precision 0.990572 1.000000  0
#> 6           max recall  0.001848 1.000000 280
#> 7           max specificity 0.990572 1.000000  0
#> 8           max absolute_mcc 0.442932 0.588667 182
#> 9   max min_per_class_accuracy 0.607830 0.785276 161
#> 10  max mean_per_class_accuracy 0.442932 0.792234 182
#>
#> Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T/`
#> Cross-Validation Metrics Summary:
#>
#>          mean         sd cv_1_valid  cv_2_valid
#> accuracy 0.8385412 0.061696008 0.64285713 0.8235294
#> auc       0.85863346 0.06217786 0.69518715 0.73333335
#> err       0.16145879 0.061696008 0.35714287 0.1764706
#> err_count 3.8666666 1.9298819 10.0      3.0
#> f0point5 0.82062894 0.08774206 0.57894737 0.6896552
#> f1        0.84875447 0.057496537 0.6875 0.72727275
#> f2        0.8901033 0.04788143 0.84615386 0.7692308

```

```

#> lift_top_group      1.7281693  0.54166335  2.5454545      0.0
#> logloss            0.5452315  0.1306371   0.6586832  0.7813473
#> max_per_class_error 0.30671087 0.113830075  0.5882353      0.2
#> mcc                0.7049009  0.10066028  0.46442038  0.60385966
#> mean_per_class_accuracy 0.8367473  0.05778133  0.7058824  0.81666666
#> mean_per_class_error  0.16325273  0.05778133  0.29411766  0.18333334
#> mse                0.16131651  0.03845805  0.23015584  0.21473385
#> precision           0.80784994  0.110800184 0.52380955  0.66666667
#> r2                  0.32223144  0.16513458  0.03506859 -0.034301396
#> recall              0.9275503  0.06717654      1.0      0.8
#> rmse                0.3942398  0.054274667  0.4797456  0.46339384
#> specificity          0.74594426  0.14763153  0.4117647  0.83333333
#>
#> cv_3_valid  cv_4_valid  cv_5_valid  cv_6_valid
#> accuracy            0.93333334  0.82608694  0.94444444  1.0
#> auc                 0.962963  0.85714287  0.96103895  1.0
#> err                 0.06666667  0.17391305  0.055555556 0.0
#> err_count            1.0          4.0          1.0      0.0
#> f0point5             0.9183673  0.81395346  0.98039216  1.0
#> f1                  0.94736844  0.875          0.95238096  1.0
#> f2                  0.9782609  0.9459459   0.9259259  1.0
#> lift_top_group       1.6666666  1.6428572  1.6363636  2.0
#> logloss              0.3135496  0.4296336   0.3291604  0.11736608
#> max_per_class_error  0.16666667  0.44444445  0.09090909 0.0
#> mcc                 0.8660254  0.6573422   0.8918826  1.0
#> mean_per_class_accuracy 0.9166667  0.77777778  0.95454544  1.0
#> mean_per_class_error  0.083333336 0.22222222  0.045454547 0.0
#> mse                 0.104235224 0.1389879  0.090775445 0.034521867
#> precision            0.9          0.77777778      1.0      1.0
#> r2                  0.5656866  0.41647142  0.6180358  0.86191255
#> recall              1.0          1.0          0.90909094 1.0
#> rmse                0.32285482 0.3728108  0.30128965 0.18580061
#> specificity          0.8333333  0.55555556      1.0      1.0
#>
#> cv_7_valid  cv_8_valid  cv_9_valid  cv_10_valid
#> accuracy            0.90909094  0.90909094  0.8076923  0.8
#> auc                 0.90178573  0.9285714  0.86928105 0.83035713
#> err                 0.09090909  0.09090909  0.1923077  0.2
#> err_count            2.0          1.0          5.0      6.0
#> f0point5             0.9375    0.9677419  0.82474226 0.75581396
#> f1                  0.85714287 0.9230769  0.8648649  0.8125
#> f2                  0.7894737  0.88235295 0.90909094 0.8783784
#> lift_top_group       2.75      1.5714285  1.5294118  2.142857
#> logloss              0.52325845 0.74918216 0.63733953 0.6056329
#> max_per_class_error  0.25      0.14285715 0.44444445  0.3125
#> mcc                 0.81009257 0.8280787  0.5608894  0.62737644
#> mean_per_class_accuracy 0.875    0.9285714  0.748366  0.80803573
#> mean_per_class_error  0.125    0.071428575 0.251634  0.19196428
#> mse                 0.14090821 0.16433945 0.16959226 0.18447527
#> precision            1.0          1.0          0.8      0.7222222
#> r2                  0.39107522 0.2898188  0.25069037 0.25880474
#> recall              0.75      0.85714287  0.9411765  0.9285714
#> rmse                0.37537742 0.40538803 0.41181582 0.42950583
#> specificity          1.0          1.0  0.5555556  0.6875
#>
#> cv_11_valid  cv_12_valid  cv_13_valid  cv_14_valid

```

```

#> accuracy           0.8636364   0.7619048       0.75   0.8064516
#> auc                0.90082645  0.7181818      0.8125   0.875
#> err                0.13636364  0.23809524     0.25   0.19354838
#> err_count          3.0          5.0          8.0      6.0
#> f0point5          0.9302326   0.71428573  0.71428573  0.7692308
#> f1                 0.84210527  0.8          0.8      0.84210527
#> f2                 0.7692308   0.90909094  0.90909094  0.9302326
#> lift_top_group     2.0          0.0          2.0      1.9375
#> logloss            0.52656233  0.8181485   0.66666746  0.49635586
#> max_per_class_error 0.27272728  0.45454547     0.5      0.4
#> mcc                0.75592893  0.6030227   0.57735026  0.66057825
#> mean_per_class_accuracy 0.8636364   0.77272725     0.75      0.8
#> mean_per_class_error 0.13636364  0.22727273     0.25      0.2
#> mse                0.15577565  0.24097086  0.22684109  0.15012316
#> precision          1.0          0.6666667   0.6666667   0.72727275
#> r2                 0.3768974   0.033925887  0.092635654  0.39888185
#> recall              0.72727275  1.0          1.0      1.0
#> rmse               0.39468426  0.49088785  0.47627836  0.3874573
#> specificity         1.0          0.54545456     0.5      0.6
#>
#> cv_15_valid
#> accuracy           0.8
#> auc                0.8333333
#> err                0.2
#> err_count          3.0
#> f0point5          0.71428573
#> f1                 0.8
#> f2                 0.90909094
#> lift_top_group     2.5
#> logloss            0.5255857
#> max_per_class_error 0.33333334
#> mcc                0.6666667
#> mean_per_class_accuracy 0.8333333
#> mean_per_class_error 0.16666667
#> mse                0.17331165
#> precision          0.6666667
#> r2                 0.2778681
#> recall              1.0
#> rmse               0.41630718
#> specificity         0.6666667
#>
#> Scoring History:
#>             timestamp      duration training_speed    epochs
#> 1  2019-09-20 15:09:41  0.000 sec        NA  0.00000
#> 2  2019-09-20 15:09:42  1 min 26.356 sec  5597 obs/sec 10.72013
#> 3  2019-09-20 15:09:42  1 min 26.981 sec  5715 obs/sec 21.44025
#> 4  2019-09-20 15:09:43  1 min 27.544 sec  5977 obs/sec 32.16038
#> 5  2019-09-20 15:09:44  1 min 28.145 sec  6017 obs/sec 42.88050
#> 6  2019-09-20 15:09:44  1 min 28.698 sec  6140 obs/sec 53.60063
#> 7  2019-09-20 15:09:45  1 min 29.245 sec  6234 obs/sec 64.32075
#> 8  2019-09-20 15:09:45  1 min 29.790 sec  6319 obs/sec 75.04088
#> 9  2019-09-20 15:09:46  1 min 30.319 sec  6401 obs/sec 85.76101
#> 10 2019-09-20 15:09:46  1 min 30.852 sec  6460 obs/sec 96.48113
#> 11 2019-09-20 15:09:47  1 min 31.392 sec  6503 obs/sec 107.20126

```

```

#> iterations      samples training_rmse training_logloss training_r2
#> 1          0    0.000000            NA            NA            NA
#> 2          1   3409.000000   0.37355  0.46858  0.44184
#> 3          2   6818.000000   0.35218  0.47369  0.50387
#> 4          3  10227.000000   0.31384  0.35018  0.60602
#> 5          4  13636.000000   0.29469  0.30311  0.65263
#> 6          5  17045.000000   0.28485  0.29392  0.67544
#> 7          6  20454.000000   0.25468  0.25312  0.74055
#> 8          7  23863.000000   0.21535  0.16877  0.81450
#> 9          8  27272.000000   0.24487  0.21845  0.76014
#> 10         9  30681.000000   0.18801  0.13523  0.85861
#> 11        10 34090.000000   0.17923  0.11736  0.87151
#> training_auc training_pr_auc training_lift
#> 1           NA            NA            NA
#> 2          0.88695  0.85448  2.00613
#> 3          0.91179  0.87339  1.50460
#> 4          0.93835  0.90497  2.00613
#> 5          0.94692  0.91555  2.00613
#> 6          0.95432  0.92980  2.00613
#> 7          0.96832  0.94387  2.00613
#> 8          0.97366  0.95149  2.00613
#> 9          0.98077  0.96509  2.00613
#> 10         0.98227  0.96737  2.00613
#> 11         0.98365  0.96430  2.00613
#> training_classification_error validation_rmse validation_logloss
#> 1           NA            NA            NA
#> 2          0.18043  0.39644  0.58228
#> 3          0.15291  0.40302  0.65206
#> 4          0.11621  0.41669  0.71722
#> 5          0.09480  0.40738  0.63853
#> 6          0.09480  0.40913  0.80101
#> 7          0.06728  0.41996  0.89227
#> 8          0.05505  0.42863  0.98097
#> 9          0.04587  0.46062  1.26800
#> 10         0.03364  0.44414  1.13089
#> 11         0.03364  0.42686  0.97588
#> validation_r2 validation_auc validation_pr_auc validation_lift
#> 1           NA            NA            NA            NA
#> 2          0.32398  0.83907  0.84912  1.58140
#> 3          0.30136  0.87535  0.88555  1.58140
#> 4          0.25316  0.86884  0.88149  1.58140
#> 5          0.28613  0.86791  0.88102  1.58140
#> 6          0.28001  0.87628  0.88782  1.58140
#> 7          0.24137  0.85674  0.86697  1.58140
#> 8          0.20974  0.85953  0.87294  1.58140
#> 9          0.08735  0.84744  0.85773  1.58140
#> 10         0.15151  0.83442  0.85225  1.58140
#> 11         0.21624  0.88372  0.90172  1.58140
#> validation_classification_error
#> 1           NA
#> 2          0.19118
#> 3          0.16176
#> 4          0.17647

```

```

#> 5          0.16176
#> 6          0.17647
#> 7          0.16176
#> 8          0.17647
#> 9          0.16176
#> 10         0.16176
#> 11         0.14706
#>
#> Variable Importances: (Extract with `h2o.varimp`)
#> =====
#>
#> Variable Importances:
#>   variable relative_importance scaled_importance percentage
#> 1     V169      1.000000      1.000000  0.014302
#> 2     V136      0.922611      0.922611  0.013195
#> 3     V239      0.913344      0.913344  0.013062
#> 4     V103      0.910889      0.910889  0.013027
#> 5     V229      0.893744      0.893744  0.012782
#>
#> ---
#>   variable relative_importance scaled_importance percentage
#> 85    V259      0.700921      0.700921  0.010024
#> 86    V168      0.698275      0.698275  0.009987
#> 87    V179      0.695452      0.695452  0.009946
#> 88    V269      0.695200      0.695200  0.009943
#> 89    V45       0.683744      0.683744  0.009779
#> 90    V33       0.664106      0.664106  0.009498

```

One performance metric we are usually interested in is the mean per class error for training and validation data.

```

h2o.mean_per_class_error(dl_model, train = TRUE, valid = TRUE, xval = TRUE)
#> train valid xval
#> 0.0336 0.1916 0.2078

```

The confusion matrix tells us, how many classes have been predicted correctly and how many predictions were accurate. Here, we see the errors in predictions on validation data.

```

h2o.confusionMatrix(dl_model, valid = TRUE)
#> Confusion Matrix (vertical: actual; across: predicted)  for max f1 @ threshold = 0.00068617013646127
#>           arrhythmia healthy Error Rate
#> arrhythmia            16     9 0.360000  =9/25
#> healthy                1    42 0.023256  =1/43
#> Totals                 17    51 0.147059  =10/68

```

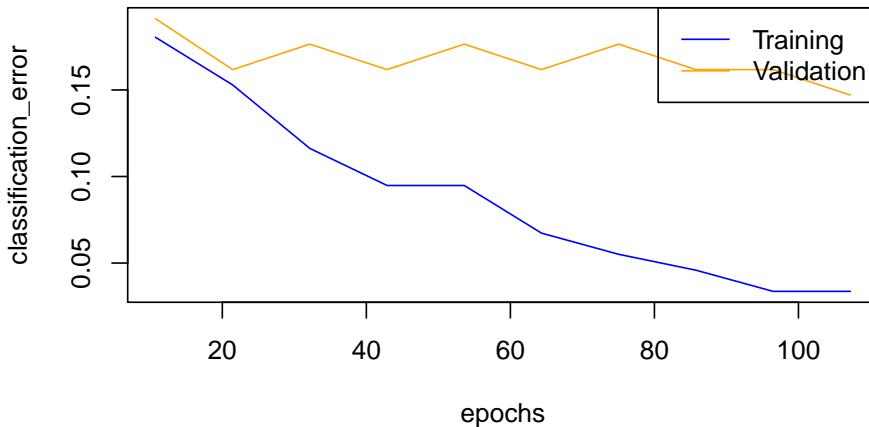
We can also plot the classification error over all epochs or samples.

```

plot(dl_model,
      timestep = "epochs",
      metric = "classification_error")

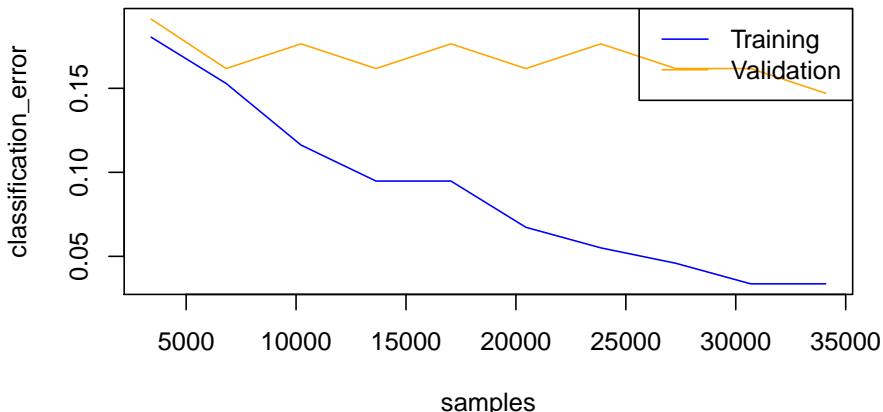
```

Scoring History



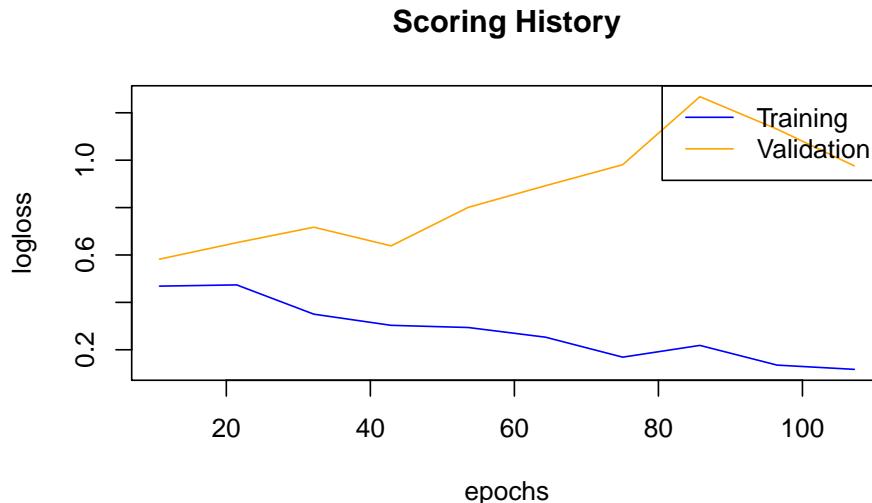
```
plot(dl_model,
      timestep = "samples",
      metric = "classification_error")
```

Scoring History



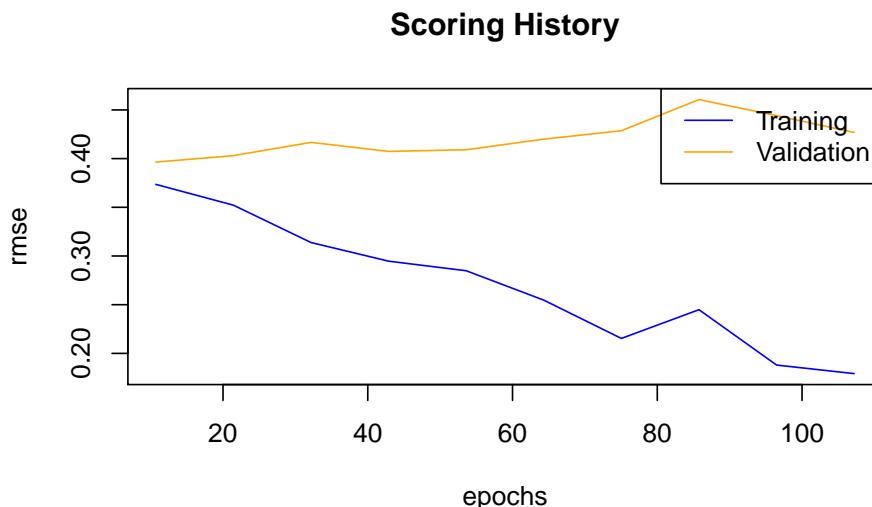
Next to the classification error, we are usually interested in the logistic loss (negative log-likelihood or log loss). It describes the sum of errors for each sample in the training or validation data or the negative logarithm of the likelihood of error for a given prediction/ classification. Simply put, the lower the loss, the better the model (if we ignore potential overfitting).

```
plot(dl_model,
      timestep = "epochs",
      metric = "logloss")
```



We can also plot the mean squared error (MSE). The **MSE** tells us the average of the prediction errors squared, i.e. the estimator's variance and bias. The closer to zero, the better a model.

```
plot(dl_model,
      timestep = "epochs",
      metric = "rmse")
```



Next, we want to know the area under the curve (AUC). **AUC** is an important metric for measuring binary classification model performances. It gives the area under the curve, i.e. the integral, of true positive vs false positive rates. The closer to 1, the better a model.

```
h2o.auc(dl_model, train = TRUE)
#> [1] 0.984
```

```
h2o.auc(dl_model, valid = TRUE)
#> [1] 0.884
```

```
h2o.auc(dl_model, xval = TRUE)
#> [1] 0.848
```

The weights for connecting two adjacent layers and per-neuron biases that we specified the model to save, can be accessed with:

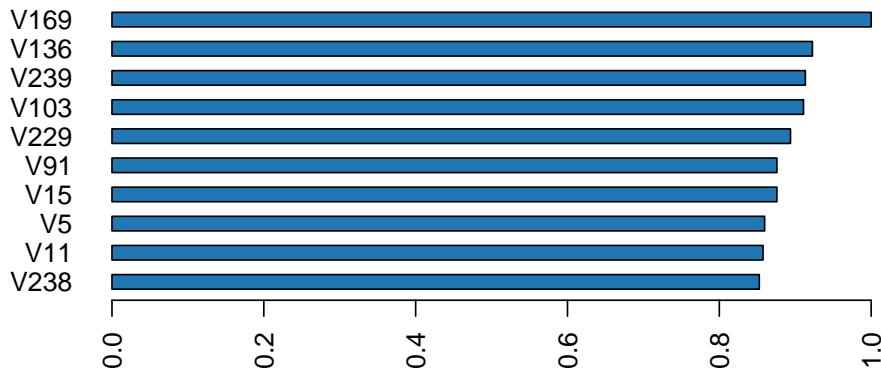
```
w <- h2o.weights(dl_model, matrix_id = 1)
b <- h2o.biases(dl_model, vector_id = 1)
```

Variable importance can be extracted as well (but keep in mind, that variable importance in deep neural networks is difficult to assess and should be considered only as rough estimates).

```
h2o.varimp(dl_model)
#> Variable Importances:
#>   variable relative_importance scaled_importance percentage
#> 1      V169          1.000000     1.000000    0.014302
#> 2      V136          0.922611     0.922611    0.013195
#> 3      V239          0.913344     0.913344    0.013062
#> 4      V103          0.910889     0.910889    0.013027
#> 5      V229          0.893744     0.893744    0.012782
#>
#> ---
#>   variable relative_importance scaled_importance percentage
#> 85     V259          0.700921     0.700921    0.010024
#> 86     V168          0.698275     0.698275    0.009987
#> 87     V179          0.695452     0.695452    0.009946
#> 88     V269          0.695200     0.695200    0.009943
#> 89     V45           0.683744     0.683744    0.009779
#> 90     V33           0.664106     0.664106    0.009498
```

```
h2o.varimp_plot(dl_model)
```

Variable Importance: Deep Learning



36.7 Test data

Now that we have a good idea about model performance on validation data, we want to know how it performed on unseen test data. A good model should find an optimal balance between accuracy on training and test data. A model that has 0% error on the training data but 40% error on the test data is in effect useless. It overfit on the training data and is thus not able to generalize to unknown data.

```
perf <- h2o.performance(dl_model, test)
perf
#> H2OBinomialMetrics: deeplearning
#>
#> MSE:  0.257
```

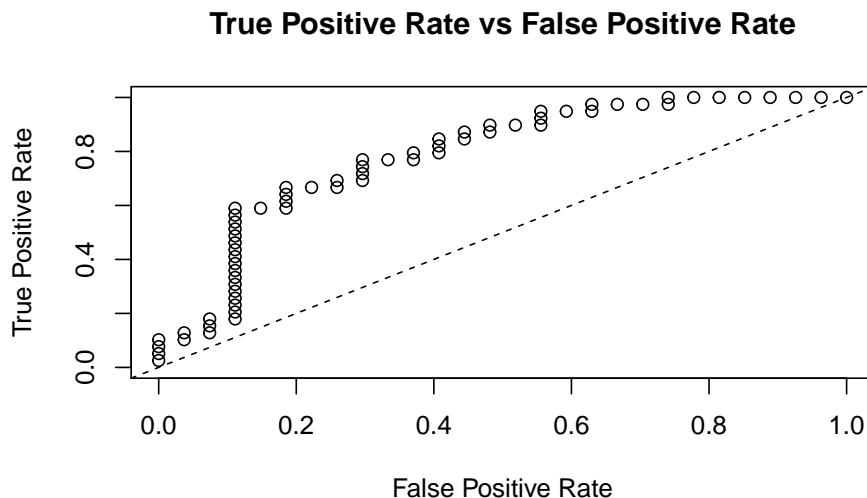
```

#> RMSE: 0.507
#> LogLoss: 1.75
#> Mean Per-Class Error: 0.303
#> AUC: 0.788
#> pr_auc: 0.783
#> Gini: 0.576
#>
#> Confusion Matrix (vertical: actual; across: predicted) for F1-optimal threshold:
#>
#>           arrhythmia healthy   Error    Rate
#> arrhythmia          12      15 0.555556 =15/27
#> healthy              2      37 0.051282 =2/39
#> Totals              14      52 0.257576 =17/66
#>
#> Maximum Metrics: Maximum metrics at their respective thresholds
#>
#>           metric threshold     value idx
#> 1          max f1 0.000002 0.813187 51
#> 2          max f2 0.000000 0.906977 58
#> 3          max f0point5 0.977116 0.804196 25
#> 4          max accuracy 0.400678 0.742424 37
#> 5          max precision 0.990653 1.000000  0
#> 6          max recall 0.000000 1.000000 58
#> 7          max specificity 0.990653 1.000000  0
#> 8          max absolute_mcc 0.977116 0.481615 25
#> 9  max min_per_class_accuracy 0.824797 0.703704 35
#> 10 max mean_per_class_accuracy 0.965774 0.740741 30
#>
#> Gains/Lift Table: Extract with `h2o.gainsLift(<model>, <data>)` or `h2o.gainsLift(<model>, valid=<T>)

```

Plotting the test performance's AUC plot shows us approximately how good the predictions are.

```
plot(perf)
```



We also want to know the log loss, MSE and AUC values, as well as other model metrics for the test data:

```
h2o.logloss(perf)
#> [1] 1.75
```

```
h2o.mse(perf)
#> [1] 0.257
```

```

h2o.auc(perf)
#> [1] 0.788

head(h2o.metric(perf))
#> Metrics for Thresholds: Binomial metrics as a function of classification thresholds
#>   threshold      f1      f2 f0points5 accuracy precision    recall
#> 1  0.990653 0.050000 0.031847 0.116279 0.424242 1.000000 0.025641
#> 2  0.990156 0.097561 0.063291 0.212766 0.439394 1.000000 0.051282
#> 3  0.989233 0.142857 0.094340 0.294118 0.454545 1.000000 0.076923
#> 4  0.989156 0.186047 0.125000 0.363636 0.469697 1.000000 0.102564
#> 5  0.988960 0.181818 0.124224 0.338983 0.454545 0.800000 0.102564
#> 6  0.988743 0.222222 0.154321 0.396825 0.469697 0.833333 0.128205
#> specificity absolute_mcc min_per_class_accuracy mean_per_class_accuracy
#> 1  1.000000 0.103203 0.025641 0.512821
#> 2  1.000000 0.147087 0.051282 0.525641
#> 3  1.000000 0.181568 0.076923 0.538462
#> 4  1.000000 0.211341 0.102564 0.551282
#> 5  0.962963 0.121754 0.102564 0.532764
#> 6  0.962963 0.155921 0.128205 0.545584
#>   tns fns fps tps      tnr      fnr      fpr      tpr idx
#> 1  27  38   0   1 1.000000 0.974359 0.000000 0.025641  0
#> 2  27  37   0   2 1.000000 0.948718 0.000000 0.051282  1
#> 3  27  36   0   3 1.000000 0.923077 0.000000 0.076923  2
#> 4  27  35   0   4 1.000000 0.897436 0.000000 0.102564  3
#> 5  26  35   1   4 0.962963 0.897436 0.037037 0.102564  4
#> 6  26  34   1   5 0.962963 0.871795 0.037037 0.128205  5

```

The confusion matrix alone can be seen with the `h2o.confusionMatrix()` function, but it is also part of the performance summary.

```

h2o.confusionMatrix(dl_model, test)
#> Confusion Matrix (vertical: actual; across: predicted)  for max f1 @ threshold = 2.48957253598693e-0
#>                               arrhythmia healthy Error Rate
#> arrhythmia                  12      15 0.5555556 =15/27
#> healthy                      2      37 0.051282 =2/39
#> Totals                      14      52 0.257576 =17/66

```

The final predictions with probabilities can be extracted with the `h2o.predict()` function. Beware though, that the number of correct and wrong classifications can be slightly different from the confusion matrix above.

Here, I combine the predictions with the actual test diagnoses and classes into a data frame. For plotting I also want to have a column, that tells me whether the predictions were correct. By default, a prediction probability above 0.5 will get scored as a prediction for the respective category. I find it often makes sense to be more stringent with this, though and set a higher threshold. Therefore, I am creating another column with stringent predictions, where I only count predictions that were made with more than 80% probability. Everything that does not fall within this range gets scored as “uncertain”. For these stringent predictions, I am also creating a column that tells me whether they were accurate.

```

|-----| 100%
finalRf_predictions$accurate <- ifelse(
  finalRf_predictions$actual == finalRf_predictions$predict, "yes", "no")

finalRf_predictions$predict_stringent <- ifelse(
  finalRf_predictions$arrhythmia > 0.8, "arrhythmia",
  ifelse(finalRf_predictions$healthy > 0.8, "healthy", "uncertain"))

finalRf_predictions$accurate_stringent <- ifelse(
  finalRf_predictions$actual == finalRf_predictions$predict_stringent, "yes",
  ifelse(finalRf_predictions$predict_stringent == "uncertain", "na", "no"))

finalRf_predictions %>%
  group_by(actual, predict) %>%
  summarise(n = n())
#> # A tibble: 4 x 3
#> # Groups:   actual [2]
#>   actual      predict     n
#>   <fct>      <fct>     <int>
#> 1 arrhythmia arrhythmia    15
#> 2 arrhythmia healthy      12
#> 3 healthy    arrhythmia     6
#> 4 healthy    healthy       33

finalRf_predictions %>%
  group_by(actual, predict_stringent) %>%
  summarise(n = n())
#> # A tibble: 5 x 3
#> # Groups:   actual [2]
#>   actual      predict_stringent     n
#>   <fct>      <chr>           <int>
#> 1 arrhythmia arrhythmia          19
#> 2 arrhythmia healthy            8
#> 3 healthy    arrhythmia          9
#> 4 healthy    healthy             28
#> 5 healthy    uncertain            2

```

To get a better overview, I am going to plot the predictions (default and stringent):

```

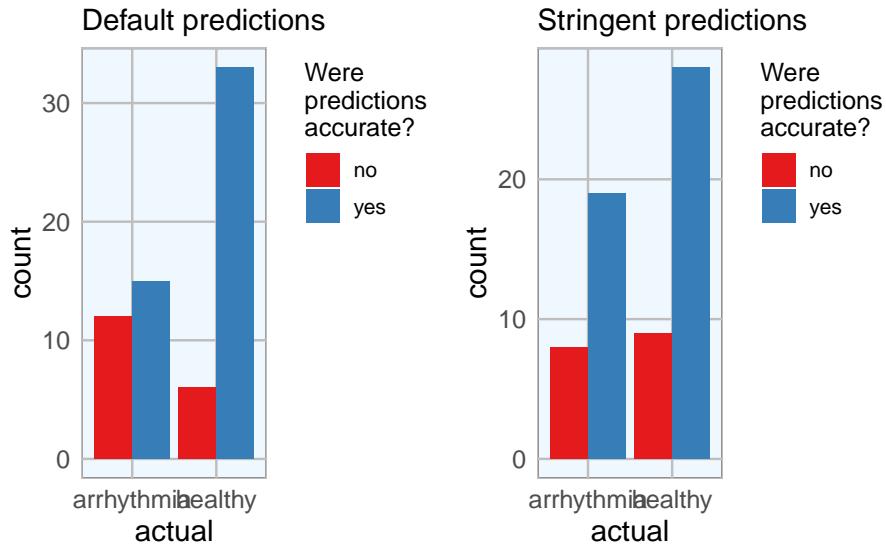
p1 <- finalRf_predictions %>%
  ggplot(aes(x = actual, fill = accurate)) +
  geom_bar(position = "dodge") +
  scale_fill_brewer(palette = "Set1") +
  my_theme() +
  labs(fill = "Were\npredictions\naccurate?", title = "Default predictions")

p2 <- finalRf_predictions %>%
  subset(accurate_stringent != "na") %>%
  ggplot(aes(x = actual, fill = accurate_stringent)) +
  geom_bar(position = "dodge") +
  scale_fill_brewer(palette = "Set1") +

```

```
my_theme() +
  labs(fill = "Were\nnpredictions\nnaccurate?",
       title = "Stringent predictions")

grid.arrange(p1, p2, ncol = 2)
```



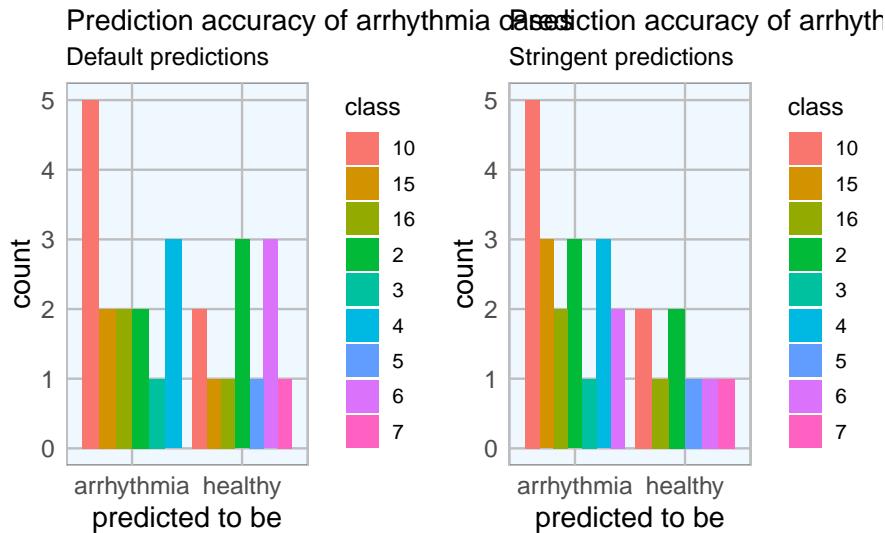
Being more stringent with the prediction threshold slightly reduced the number of errors but not by much.

I also want to know whether there are certain classes of arrhythmia that are especially prone to being misclassified:

```
p1 <- subset(finalRf_predictions, actual == "arrhythmia") %>%
  ggplot(aes(x = predict, fill = class)) +
  geom_bar(position = "dodge") +
  my_theme() +
  labs(title = "Prediction accuracy of arrhythmia cases",
       subtitle = "Default predictions",
       x = "predicted to be")

p2 <- subset(finalRf_predictions, actual == "arrhythmia") %>%
  ggplot(aes(x = predict_stringent, fill = class)) +
  geom_bar(position = "dodge") +
  my_theme() +
  labs(title = "Prediction accuracy of arrhythmia cases",
       subtitle = "Stringent predictions",
       x = "predicted to be")

grid.arrange(p1, p2, ncol = 2)
```



There are no obvious biases towards some classes but with the small number of samples for most classes, this is difficult to assess.

36.8 Final conclusions: How useful is the model?

Most samples were classified correctly, but the total error was not particularly good. Moreover, when evaluating the usefulness of a specific model, we need to keep in mind what we want to achieve with it and which questions we want to answer. If we wanted to deploy this model in a clinical setting, it should assist with diagnosing patients. So, we need to think about what the consequences of wrong classifications would be. Would it be better to optimize for high sensitivity, in this example as many arrhythmia cases as possible get detected - with the drawback that we probably also diagnose a few healthy people? Or do we want to maximize precision, meaning that we could be confident that a patient who got predicted to have arrhythmia does indeed have it, while accepting that a few arrhythmia cases would remain undiagnosed? When we consider stringent predictions, this model correctly classified 19 out of 27 arrhythmia cases, but 6 were misdiagnosed. This would mean that some patients who were actually sick, wouldn't have gotten the correct treatment (if decided solely based on this model). For real-life application, this is obviously not sufficient!

Next week, I'll be trying to improve the model by doing a grid search for hyper-parameter tuning.

So, stay tuned... (sorry, couldn't resist ;-))

```
sessionInfo()
#> R version 3.6.0 (2019-04-26)
#> Platform: x86_64-pc-linux-gnu (64-bit)
#> Running under: Ubuntu 18.04.3 LTS
#>
#> Matrix products: default
#> BLAS/LAPACK: /usr/lib/x86_64-linux-gnu/libopenblas-r0.2.20.so
#>
#> locale:
#> [1] LC_CTYPE=en_US.UTF-8          LC_NUMERIC=C
#> [3] LC_TIME=en_US.UTF-8          LC_COLLATE=en_US.UTF-8
#> [5] LC_MONETARY=en_US.UTF-8      LC_MESSAGES=en_US.UTF-8
#> [7] LC_PAPER=en_US.UTF-8        LC_NAME=C
#> [9] LC_ADDRESS=C                 LC_TELEPHONE=C
#> [11] LC_MEASUREMENT=en_US.UTF-8 LC_IDENTIFICATION=C
```

```
#>
#> attached base packages:
#> [1] stats4      parallel   grid       stats      graphics  grDevices utils
#> [8] datasets    methods    base
#>
#> other attached packages:
#> [1] reshape2_1.4.3      tidyverse_0.8.3        matrixStats_0.54.0
#> [4] pcaGoPromoter_1.28.0 Biostrings_2.52.0      XVector_0.24.0
#> [7] IRanges_2.18.0      S4Vectors_0.22.0      BiocGenerics_0.30.0
#> [10] ellipse_0.4.1      gridExtra_2.3        ggrepel_0.8.1
#> [13] ggplot2_3.1.1      h2o_3.22.1.1        dplyr_0.8.0.1
#> [16] logging_0.9-107
#>
#> loaded via a namespace (and not attached):
#> [1] Rcpp_1.0.1           assertthat_0.2.1     zeallot_0.1.0
#> [4] rprojroot_1.3-2      digest_0.6.18        utf8_1.1.4
#> [7] R6_2.4.0             plyr_1.8.4          backports_1.1.4
#> [10] RSQLite_2.1.1        evaluate_0.13       pillar_1.4.0
#> [13] zlibbioc_1.30.0     rlang_0.3.4          lazyeval_0.2.2
#> [16] rstudioapi_0.10     blob_1.1.1          rmarkdown_1.12
#> [19] labeling_0.3         stringr_1.4.0       RCurl_1.95-4.12
#> [22] bit_1.1-14          munsell_0.5.0       compiler_3.6.0
#> [25] xfun_0.6            pkgconfig_2.0.2     htmltools_0.3.6
#> [28] tidyselect_0.2.5     tibble_2.1.1         bookdown_0.10
#> [31] fansi_0.4.0          crayon_1.3.4        withr_2.1.2
#> [34] bitops_1.0-6         jsonlite_1.6        gtable_0.3.0
#> [37] DBI_1.0.0            magrittr_1.5        scales_1.0.0
#> [40] cli_1.1.0            stringi_1.4.3      vctrs_0.1.0
#> [43] RColorBrewer_1.1-2  tools_3.6.0          bit64_0.9-7
#> [46] Biobase_2.44.0       glue_1.3.1          purrrr_0.3.2
#> [49] yaml_2.2.0            AnnotationDbi_1.46.0 colorspace_1.4-1
#> [52] memoise_1.1.0        knitr_1.22
```


Chapter 37

Credit Scoring

37.1 Introduction

Source: <https://www.r-bloggers.com/using-neural-networks-for-credit-scoring-a-simple-example/>

37.2 Motivation

Credit scoring is the practice of analysing a persons background and credit application in order to assess the creditworthiness of the person. One can take numerous approaches on analysing this creditworthiness. In the end it basically comes down to first selecting the correct independent variables (e.g. income, age, gender) that lead to a given level of creditworthiness.

In other words: `creditworthiness = f(income, age, gender, ...)`.

A creditscoring system can be represented by linear regression, logistic regression, machine learning or a combination of these. Neural networks are situated in the domain of machine learning. The following is an strongly simplified example. The actual procedure of building a credit scoring system is much more complex and the resulting model will most likely not consist of solely or even a neural network.

If you're unsure on what a neural network exactly is, I find this a good place to start.

For this example the R package `neuralnet` is used, for a more in-depth view on the exact workings of the package see `neuralnet: Training of Neural Networks` by F. Günther and S. Fritsch.

37.3 load the data

Dataset downloaded: <https://gist.github.com/Bart6114/8675941#file-creditset-csv>

```
set.seed(1234567890)

library(neuralnet)

dataset <- read.csv(file.path(data_raw_dir, "creditset.csv"))
head(dataset)
#>   clientid income  age    loan      LTI default10yr
#> 1       1 66156 59.0 8106.5 0.122537        0
#> 2       2 34415 48.1 6564.7 0.190752        0
```

```

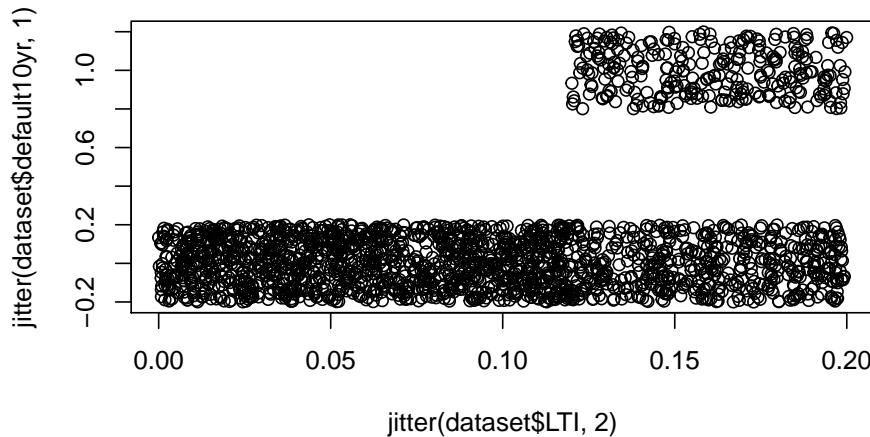
#> 3      3 57317 63.1 8021.0 0.139940          0
#> 4      4 42710 45.8 6103.6 0.142911          0
#> 5      5 66953 18.6 8770.1 0.130989          1
#> 6      6 24904 57.5   15.5 0.000622          0

names(dataset)
#> [1] "clientid"     "income"        "age"           "loan"          "LTI"
#> [6] "default10yr"

summary(dataset)
#>   clientid       income        age        loan
#>   Min. : 1   Min. :20014   Min. :18.1   Min. : 1
#>   1st Qu.: 501 1st Qu.:32796  1st Qu.:29.1  1st Qu.: 1940
#>   Median :1000 Median :45789  Median :41.4   Median : 3975
#>   Mean   :1000  Mean   :45332  Mean   :40.9   Mean   : 4444
#>   3rd Qu.:1500 3rd Qu.:577791 3rd Qu.:52.6  3rd Qu.: 6432
#>   Max.   :2000  Max.   :69996  Max.   :64.0   Max.   :13766
#>   LTI            default10yr
#>   Min.   :0.0000  Min.   :0.000
#>   1st Qu.:0.0479  1st Qu.:0.000
#>   Median :0.0994  Median :0.000
#>   Mean   :0.0984  Mean   :0.142
#>   3rd Qu.:0.1476  3rd Qu.:0.000
#>   Max.   :0.1999  Max.   :1.000

# distribution of defaults
table(dataset$default10yr)
#>
#> 0    1
#> 1717 283
min(dataset$LTI)
#> [1] 4.91e-05
plot(jitter(dataset$default10yr, 1) ~ jitter(dataset$LTI, 2))

```

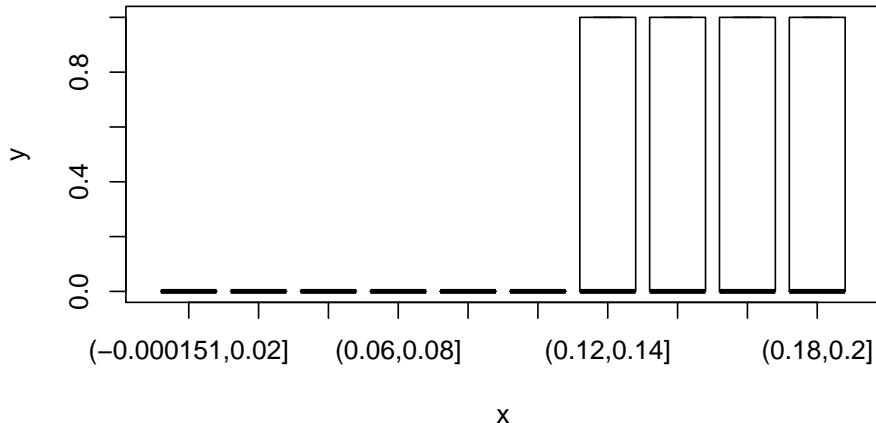


```

# convert LTI continuous variable to categorical
dataset$LTIrng <- cut(dataset$LTI, breaks = 10)
unique(dataset$LTIrng)
#> [1] (0.12,0.14]      (0.14,0.16]      (-0.000151,0.02]
#> [5] (0.1,0.12]       (0.04,0.06]      (0.06,0.08]      (0.08,0.1]
#> [9] (0.16,0.18]      (0.02,0.04]

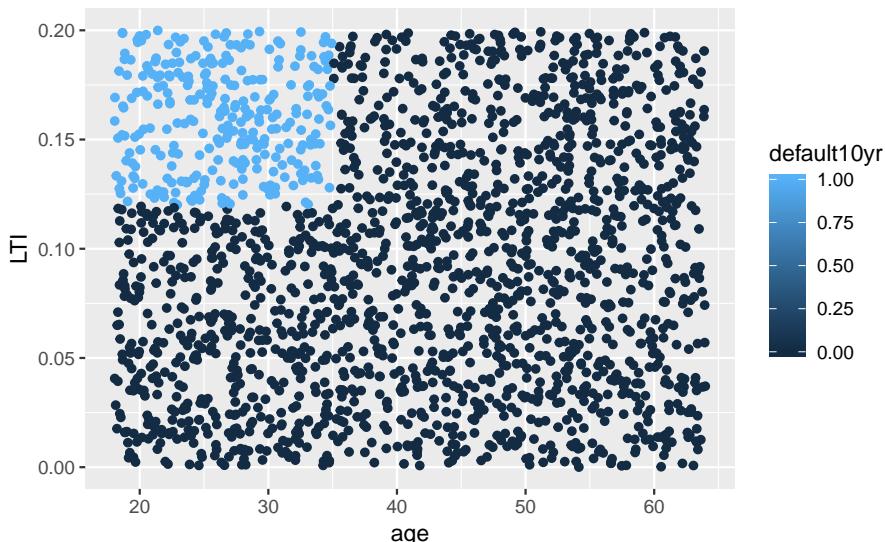
```

```
#> 10 Levels: (-0.000151,0.02] (0.02,0.04] (0.04,0.06] ... (0.18,0.2]
plot(dataset$LTIrng, dataset$default10yr)
```



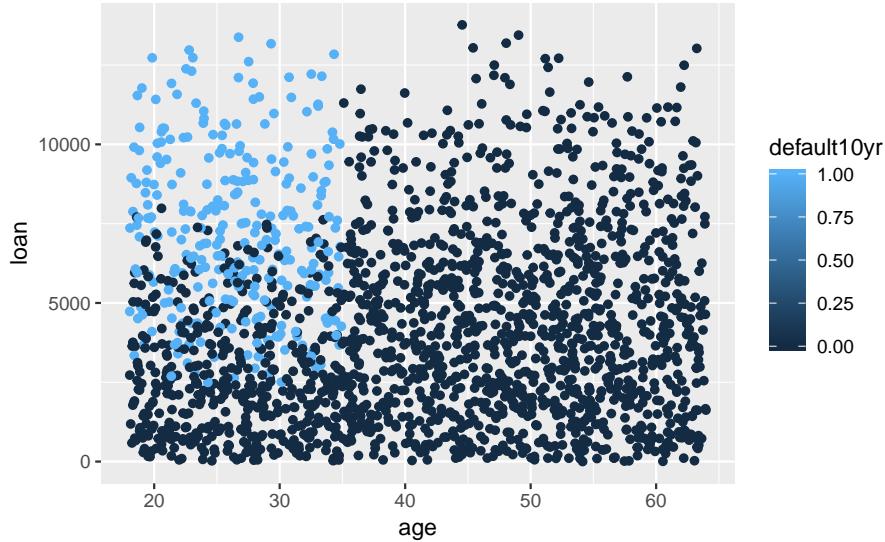
```
# what age and LTI is more likely to default
library(ggplot2)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures    rlang
#>   c.quosures    rlang
#>   print.quosures rlang

ggplot(dataset, aes(x = age, y = LTI, col = default10yr)) +
  geom_point()
```



```
# what age and loan size is more likely to default
library(ggplot2)

ggplot(dataset, aes(x = age, y = loan, col = default10yr)) +
  geom_point()
```



37.4 Objective

The dataset contains information on different clients who received a loan at least 10 years ago. The variables income (yearly), age, loan (size in euros) and LTI (the loan to yearly income ratio) are available. Our goal is to devise a model which predicts, based on the input variables LTI and age, whether or not a default will occur within 10 years.

37.5 Steps

The dataset will be split up in a subset used for training the neural network and another set used for testing. As the ordering of the dataset is completely random, we do not have to extract random rows and can just take the first x rows.

```
## extract a set to train the NN
trainset <- dataset[1:800, ]

## select the test set
testset <- dataset[801:2000, ]
```

37.5.1 Build the neural network

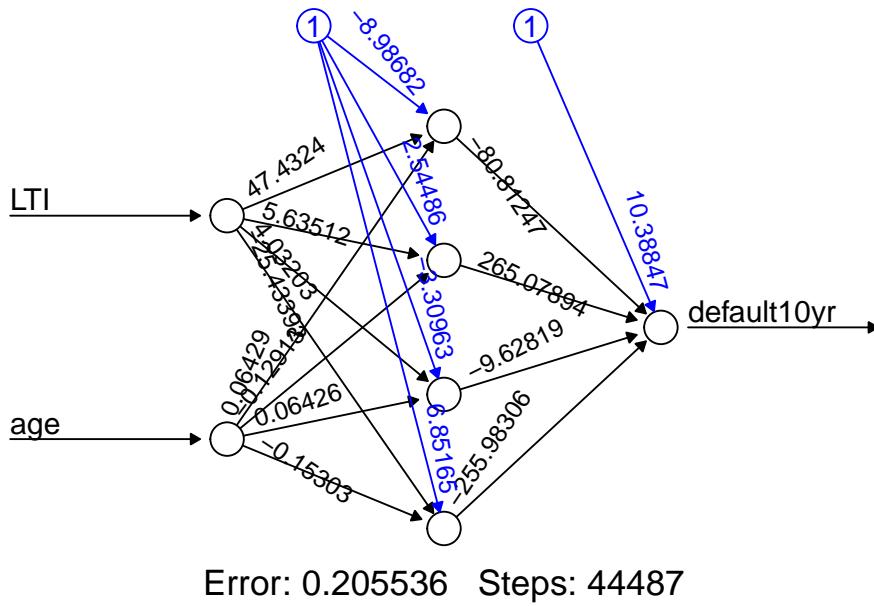
Now we'll build a neural network with 4 hidden nodes (a neural network is comprised of an input, hidden and output nodes). The number of nodes is chosen here without a clear method, however there are some rules of thumb. The `lifesign` option refers to the verbosity. The `output` is not linear and we will use a `threshold` value of 10%. The `neuralnet` package uses resilient backpropagation with weight backtracking as its standard algorithm.

```
## build the neural network (NN)
creditnet <- neuralnet(default10yr ~ LTI + age, trainset,
                        hidden = 4,
                        lifesign = "minimal",
                        linear.output = FALSE,
```

```
threshold = 0.1)
#> hidden: 4      thresh: 0.1      rep: 1/1      steps: 44487    error: 0.20554   time: 9.49 secs
```

The neuralnet package also has the possibility to visualize the generated model and show the found weights.

```
## plot the NN
plot(creditnet, rep = "best")
```



37.6 Test the neural network

Once we've trained the neural network we are ready to test it. We use the testset subset for this. The compute function is applied for computing the outputs based on the LTI and age inputs from the testset.

```
## test the resulting output
temp_test <- subset(testset, select = c("LTI", "age"))

creditnet.results <- compute(creditnet, temp_test)
```

The temp dataset contains only the columns LTI and age of the train set. Only these variables are used for input. The set looks as follows:

```
head(temp_test)
#>      LTI  age
#> 801 0.0231 25.9
#> 802 0.1373 40.8
#> 803 0.1046 32.5
#> 804 0.1599 53.2
#> 805 0.1116 46.5
#> 806 0.1149 47.1
```

Let's have a look at what the neural network produced:

```
results <- data.frame(actual = testset$default10yr, prediction = creditnet.results$net.result)
results[100:115, ]
#>     actual prediction
```

```
#> 900      0  7.29e-32
#> 901      0  8.17e-11
#> 902      0  4.33e-45
#> 903      1  1.00e+00
#> 904      0  8.06e-04
#> 905      0  3.54e-40
#> 906      0  1.48e-24
#> 907      1  1.00e+00
#> 908      0  1.11e-02
#> 909      0  8.05e-44
#> 910      0  6.72e-07
#> 911      1  1.00e+00
#> 912      0  9.97e-59
#> 913      1  1.00e+00
#> 914      0  3.39e-37
#> 915      0  1.18e-07
```

We can round to the nearest integer to improve readability:

```
results$prediction <- round(results$prediction)
results[100:115, ]
#>      actual prediction
#> 900      0          0
#> 901      0          0
#> 902      0          0
#> 903      1          1
#> 904      0          0
#> 905      0          0
#> 906      0          0
#> 907      1          1
#> 908      0          0
#> 909      0          0
#> 910      0          0
#> 911      1          1
#> 912      0          0
#> 913      1          1
#> 914      0          0
#> 915      0          0
```

As you can see it is pretty close! As already stated, this is a strongly simplified example. But it might serve as a basis for you to play around with your first neural network.

```
# how many predictions were wrong
indices <- which(results$actual != results$prediction)
indices
#> [1] 330 1008

# what are the predictions that failed
results[indices,]
#>      actual prediction
#> 1130      0          1
#> 1808      1          0
```

Chapter 38

Build a fully connected neural network from scratch

38.1 Introduction

<http://www.parallelr.com/r-deep-neural-network-from-scratch/>

```
library(neuralnet)

# Copyright 2016: www.ParallelR.com
# Parallel Blog : R For Deep Learning (I): Build Fully Connected Neural Network From Scratch
# Classification by 2-layers DNN and tested by iris dataset
# Author: Peng Zhao, patric.zhao@gmail.com

# Prediction
predict.dnn <- function(model, data = X.test) {
  # new data, transfer to matrix
  new.data <- data.matrix(data)

  # Feed Forwad
  hidden.layer <- sweep(new.data %*% model$W1 ,2, model$b1, '+')
  # neurons : Rectified Linear
  hidden.layer <- pmax(hidden.layer, 0)
  score <- sweep(hidden.layer %*% model$W2, 2, model$b2, '+')

  # Loss Function: softmax
  score.exp <- exp(score)
  probs <- sweep(score.exp, 1, rowSums(score.exp), '/')

  # select max possibility
  labels.predicted <- max.col(probs)
  return(labels.predicted)
}

# Train: build and train a 2-layers neural network
train.dnn <- function(x, y, traindata=data, testdata=NULL,
                      model = NULL,
                      # set hidden layers and neurons
```

```

# currently, only support 1 hidden layer
hidden=c(6),
# max iteration steps
maxit=2000,
# delta loss
abstol=1e-2,
# learning rate
lr = 1e-2,
# regularization rate
reg = 1e-3,
# show results every 'display' step
display = 100,
random.seed = 1)
{
# to make the case reproducible.
set.seed(random.seed)

# total number of training set
N <- nrow(traindata)

# extract the data and label
# don't need attribute
X <- unname(data.matrix(traindata[,x]))
# correct categories represented by integer
Y <- traindata[,y]
if(is.factor(Y)) { Y <- as.integer(Y) }
# create index for both row and col
# create index for both row and col
Y.len <- length(unique(Y))
Y.set <- sort(unique(Y))
Y.index <- cbind(1:N, match(Y, Y.set))

# create model or get model from parameter
if(is.null(model)) {
    # number of input features
    D <- ncol(X)
    # number of categories for classification
    K <- length(unique(Y))
    H <- hidden

    # create and init weights and bias
    W1 <- 0.01*matrix(rnorm(D*H), nrow=D, ncol=H)
    b1 <- matrix(0, nrow=1, ncol=H)

    W2 <- 0.01*matrix(rnorm(H*K), nrow=H, ncol=K)
    b2 <- matrix(0, nrow=1, ncol=K)
} else {
    D <- model$D
    K <- model$K
    H <- model$H
    W1 <- model$W1
    b1 <- model$b1
    W2 <- model$W2
}

```

```

    b2 <- model$b2
}

# use all train data to update weights since it's a small dataset
batchsize <- N
# init loss to a very big value
loss <- 100000

# Training the network
i <- 0
while(i < maxit && loss > abstol) {

  # iteration index
  i <- i +1

  # forward ....
  # 1 indicate row, 2 indicate col
  hidden.layer <- sweep(X %*% W1 ,2, b1, '+')
  # neurons : ReLU
  hidden.layer <- pmax(hidden.layer, 0)
  score <- sweep(hidden.layer %*% W2, 2, b2, '+')

  # softmax
  score.exp <- exp(score)
  # debug
  probs <- score.exp/rowSums(score.exp)

  # compute the loss
  corect.logprobs <- -log(probs[Y.index])
  data.loss <- sum(corect.logprobs)/batchsize
  reg.loss <- 0.5*reg* (sum(W1*W1) + sum(W2*W2))
  loss <- data.loss + reg.loss

  # display results and update model
  if( i %% display == 0) {
    if(!is.null(testdata)) {
      model <- list( D = D,
                     H = H,
                     K = K,
                     # weights and bias
                     W1 = W1,
                     b1 = b1,
                     W2 = W2,
                     b2 = b2)
      labs <- predict.dnn(model, testdata[,-y])
      accuracy <- mean(as.integer(testdata[,y]) == Y.set[labs])
      cat(i, loss, accuracy, "\n")
    } else {
      cat(i, loss, "\n")
    }
  }

  # backward ....
}

```

```

dscores <- probs
dscores[Y.index] <- dscores[Y.index] -1
dscores <- dscores / batchsize

dW2 <- t(hidden.layer) %*% dscores
db2 <- colSums(dscores)

dhidden <- dscores %*% t(W2)
dhidden[hidden.layer <= 0] <- 0

dW1 <- t(X) %*% dhidden
db1 <- colSums(dhidden)

# update ....
dW2 <- dW2 + reg*W2
dW1 <- dW1 + reg*W1

W1 <- W1 - lr * dW1
b1 <- b1 - lr * db1

W2 <- W2 - lr * dW2
b2 <- b2 - lr * db2

}

# final results
# creat list to store learned parameters
# you can add more parameters for debug and visualization
# such as residuals, fitted.values ...
model <- list( D = D,
                H = H,
                K = K,
                # weights and bias
                W1= W1,
                b1= b1,
                W2= W2,
                b2= b2)

return(model)
}

#####
# testing
#####
set.seed(1)

# 0. EDA
summary(iris)
#> Sepal.Length  Sepal.Width   Petal.Length  Petal.Width
#> Min.    :4.30  Min.    :2.00  Min.    :1.00  Min.    :0.1

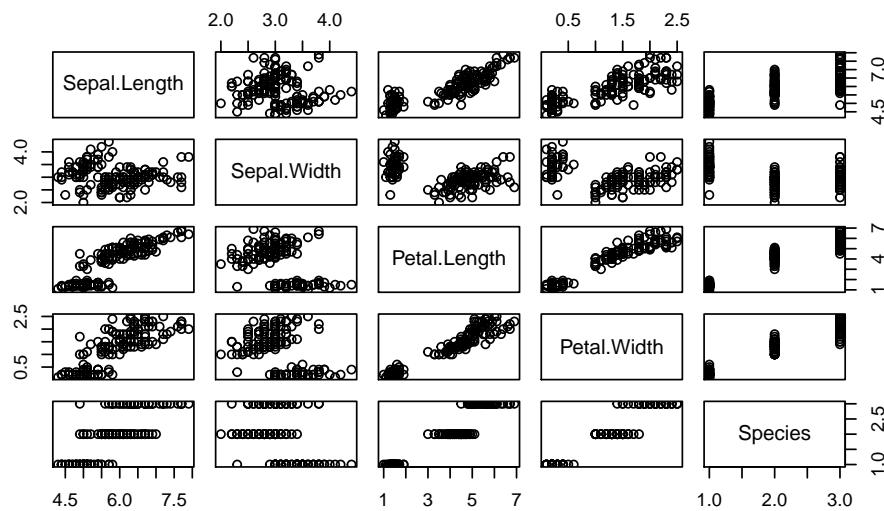
```

```
#> 1st Qu.:5.10    1st Qu.:2.80    1st Qu.:1.60    1st Qu.:0.3
#> Median :5.80    Median :3.00    Median :4.35    Median :1.3
#> Mean   :5.84    Mean   :3.06    Mean   :3.76    Mean   :1.2
#> 3rd Qu.:6.40    3rd Qu.:3.30    3rd Qu.:5.10    3rd Qu.:1.8
#> Max.   :7.90    Max.   :4.40    Max.   :6.90    Max.   :2.5
#>           Species
#> setosa      :50
#> versicolor:50
#> virginica  :50
#>
#>
#>
plot(iris)

# 1. split data into test/train
samp <- c(sample(1:50,25), sample(51:100,25), sample(101:150,25))

# 2. train model
ir.model <- train.dnn(x=1:4, y=5, traindata=iris[samp,],testdata=iris[-samp,], hidden=10, maxit=2000,
#> 50 1.1 0.333
#> 100 1.1 0.333
#> 150 1.09 0.333
#> 200 1.08 0.333
#> 250 1.05 0.333
#> 300 1 0.333
#> 350 0.933 0.667
#> 400 0.855 0.667
#> 450 0.775 0.667
#> 500 0.689 0.667
#> 550 0.611 0.68
#> 600 0.552 0.693
#> 650 0.507 0.747
#> 700 0.473 0.84
#> 750 0.445 0.88
#> 800 0.421 0.92
#> 850 0.399 0.947
#> 900 0.379 0.96
#> 950 0.36 0.96
#> 1000 0.341 0.973
#> 1050 0.324 0.973
#> 1100 0.307 0.973
#> 1150 0.292 0.973
#> 1200 0.277 0.973
#> 1250 0.263 0.973
#> 1300 0.25 0.973
#> 1350 0.238 0.973
#> 1400 0.227 0.973
#> 1450 0.216 0.973
#> 1500 0.207 0.973
#> 1550 0.198 0.973
#> 1600 0.19 0.973
#> 1650 0.183 0.973
#> 1700 0.176 0.973
```

```
#> 1750 0.17 0.973
#> 1800 0.164 0.973
#> 1850 0.158 0.973
#> 1900 0.153 0.973
#> 1950 0.149 0.973
#> 2000 0.144 0.973
# ir.model <- train.dnn(x=1:4, y=5, traindata=iris[samp,], hidden=6, maxit=2000, display=50)
```



```
# 3. prediction
# NOTE: if the predict is factor, we need to transfer the number into class manually.
#       To make the code clear, I don't write this change into predict.dnn function.
labels.dnn <- predict.dnn(ir.model, iris[-samp, -5])
```

```
# 4. verify the results
table(iris[-samp,5], labels.dnn)
#>           labels.dnn
#>             1 2 3
#>   setosa     25 0 0
#>   versicolor 0 23 2
#>   virginica   0 0 25
#>           labels.dnn
#>             1 2 3
#> #setosa     25 0 0
#> #versicolor 0 24 1
#> #virginica   0 0 25
```

```
#accuracy
mean(as.integer(iris[-samp, 5]) == labels.dnn)
#> [1] 0.973
# 0.98
```

```
# 5. compare with nnet
library(nnet)
ird <- data.frame(rbind(iris3[,1], iris3[,2], iris3[,3]),
                  species = factor(c(rep("s",50), rep("c", 50), rep("v", 50))))
ir.nn2 <- nnet(species ~ ., data = ird, subset = samp, size = 6, rang = 0.1,
                decay = 1e-2, maxit = 2000)
#> # weights:  51
```

```

#> initial value 82.293110
#> iter 10 value 29.196376
#> iter 20 value 5.446284
#> iter 30 value 4.782022
#> iter 40 value 4.379729
#> iter 50 value 4.188725
#> iter 60 value 4.120587
#> iter 70 value 4.091706
#> iter 80 value 4.086017
#> iter 90 value 4.081664
#> iter 100 value 4.074111
#> iter 110 value 4.072894
#> iter 120 value 4.069011
#> iter 130 value 4.067690
#> iter 140 value 4.067633
#> final value 4.067633
#> converged

labels.nnet <- predict(ir.nn2, ird[,-samp], type="class")
table(ird$species[,-samp], labels.nnet)
#> labels.nnet
#>      c   s   v
#>      c 23   0   2
#>      s   0 25   0
#>      v   0   0 25
# labels.nnet
#      c   s   v
#c 22   0   3
#s  0 25   0
#v  3   0 22

# accuracy
mean(ird$species[,-samp] == labels.nnet)
#> [1] 0.973
# 0.96

# Visualization
# the output from screen, copy and paste here.
data1 <- ("i loss accuracy
50 1.098421 0.3333333
100 1.098021 0.3333333
150 1.096843 0.3333333
200 1.093393 0.3333333
250 1.084069 0.3333333
300 1.063278 0.3333333
350 1.027273 0.3333333
400 0.9707605 0.64
450 0.8996356 0.6666667
500 0.8335469 0.6666667
550 0.7662386 0.6666667
600 0.6914156 0.6666667
650 0.6195753 0.68
")

```

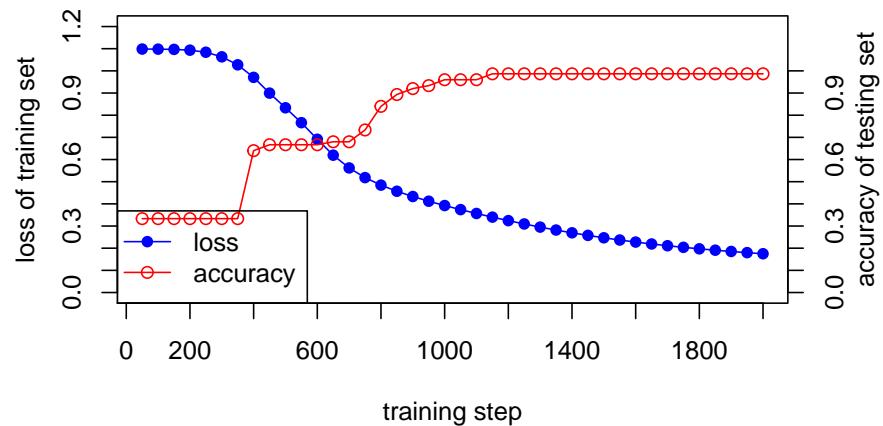
```

700 0.5620381 0.68
750 0.5184008 0.7333333
800 0.4844815 0.84
850 0.4568258 0.8933333
900 0.4331083 0.92
950 0.4118948 0.9333333
1000 0.392368 0.96
1050 0.3740457 0.96
1100 0.3566594 0.96
1150 0.3400993 0.9866667
1200 0.3243276 0.9866667
1250 0.3093422 0.9866667
1300 0.2951787 0.9866667
1350 0.2818472 0.9866667
1400 0.2693641 0.9866667
1450 0.2577245 0.9866667
1500 0.2469068 0.9866667
1550 0.2368819 0.9866667
1600 0.2276124 0.9866667
1650 0.2190535 0.9866667
1700 0.2111565 0.9866667
1750 0.2038719 0.9866667
1800 0.1971507 0.9866667
1850 0.1909452 0.9866667
1900 0.1852105 0.9866667
1950 0.1799045 0.9866667
2000 0.1749881 0.9866667 ")

data.v <- read.table(text=data1, header=T)
par(mar=c(5.1, 4.1, 4.1, 4.1))
plot(x=data.v$i, y=data.v$loss, type="o", col="blue", pch=16,
      main="IRIS loss and accuracy by 2-layers DNN",
      ylim=c(0, 1.2),
      xlab="",
      ylab="",
      axe =F)
lines(x=data.v$i, y=data.v$accuracy, type="o", col="red", pch=1)
box()
axis(1, at=seq(0,2000,by=200))
axis(4, at=seq(0,1.0,by=0.1))
axis(2, at=seq(0,1.2,by=0.1))
mtext("training step", 1, line=3)
mtext("loss of training set", 2, line=2.5)
mtext("accuracy of testing set", 4, line=2)

legend("bottomleft",
       legend = c("loss", "accuracy"),
       pch = c(16,1),
       col = c("blue","red"),
       lwd=c(1,1)
)

```

IRIS loss and accuracy by 2-layers DNN

Chapter 39

Wine with neuralnet

Source: <https://www.r-bloggers.com/multilabel-classification-with-neuralnet-package/>

The neuralnet package is perhaps not the best option in R for using neural networks. If you ask why, for starters it does not recognize the typical formula $y \sim .$, it does not support factors, it does not provide a lot of models other than a standard MLP, and it has great competitors in the nnet package that seems to be better integrated in R and can be used with the caret package, and in the MXnet package that is a high level deep learning library which provides a wide variety of neural networks.

But still, I think there is some value in the ease of use of the neuralnet package, especially for a beginner, therefore I'll be using it.

I'm going to be using both the neuralnet and, curiously enough, the nnet package. Let's load them:

```
# load libs
require(neuralnet)
#> Loading required package: neuralnet
require(nnet)
#> Loading required package: nnet
require(ggplot2)
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method          from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
set.seed(10)
```

39.1 The dataset

I looked in the UCI Machine Learning Repository¹ and found the wine dataset.

This dataset contains the results of a chemical analysis on 3 different kind of wines. The target variable is the label of the wine which is a factor with 3 (unordered) levels. The predictors are all continuous and represent 13 variables obtained as a result of chemical measurements.

```
# get the data file from the package location
wine_dataset_path <- file.path(data_raw_dir, "wine.data")
wine_dataset_path
#> [1] "/home/datascience/repos/machine-learning-rsuite/import/wine.data"
```

```

wines <- read.csv(wine_dataset_path)
wines
#>   X1 X14.23 X1.71 X2.43 X15.6 X127 X2.8 X3.06 X.28 X2.29 X5.64 X1.04
#> 1  1 13.2  1.78 2.14 11.2 100 2.65 2.76 0.26 1.28 4.38 1.050
#> 2  1 13.2  2.36 2.67 18.6 101 2.80 3.24 0.30 2.81 5.68 1.030
#> 3  1 14.4  1.95 2.50 16.8 113 3.85 3.49 0.24 2.18 7.80 0.860
#> 4  1 13.2  2.59 2.87 21.0 118 2.80 2.69 0.39 1.82 4.32 1.040
#> 5  1 14.2  1.76 2.45 15.2 112 3.27 3.39 0.34 1.97 6.75 1.050
#> 6  1 14.4  1.87 2.45 14.6  96 2.50 2.52 0.30 1.98 5.25 1.020
#> 7  1 14.1  2.15 2.61 17.6 121 2.60 2.51 0.31 1.25 5.05 1.060
#> 8  1 14.8  1.64 2.17 14.0  97 2.80 2.98 0.29 1.98 5.20 1.080
#> 9  1 13.9  1.35 2.27 16.0  98 2.98 3.15 0.22 1.85 7.22 1.010
#> 10 1 14.1  2.16 2.30 18.0 105 2.95 3.32 0.22 2.38 5.75 1.250
#> 11 1 14.1  1.48 2.32 16.8  95 2.20 2.43 0.26 1.57 5.00 1.170
#> 12 1 13.8  1.73 2.41 16.0  89 2.60 2.76 0.29 1.81 5.60 1.150
#> 13 1 14.8  1.73 2.39 11.4  91 3.10 3.69 0.43 2.81 5.40 1.250
#> 14 1 14.4  1.87 2.38 12.0 102 3.30 3.64 0.29 2.96 7.50 1.200
#> 15 1 13.6  1.81 2.70 17.2 112 2.85 2.91 0.30 1.46 7.30 1.280
#> 16 1 14.3  1.92 2.72 20.0 120 2.80 3.14 0.33 1.97 6.20 1.070
#> 17 1 13.8  1.57 2.62 20.0 115 2.95 3.40 0.40 1.72 6.60 1.130
#> 18 1 14.2  1.59 2.48 16.5 108 3.30 3.93 0.32 1.86 8.70 1.230
#> 19 1 13.6  3.10 2.56 15.2 116 2.70 3.03 0.17 1.66 5.10 0.960
#> 20 1 14.1  1.63 2.28 16.0 126 3.00 3.17 0.24 2.10 5.65 1.090
#> 21 1 12.9  3.80 2.65 18.6 102 2.41 2.41 0.25 1.98 4.50 1.030
#> 22 1 13.7  1.86 2.36 16.6 101 2.61 2.88 0.27 1.69 3.80 1.110
#> 23 1 12.8  1.60 2.52 17.8  95 2.48 2.37 0.26 1.46 3.93 1.090
#> 24 1 13.5  1.81 2.61 20.0  96 2.53 2.61 0.28 1.66 3.52 1.120
#> 25 1 13.1  2.05 3.22 25.0 124 2.63 2.68 0.47 1.92 3.58 1.130
#> 26 1 13.4  1.77 2.62 16.1  93 2.85 2.94 0.34 1.45 4.80 0.920
#> 27 1 13.3  1.72 2.14 17.0  94 2.40 2.19 0.27 1.35 3.95 1.020
#> 28 1 13.9  1.90 2.80 19.4 107 2.95 2.97 0.37 1.76 4.50 1.250
#> 29 1 14.0  1.68 2.21 16.0  96 2.65 2.33 0.26 1.98 4.70 1.040
#> 30 1 13.7  1.50 2.70 22.5 101 3.00 3.25 0.29 2.38 5.70 1.190
#> 31 1 13.6  1.66 2.36 19.1 106 2.86 3.19 0.22 1.95 6.90 1.090
#> 32 1 13.7  1.83 2.36 17.2 104 2.42 2.69 0.42 1.97 3.84 1.230
#> 33 1 13.8  1.53 2.70 19.5 132 2.95 2.74 0.50 1.35 5.40 1.250
#> 34 1 13.5  1.80 2.65 19.0 110 2.35 2.53 0.29 1.54 4.20 1.100
#> 35 1 13.5  1.81 2.41 20.5 100 2.70 2.98 0.26 1.86 5.10 1.040
#> 36 1 13.3  1.64 2.84 15.5 110 2.60 2.68 0.34 1.36 4.60 1.090
#> 37 1 13.1  1.65 2.55 18.0  98 2.45 2.43 0.29 1.44 4.25 1.120
#> 38 1 13.1  1.50 2.10 15.5  98 2.40 2.64 0.28 1.37 3.70 1.180
#> 39 1 14.2  3.99 2.51 13.2 128 3.00 3.04 0.20 2.08 5.10 0.890
#> 40 1 13.6  1.71 2.31 16.2 117 3.15 3.29 0.34 2.34 6.13 0.950
#> 41 1 13.4  3.84 2.12 18.8  90 2.45 2.68 0.27 1.48 4.28 0.910
#> 42 1 13.9  1.89 2.59 15.0 101 3.25 3.56 0.17 1.70 5.43 0.880
#> 43 1 13.2  3.98 2.29 17.5 103 2.64 2.63 0.32 1.66 4.36 0.820
#> 44 1 13.1  1.77 2.10 17.0 107 3.00 3.00 0.28 2.03 5.04 0.880
#> 45 1 14.2  4.04 2.44 18.9 111 2.85 2.65 0.30 1.25 5.24 0.870
#> 46 1 14.4  3.59 2.28 16.0 102 3.25 3.17 0.27 2.19 4.90 1.040
#> 47 1 13.9  1.68 2.12 16.0 101 3.10 3.39 0.21 2.14 6.10 0.910
#> 48 1 14.1  2.02 2.40 18.8 103 2.75 2.92 0.32 2.38 6.20 1.070
#> 49 1 13.9  1.73 2.27 17.4 108 2.88 3.54 0.32 2.08 8.90 1.120
#> 50 1 13.1  1.73 2.04 12.4  92 2.72 3.27 0.17 2.91 7.20 1.120

```

#> 51	1	13.8	1.65	2.60	17.2	94	2.45	2.99	0.22	2.29	5.60	1.240
#> 52	1	13.8	1.75	2.42	14.0	111	3.88	3.74	0.32	1.87	7.05	1.010
#> 53	1	13.8	1.90	2.68	17.1	115	3.00	2.79	0.39	1.68	6.30	1.130
#> 54	1	13.7	1.67	2.25	16.4	118	2.60	2.90	0.21	1.62	5.85	0.920
#> 55	1	13.6	1.73	2.46	20.5	116	2.96	2.78	0.20	2.45	6.25	0.980
#> 56	1	14.2	1.70	2.30	16.3	118	3.20	3.00	0.26	2.03	6.38	0.940
#> 57	1	13.3	1.97	2.68	16.8	102	3.00	3.23	0.31	1.66	6.00	1.070
#> 58	1	13.7	1.43	2.50	16.7	108	3.40	3.67	0.19	2.04	6.80	0.890
#> 59	2	12.4	0.94	1.36	10.6	88	1.98	0.57	0.28	0.42	1.95	1.050
#> 60	2	12.3	1.10	2.28	16.0	101	2.05	1.09	0.63	0.41	3.27	1.250
#> 61	2	12.6	1.36	2.02	16.8	100	2.02	1.41	0.53	0.62	5.75	0.980
#> 62	2	13.7	1.25	1.92	18.0	94	2.10	1.79	0.32	0.73	3.80	1.230
#> 63	2	12.4	1.13	2.16	19.0	87	3.50	3.10	0.19	1.87	4.45	1.220
#> 64	2	12.2	1.45	2.53	19.0	104	1.89	1.75	0.45	1.03	2.95	1.450
#> 65	2	12.4	1.21	2.56	18.1	98	2.42	2.65	0.37	2.08	4.60	1.190
#> 66	2	13.1	1.01	1.70	15.0	78	2.98	3.18	0.26	2.28	5.30	1.120
#> 67	2	12.4	1.17	1.92	19.6	78	2.11	2.00	0.27	1.04	4.68	1.120
#> 68	2	13.3	0.94	2.36	17.0	110	2.53	1.30	0.55	0.42	3.17	1.020
#> 69	2	12.2	1.19	1.75	16.8	151	1.85	1.28	0.14	2.50	2.85	1.280
#> 70	2	12.3	1.61	2.21	20.4	103	1.10	1.02	0.37	1.46	3.05	0.906
#> 71	2	13.9	1.51	2.67	25.0	86	2.95	2.86	0.21	1.87	3.38	1.360
#> 72	2	13.5	1.66	2.24	24.0	87	1.88	1.84	0.27	1.03	3.74	0.980
#> 73	2	13.0	1.67	2.60	30.0	139	3.30	2.89	0.21	1.96	3.35	1.310
#> 74	2	12.0	1.09	2.30	21.0	101	3.38	2.14	0.13	1.65	3.21	0.990
#> 75	2	11.7	1.88	1.92	16.0	97	1.61	1.57	0.34	1.15	3.80	1.230
#> 76	2	13.0	0.90	1.71	16.0	86	1.95	2.03	0.24	1.46	4.60	1.190
#> 77	2	11.8	2.89	2.23	18.0	112	1.72	1.32	0.43	0.95	2.65	0.960
#> 78	2	12.3	0.99	1.95	14.8	136	1.90	1.85	0.35	2.76	3.40	1.060
#> 79	2	12.7	3.87	2.40	23.0	101	2.83	2.55	0.43	1.95	2.57	1.190
#> 80	2	12.0	0.92	2.00	19.0	86	2.42	2.26	0.30	1.43	2.50	1.380
#> 81	2	12.7	1.81	2.20	18.8	86	2.20	2.53	0.26	1.77	3.90	1.160
#> 82	2	12.1	1.13	2.51	24.0	78	2.00	1.58	0.40	1.40	2.20	1.310
#> 83	2	13.1	3.86	2.32	22.5	85	1.65	1.59	0.61	1.62	4.80	0.840
#> 84	2	11.8	0.89	2.58	18.0	94	2.20	2.21	0.22	2.35	3.05	0.790
#> 85	2	12.7	0.98	2.24	18.0	99	2.20	1.94	0.30	1.46	2.62	1.230
#> 86	2	12.2	1.61	2.31	22.8	90	1.78	1.69	0.43	1.56	2.45	1.330
#> 87	2	11.7	1.67	2.62	26.0	88	1.92	1.61	0.40	1.34	2.60	1.360
#> 88	2	11.6	2.06	2.46	21.6	84	1.95	1.69	0.48	1.35	2.80	1.000
#> 89	2	12.1	1.33	2.30	23.6	70	2.20	1.59	0.42	1.38	1.74	1.070
#> 90	2	12.1	1.83	2.32	18.5	81	1.60	1.50	0.52	1.64	2.40	1.080
#> 91	2	12.0	1.51	2.42	22.0	86	1.45	1.25	0.50	1.63	3.60	1.050
#> 92	2	12.7	1.53	2.26	20.7	80	1.38	1.46	0.58	1.62	3.05	0.960
#> 93	2	12.3	2.83	2.22	18.0	88	2.45	2.25	0.25	1.99	2.15	1.150
#> 94	2	11.6	1.99	2.28	18.0	98	3.02	2.26	0.17	1.35	3.25	1.160
#> 95	2	12.5	1.52	2.20	19.0	162	2.50	2.27	0.32	3.28	2.60	1.160
#> 96	2	11.8	2.12	2.74	21.5	134	1.60	0.99	0.14	1.56	2.50	0.950
#> 97	2	12.3	1.41	1.98	16.0	85	2.55	2.50	0.29	1.77	2.90	1.230
#> 98	2	12.4	1.07	2.10	18.5	88	3.52	3.75	0.24	1.95	4.50	1.040
#> 99	2	12.3	3.17	2.21	18.0	88	2.85	2.99	0.45	2.81	2.30	1.420
#> 100	2	12.1	2.08	1.70	17.5	97	2.23	2.17	0.26	1.40	3.30	1.270
#> 101	2	12.6	1.34	1.90	18.5	88	1.45	1.36	0.29	1.35	2.45	1.040
#> 102	2	12.3	2.45	2.46	21.0	98	2.56	2.11	0.34	1.31	2.80	0.800
#> 103	2	11.8	1.72	1.88	19.5	86	2.50	1.64	0.37	1.42	2.06	0.940

#> 104	2	12.5	1.73	1.98	20.5	85	2.20	1.92	0.32	1.48	2.94	1.040
#> 105	2	12.4	2.55	2.27	22.0	90	1.68	1.84	0.66	1.42	2.70	0.860
#> 106	2	12.2	1.73	2.12	19.0	80	1.65	2.03	0.37	1.63	3.40	1.000
#> 107	2	12.7	1.75	2.28	22.5	84	1.38	1.76	0.48	1.63	3.30	0.880
#> 108	2	12.2	1.29	1.94	19.0	92	2.36	2.04	0.39	2.08	2.70	0.860
#> 109	2	11.6	1.35	2.70	20.0	94	2.74	2.92	0.29	2.49	2.65	0.960
#> 110	2	11.5	3.74	1.82	19.5	107	3.18	2.58	0.24	3.58	2.90	0.750
#> 111	2	12.5	2.43	2.17	21.0	88	2.55	2.27	0.26	1.22	2.00	0.900
#> 112	2	11.8	2.68	2.92	20.0	103	1.75	2.03	0.60	1.05	3.80	1.230
#> 113	2	11.4	0.74	2.50	21.0	88	2.48	2.01	0.42	1.44	3.08	1.100
#> 114	2	12.1	1.39	2.50	22.5	84	2.56	2.29	0.43	1.04	2.90	0.930
#> 115	2	11.0	1.51	2.20	21.5	85	2.46	2.17	0.52	2.01	1.90	1.710
#> 116	2	11.8	1.47	1.99	20.8	86	1.98	1.60	0.30	1.53	1.95	0.950
#> 117	2	12.4	1.61	2.19	22.5	108	2.00	2.09	0.34	1.61	2.06	1.060
#> 118	2	12.8	3.43	1.98	16.0	80	1.63	1.25	0.43	0.83	3.40	0.700
#> 119	2	12.0	3.43	2.00	19.0	87	2.00	1.64	0.37	1.87	1.28	0.930
#> 120	2	11.4	2.40	2.42	20.0	96	2.90	2.79	0.32	1.83	3.25	0.800
#> 121	2	11.6	2.05	3.23	28.5	119	3.18	5.08	0.47	1.87	6.00	0.930
#> 122	2	12.4	4.43	2.73	26.5	102	2.20	2.13	0.43	1.71	2.08	0.920
#> 123	2	13.1	5.80	2.13	21.5	86	2.62	2.65	0.30	2.01	2.60	0.730
#> 124	2	11.9	4.31	2.39	21.0	82	2.86	3.03	0.21	2.91	2.80	0.750
#> 125	2	12.1	2.16	2.17	21.0	85	2.60	2.65	0.37	1.35	2.76	0.860
#> 126	2	12.4	1.53	2.29	21.5	86	2.74	3.15	0.39	1.77	3.94	0.690
#> 127	2	11.8	2.13	2.78	28.5	92	2.13	2.24	0.58	1.76	3.00	0.970
#> 128	2	12.4	1.63	2.30	24.5	88	2.22	2.45	0.40	1.90	2.12	0.890
#> 129	2	12.0	4.30	2.38	22.0	80	2.10	1.75	0.42	1.35	2.60	0.790
#> 130	3	12.9	1.35	2.32	18.0	122	1.51	1.25	0.21	0.94	4.10	0.760
#> 131	3	12.9	2.99	2.40	20.0	104	1.30	1.22	0.24	0.83	5.40	0.740
#> 132	3	12.8	2.31	2.40	24.0	98	1.15	1.09	0.27	0.83	5.70	0.660
#> 133	3	12.7	3.55	2.36	21.5	106	1.70	1.20	0.17	0.84	5.00	0.780
#> 134	3	12.5	1.24	2.25	17.5	85	2.00	0.58	0.60	1.25	5.45	0.750
#> 135	3	12.6	2.46	2.20	18.5	94	1.62	0.66	0.63	0.94	7.10	0.730
#> 136	3	12.2	4.72	2.54	21.0	89	1.38	0.47	0.53	0.80	3.85	0.750
#> 137	3	12.5	5.51	2.64	25.0	96	1.79	0.60	0.63	1.10	5.00	0.820
#> 138	3	13.5	3.59	2.19	19.5	88	1.62	0.48	0.58	0.88	5.70	0.810
#> 139	3	12.8	2.96	2.61	24.0	101	2.32	0.60	0.53	0.81	4.92	0.890
#> 140	3	12.9	2.81	2.70	21.0	96	1.54	0.50	0.53	0.75	4.60	0.770
#> 141	3	13.4	2.56	2.35	20.0	89	1.40	0.50	0.37	0.64	5.60	0.700
#> 142	3	13.5	3.17	2.72	23.5	97	1.55	0.52	0.50	0.55	4.35	0.890
#> 143	3	13.6	4.95	2.35	20.0	92	2.00	0.80	0.47	1.02	4.40	0.910
#> 144	3	12.2	3.88	2.20	18.5	112	1.38	0.78	0.29	1.14	8.21	0.650
#> 145	3	13.2	3.57	2.15	21.0	102	1.50	0.55	0.43	1.30	4.00	0.600
#> 146	3	13.9	5.04	2.23	20.0	80	0.98	0.34	0.40	0.68	4.90	0.580
#> 147	3	12.9	4.61	2.48	21.5	86	1.70	0.65	0.47	0.86	7.65	0.540
#> 148	3	13.3	3.24	2.38	21.5	92	1.93	0.76	0.45	1.25	8.42	0.550
#> 149	3	13.1	3.90	2.36	21.5	113	1.41	1.39	0.34	1.14	9.40	0.570
#> 150	3	13.5	3.12	2.62	24.0	123	1.40	1.57	0.22	1.25	8.60	0.590
#> 151	3	12.8	2.67	2.48	22.0	112	1.48	1.36	0.24	1.26	10.80	0.480
#> 152	3	13.1	1.90	2.75	25.5	116	2.20	1.28	0.26	1.56	7.10	0.610
#> 153	3	13.2	3.30	2.28	18.5	98	1.80	0.83	0.61	1.87	10.52	0.560
#> 154	3	12.6	1.29	2.10	20.0	103	1.48	0.58	0.53	1.40	7.60	0.580
#> 155	3	13.2	5.19	2.32	22.0	93	1.74	0.63	0.61	1.55	7.90	0.600
#> 156	3	13.8	4.12	2.38	19.5	89	1.80	0.83	0.48	1.56	9.01	0.570

```

#> 157 3 12.4 3.03 2.64 27.0 97 1.90 0.58 0.63 1.14 7.50 0.670
#> 158 3 14.3 1.68 2.70 25.0 98 2.80 1.31 0.53 2.70 13.00 0.570
#> 159 3 13.5 1.67 2.64 22.5 89 2.60 1.10 0.52 2.29 11.75 0.570
#> 160 3 12.4 3.83 2.38 21.0 88 2.30 0.92 0.50 1.04 7.65 0.560
#> 161 3 13.7 3.26 2.54 20.0 107 1.83 0.56 0.50 0.80 5.88 0.960
#> 162 3 12.8 3.27 2.58 22.0 106 1.65 0.60 0.60 0.96 5.58 0.870
#> 163 3 13.0 3.45 2.35 18.5 106 1.39 0.70 0.40 0.94 5.28 0.680
#> 164 3 13.8 2.76 2.30 22.0 90 1.35 0.68 0.41 1.03 9.58 0.700
#> 165 3 13.7 4.36 2.26 22.5 88 1.28 0.47 0.52 1.15 6.62 0.780
#> 166 3 13.4 3.70 2.60 23.0 111 1.70 0.92 0.43 1.46 10.68 0.850
#> 167 3 12.8 3.37 2.30 19.5 88 1.48 0.66 0.40 0.97 10.26 0.720
#> 168 3 13.6 2.58 2.69 24.5 105 1.55 0.84 0.39 1.54 8.66 0.740
#> 169 3 13.4 4.60 2.86 25.0 112 1.98 0.96 0.27 1.11 8.50 0.670
#> 170 3 12.2 3.03 2.32 19.0 96 1.25 0.49 0.40 0.73 5.50 0.660
#> 171 3 12.8 2.39 2.28 19.5 86 1.39 0.51 0.48 0.64 9.90 0.570
#> 172 3 14.2 2.51 2.48 20.0 91 1.68 0.70 0.44 1.24 9.70 0.620
#> 173 3 13.7 5.65 2.45 20.5 95 1.68 0.61 0.52 1.06 7.70 0.640
#> 174 3 13.4 3.91 2.48 23.0 102 1.80 0.75 0.43 1.41 7.30 0.700
#> 175 3 13.3 4.28 2.26 20.0 120 1.59 0.69 0.43 1.35 10.20 0.590
#> 176 3 13.2 2.59 2.37 20.0 120 1.65 0.68 0.53 1.46 9.30 0.600
#> 177 3 14.1 4.10 2.74 24.5 96 2.05 0.76 0.56 1.35 9.20 0.610
#> X3.92 X1065
#> 1 3.40 1050
#> 2 3.17 1185
#> 3 3.45 1480
#> 4 2.93 735
#> 5 2.85 1450
#> 6 3.58 1290
#> 7 3.58 1295
#> 8 2.85 1045
#> 9 3.55 1045
#> 10 3.17 1510
#> 11 2.82 1280
#> 12 2.90 1320
#> 13 2.73 1150
#> 14 3.00 1547
#> 15 2.88 1310
#> 16 2.65 1280
#> 17 2.57 1130
#> 18 2.82 1680
#> 19 3.36 845
#> 20 3.71 780
#> 21 3.52 770
#> 22 4.00 1035
#> 23 3.63 1015
#> 24 3.82 845
#> 25 3.20 830
#> 26 3.22 1195
#> 27 2.77 1285
#> 28 3.40 915
#> 29 3.59 1035
#> 30 2.71 1285
#> 31 2.88 1515

```

```
#> 32  2.87  990
#> 33  3.00  1235
#> 34  2.87  1095
#> 35  3.47  920
#> 36  2.78  880
#> 37  2.51  1105
#> 38  2.69  1020
#> 39  3.53  760
#> 40  3.38  795
#> 41  3.00  1035
#> 42  3.56  1095
#> 43  3.00  680
#> 44  3.35  885
#> 45  3.33  1080
#> 46  3.44  1065
#> 47  3.33  985
#> 48  2.75  1060
#> 49  3.10  1260
#> 50  2.91  1150
#> 51  3.37  1265
#> 52  3.26  1190
#> 53  2.93  1375
#> 54  3.20  1060
#> 55  3.03  1120
#> 56  3.31  970
#> 57  2.84  1270
#> 58  2.87  1285
#> 59  1.82  520
#> 60  1.67  680
#> 61  1.59  450
#> 62  2.46  630
#> 63  2.87  420
#> 64  2.23  355
#> 65  2.30  678
#> 66  3.18  502
#> 67  3.48  510
#> 68  1.93  750
#> 69  3.07  718
#> 70  1.82  870
#> 71  3.16  410
#> 72  2.78  472
#> 73  3.50  985
#> 74  3.13  886
#> 75  2.14  428
#> 76  2.48  392
#> 77  2.52  500
#> 78  2.31  750
#> 79  3.13  463
#> 80  3.12  278
#> 81  3.14  714
#> 82  2.72  630
#> 83  2.01  515
#> 84  3.08  520
```

```
#> 85 3.16 450
#> 86 2.26 495
#> 87 3.21 562
#> 88 2.75 680
#> 89 3.21 625
#> 90 2.27 480
#> 91 2.65 450
#> 92 2.06 495
#> 93 3.30 290
#> 94 2.96 345
#> 95 2.63 937
#> 96 2.26 625
#> 97 2.74 428
#> 98 2.77 660
#> 99 2.83 406
#> 100 2.96 710
#> 101 2.77 562
#> 102 3.38 438
#> 103 2.44 415
#> 104 3.57 672
#> 105 3.30 315
#> 106 3.17 510
#> 107 2.42 488
#> 108 3.02 312
#> 109 3.26 680
#> 110 2.81 562
#> 111 2.78 325
#> 112 2.50 607
#> 113 2.31 434
#> 114 3.19 385
#> 115 2.87 407
#> 116 3.33 495
#> 117 2.96 345
#> 118 2.12 372
#> 119 3.05 564
#> 120 3.39 625
#> 121 3.69 465
#> 122 3.12 365
#> 123 3.10 380
#> 124 3.64 380
#> 125 3.28 378
#> 126 2.84 352
#> 127 2.44 466
#> 128 2.78 342
#> 129 2.57 580
#> 130 1.29 630
#> 131 1.42 530
#> 132 1.36 560
#> 133 1.29 600
#> 134 1.51 650
#> 135 1.58 695
#> 136 1.27 720
#> 137 1.69 515
```

```
#> 138 1.82 580
#> 139 2.15 590
#> 140 2.31 600
#> 141 2.47 780
#> 142 2.06 520
#> 143 2.05 550
#> 144 2.00 855
#> 145 1.68 830
#> 146 1.33 415
#> 147 1.86 625
#> 148 1.62 650
#> 149 1.33 550
#> 150 1.30 500
#> 151 1.47 480
#> 152 1.33 425
#> 153 1.51 675
#> 154 1.55 640
#> 155 1.48 725
#> 156 1.64 480
#> 157 1.73 880
#> 158 1.96 660
#> 159 1.78 620
#> 160 1.58 520
#> 161 1.82 680
#> 162 2.11 570
#> 163 1.75 675
#> 164 1.68 615
#> 165 1.75 520
#> 166 1.56 695
#> 167 1.75 685
#> 168 1.80 750
#> 169 1.92 630
#> 170 1.83 510
#> 171 1.63 470
#> 172 1.71 660
#> 173 1.74 740
#> 174 1.56 750
#> 175 1.56 835
#> 176 1.62 840
#> 177 1.60 560

names(wines) <- c("label",
                  "Alcohol",
                  "Malic_acid",
                  "Ash",
                  "Alcalinity_of_ash",
                  "Magnesium",
                  "Total_phenols",
                  "Flavanoids",
                  "Nonflavonoid_phenols",
                  "Proanthocyanins",
                  "Color_intensity",
                  "Hue",
                  "OD280_OD315_of_diluted_wines",
```

```

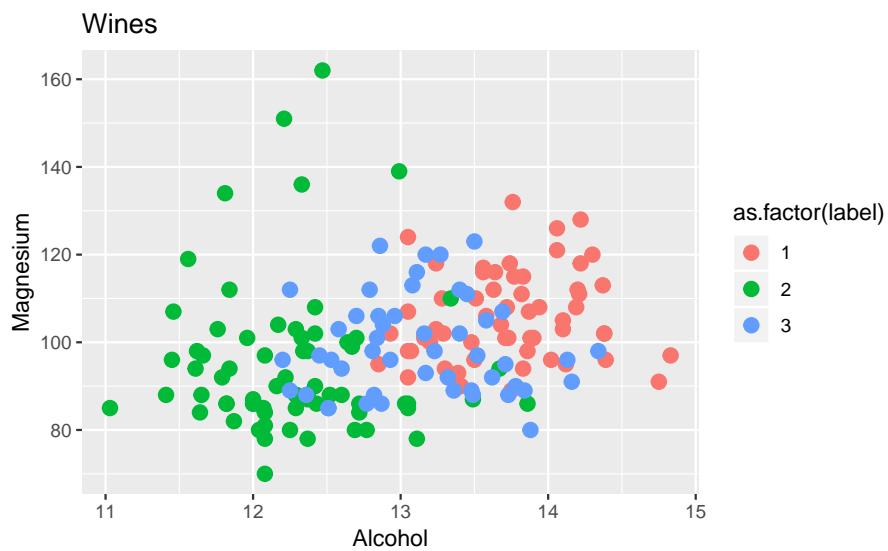
    "Proline")

head(wines)
#>   label Alcohol Malic_acid Ash Alcalinity_of_ash Magnesium Total_phenols
#> 1     1    13.2      1.78 2.14          11.2       100        2.65
#> 2     1    13.2      2.36 2.67          18.6       101        2.80
#> 3     1    14.4      1.95 2.50          16.8       113        3.85
#> 4     1    13.2      2.59 2.87          21.0       118        2.80
#> 5     1    14.2      1.76 2.45          15.2       112        3.27
#> 6     1    14.4      1.87 2.45          14.6        96        2.50
#>   Flavanoids Nonflavanoid_phenols Proanthocyanins Color_intensity Hue
#> 1     2.76                  0.26          1.28        4.38 1.05
#> 2     3.24                  0.30          2.81        5.68 1.03
#> 3     3.49                  0.24          2.18        7.80 0.86
#> 4     2.69                  0.39          1.82        4.32 1.04
#> 5     3.39                  0.34          1.97        6.75 1.05
#> 6     2.52                  0.30          1.98        5.25 1.02
#>   OD280_OD315_of_diluted_wines Proline
#> 1                   3.40      1050
#> 2                   3.17      1185
#> 3                   3.45      1480
#> 4                   2.93      735
#> 5                   2.85      1450
#> 6                   3.58      1290

plt1 <- ggplot(wines, aes(x = Alcohol, y = Magnesium, colour = as.factor(label))) +
  geom_point(size=3) +
  ggtitle("Wines")

plt1

```

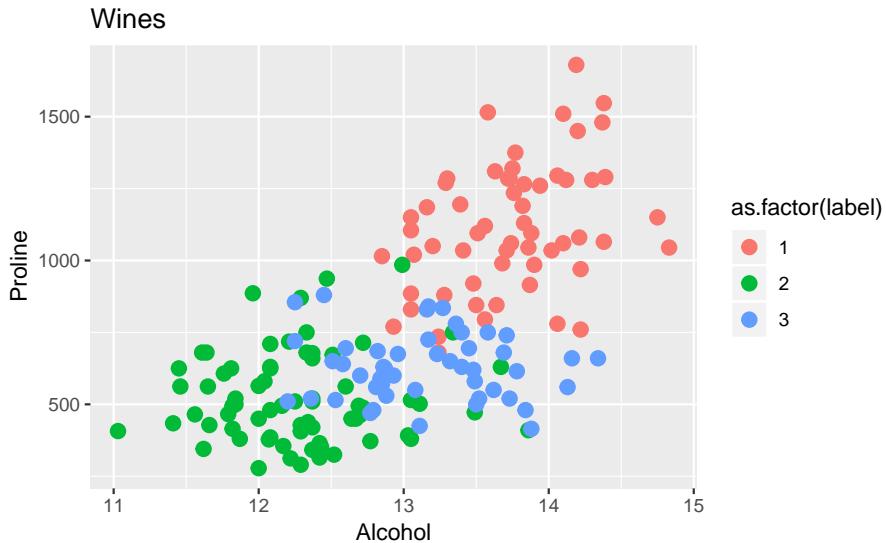


```

plt2 <- ggplot(wines, aes(x = Alcohol, y = Proline, colour = as.factor(label))) +
  geom_point(size=3) +
  ggtitle("Wines")

plt2

```



39.2 Preprocessing

During the preprocessing phase, I have to do at least the following two things:

Encode the categorical variables. Standardize the predictors. First of all, let's encode our target variable. The encoding of the categorical variables is needed when using neuralnet since it does not like factors at all. It will shout at you if you try to feed in a factor (I am told nnet likes factors though).

In the wine dataset the variable label contains three different labels: 1,2 and 3.

The usual practice, as far as I know, is to encode categorical variables as a “one hot” vector. For instance, if I had three classes, like in this case, I'd need to replace the label variable with three variables like these:

```
# 11,12,13
# 1,0,0
# 0,0,1
# ...
```

In this case the first observation would be labelled as a 1, the second would be labelled as a 2, and so on. Ironically, the `nnet` package provides a function to perform this encoding in a painless way:

```
# Encode as a one hot vector multilabel data
train <- cbind(wines[, 2:14], class.ind(as.factor(wines$label)))

# Set labels name
names(train) <- c(names(wines)[2:14], "l1", "l2", "l3")
```

By the way, since the predictors are all continuous, you do not need to encode any of them, however, in case you needed to, you could apply the same strategy applied above to all the categorical predictors. Unless of course you'd like to try some other kind of custom encoding.

Now let's standardize the predictors in the $[0-1] \rightarrow [0-1]$ interval by leveraging the `lapply` function:

```
# Scale data
scl <- function(x) { (x - min(x))/(max(x) - min(x)) }
train[, 1:13] <- data.frame(lapply(train[, 1:13], scl))
head(train)
#>   Alcohol Malic_acid   Ash Alcalinity_of_ash Magnesium Total_phenols
#> 1    0.571      0.206  0.417           0.0309     0.326        0.576
```

```
#> 2 0.561 0.320 0.701 0.4124 0.337 0.628
#> 3 0.879 0.239 0.610 0.3196 0.467 0.990
#> 4 0.582 0.366 0.807 0.5361 0.522 0.628
#> 5 0.834 0.202 0.583 0.2371 0.457 0.790
#> 6 0.884 0.223 0.583 0.2062 0.283 0.524
#>   Flavanoids Nonflavanoid_phenols Proanthocyanins Color_intensity Hue
#> 1 0.511 0.245 0.274 0.265 0.463
#> 2 0.612 0.321 0.757 0.375 0.447
#> 3 0.665 0.208 0.558 0.556 0.309
#> 4 0.496 0.491 0.445 0.259 0.455
#> 5 0.643 0.396 0.492 0.467 0.463
#> 6 0.460 0.321 0.495 0.339 0.439
#>   OD280_OD315_of_diluted_wines Proline l1 l2 l3
#> 1 0.780 0.551 1 0 0
#> 2 0.696 0.647 1 0 0
#> 3 0.799 0.857 1 0 0
#> 4 0.608 0.326 1 0 0
#> 5 0.579 0.836 1 0 0
#> 6 0.846 0.722 1 0 0
```

39.3 Fitting the model with neuralnet

Now it is finally time to fit the model.

As you might remember from the old post I wrote, `neuralnet` does not like the formula `y~..`. Fear not, you can build the formula to be used in a simple step:

```
# Set up formula
n <- names(train)
f <- as.formula(paste("l1 + l2 + l3 ~", paste(n[!n %in% c("l1", "l2", "l3")], collapse = " + ")))
f
#> l1 + l2 + l3 ~ Alcohol + Malic_acid + Ash + Alcalinity_of_ash +
#>     Magnesium + Total_phenols + Flavanoids + Nonflavanoid_phenols +
#>     Proanthocyanins + Color_intensity + Hue + OD280_OD315_of_diluted_wines +
#>     Proline
```

Note that the characters in the vector are not pasted to the right of the “`~`” symbol.

Just remember to check that the formula is indeed correct and then you are good to go.

Let's train the neural network with the full dataset. It should take very little time to converge. If you did not standardize the predictors it could take a lot more though.

```
nn <- neuralnet(f,
                  data = train,
                  hidden = c(13, 10, 3),
                  act.fct = "logistic",
                  linear.output = FALSE,
                  lifesign = "minimal")
#> hidden: 13, 10, 3    thresh: 0.01    rep: 1/1    steps:      88  error: 0.03039  time: 0.06 secs
```

Note that I set the argument `linear.output` to `FALSE` in order to tell the model that I want to apply the activation function `act.fct` and that I am not doing a regression task. Then I set the activation function to `logistic` (which by the way is the default option) in order to apply the logistic function. The other available option is `tanh` but the model seems to perform a little worse with it so I opted for the default option. As

far as I know these two are the only two available options, there is no “relu” function available although it seems to be a common activation function in other packages.

As far as the number of hidden neurons, I tried some combination and the one used seems to perform slightly better than the others (around 1% of accuracy difference in cross validation score).

By using the in-built plot method you can get a visual take on what is actually happening inside the model, however the plot is not that helpful I think

```
plot(nn)
```

Let's have a look at the accuracy on the training set:

```
# Compute predictions
pr.nn <- compute(nn, train[, 1:13])

# Extract results
pr.nn_ <- pr.nn$net.result
head(pr.nn_)
#>      [,1]     [,2]     [,3]
#> [1,] 0.990 0.00317 6.99e-06
#> [2,] 0.991 0.00233 8.69e-06
#> [3,] 0.991 0.00210 8.65e-06
#> [4,] 0.986 0.00442 8.74e-06
#> [5,] 0.992 0.00212 8.32e-06
#> [6,] 0.992 0.00214 8.34e-06

# Accuracy (training set)
original_values <- max.col(train[, 14:16])
pr.nn_2 <- max.col(pr.nn_)
mean(pr.nn_2 == original_values)
#> [1] 1
```

100% not bad! But wait, this may be because our model over fitted the data, furthermore evaluating accuracy on the training set is kind of cheating since the model already “knows” (or should know) the answers. In order to assess the “true accuracy” of the model you need to perform some kind of cross validation.

39.4 Cross validating the classifier

Let's crossvalidate the model using the evergreen 10 fold cross validation with the following train and test split: 95% of the dataset will be used as training set while the remaining 5% as test set.

Just out of curiosity I decided to run a LOOCV round too. In case you'd like to run this cross validation technique, just set the proportion variable to 0.995: this will select just one observation for as test set and leave all the other observations as training set. Running LOOCV you should get similar results to the 10 fold cross validation.

```
# Set seed for reproducibility purposes
set.seed(500)
# 10 fold cross validation
k <- 10
# Results from cv
outs <- NULL
# Train test split proportions
proportion <- 0.95 # Set to 0.995 for LOOCV
```

```

# Crossvalidate, go!
for(i in 1:k)
{
  index <- sample(1:nrow(train), round(proportion*nrow(train)))
  train_cv <- train[index, ]
  test_cv <- train[-index, ]
  nn_cv <- neuralnet(f,
    data = train_cv,
    hidden = c(13, 10, 3),
    act.fct = "logistic",
    linear.output = FALSE)

  # Compute predictions
  pr.nn <- compute(nn_cv, test_cv[, 1:13])
  # Extract results
  pr.nn_ <- pr.nn$net.result
  # Accuracy (test set)
  original_values <- max.col(test_cv[, 14:16])
  pr.nn_2 <- max.col(pr.nn_)
  outs[i] <- mean(pr.nn_2 == original_values)
}

mean(outs)
#> [1] 0.978

```

98.8%, awesome! Next time when you are invited to a relaxing evening that includes a wine tasting competition I think you should definitely bring your laptop as a contestant!

Aside from that poor taste joke, (I made it again!), indeed this dataset is not the most challenging, I think with some more tweaking a better cross validation score could be achieved. Nevertheless I hope you found this tutorial useful. A gist with the entire code for this tutorial can be found [here](#).

Thank you for reading this article, please feel free to leave a comment if you have any questions or suggestions and share the post with others if you find it useful.

Notes:

Chapter 40

Classification and Regression with H2O Deep Learning

40.1 Introduction

Source: <http://docs.h2o.ai/h2o-tutorials/latest-stable/tutorials/deeplearning/index.html>

Repo: <https://github.com/h2oai/h2o-tutorials>

This tutorial shows how a H2O Deep Learning model can be used to do supervised classification and regression. A great tutorial about Deep Learning is given by Quoc Le here and here. This tutorial covers usage of H2O from R. A python version of this tutorial will be available as well in a separate document. This file is available in plain R, R markdown and regular markdown formats, and the plots are available as PDF files. All documents are available on Github.

If run from plain R, execute R in the directory of this script. If run from RStudio, be sure to setwd() to the location of this script. `h2o.init()` starts H2O in R's current working directory. `h2o.importFile()` looks for files from the perspective of where H2O was started.

More examples and explanations can be found in our H2O Deep Learning booklet and on our H2O Github Repository. The PDF slide deck can be found on Github.

40.2 H2O R Package

Load the H2O R package:

Source: <http://docs.h2o.ai/h2o-tutorials/latest-stable/tutorials/deeplearning/index.html>

```
## R installation instructions are at http://h2o.ai/download
library(h2o)
#>
#> -----
#>
#> Your next step is to start H2O:
#>     > h2o.init()
#>
#> For H2O package documentation, ask for help:
#>     > ??h2o
#>
```

```
#> After starting H2O, you can use the Web UI at http://localhost:54321
#> For more information visit http://docs.h2o.ai
#>
#> -----
#>
#> Attaching package: 'h2o'
#> The following objects are masked from 'package:stats':
#>
#>     cor, sd, var
#> The following objects are masked from 'package:base':
#>
#>     %&, %*, %in%, ||, apply, as.factor, as.numeric, colnames,
#>     colnames<-, ifelse, is.character, is.factor, is.numeric, log,
#>     log10, log1p, log2, round, signif, trunc
```

40.3 Start H2O

Start up a 1-node H2O server on your local machine, and allow it to use all CPU cores and up to 2GB of memory:

```
h2o.init(nthreads=-1, max_mem_size="2G")
#>
#> H2O is not running yet, starting it now...
#>
#> Note: In case of errors look at the following log files:
#>       /tmp/RtmpaoH2ki/h2o_datascience_started_from_r.out
#>       /tmp/RtmpaoH2ki/h2o_datascience_started_from_r.err
#>
#>
#> Starting H2O JVM and connecting: . Connection successful!
#>
#> R is connected to the H2O cluster:
#>   H2O cluster uptime:      1 seconds 270 milliseconds
#>   H2O cluster timezone:    America/Chicago
#>   H2O data parsing timezone: UTC
#>   H2O cluster version:    3.22.1.1
#>   H2O cluster version age: 8 months and 23 days !!!
#>   H2O cluster name:       H2O_started_from_R_datascience_mwl453
#>   H2O cluster total nodes: 1
#>   H2O cluster total memory: 1.78 GB
#>   H2O cluster total cores: 8
#>   H2O cluster allowed cores: 8
#>   H2O cluster healthy:     TRUE
#>   H2O Connection ip:       localhost
#>   H2O Connection port:     54321
#>   H2O Connection proxy:    NA
#>   H2O Internal Security:  FALSE
#>   H2O API Extensions:    XGBoost, Algos, AutoML, Core V3, Core V4
#>   R Version:              R version 3.6.0 (2019-04-26)
#> Warning in h2o.clusterInfo():
#> Your H2O cluster version is too old (8 months and 23 days) !
#> Please download and install the latest version from http://h2o.ai/download/
```

```
h2o.removeAll() ## clean slate - just in case the cluster was already running
#> [1] 0
```

The `h2o.deeplearning` function fits H2O's Deep Learning models from within R. We can run the example from the man page using the example function, or run a longer demonstration from the `h2o` package using the `demo` function::

```
args(h2o.deeplearning)
#> function (x, y, training_frame, model_id = NULL, validation_frame = NULL,
#>   nfolds = 0, keep_cross_validation_models = TRUE, keep_cross_validation_predictions = FALSE,
#>   keep_cross_validation_fold_assignment = FALSE, fold_assignment = c("AUTO",
#>     "Random", "Modulo", "Stratified"), fold_column = NULL,
#>   ignore_const_cols = TRUE, score_each_iteration = FALSE, weights_column = NULL,
#>   offset_column = NULL, balance_classes = FALSE, class_sampling_factors = NULL,
#>   max_after_balance_size = 5, max_hit_ratio_k = 0, checkpoint = NULL,
#>   pretrained_autoencoder = NULL, overwrite_with_best_model = TRUE,
#>   use_all_factor_levels = TRUE, standardize = TRUE, activation = c("Tanh",
#>     "TanhWithDropout", "Rectifier", "RectifierWithDropout",
#>     "Maxout", "MaxoutWithDropout"), hidden = c(200, 200),
#>   epochs = 10, train_samples_per_iteration = -2, target_ratio_comm_to_comp = 0.05,
#>   seed = -1, adaptive_rate = TRUE, rho = 0.99, epsilon = 1e-08,
#>   rate = 0.005, rate_annealing = 1e-06, rate_decay = 1, momentum_start = 0,
#>   momentum_ramp = 1e+06, momentum_stable = 0, nesterov_accelerated_gradient = TRUE,
#>   input_dropout_ratio = 0, hidden_dropout_ratios = NULL, l1 = 0,
#>   l2 = 0, max_w2 = 3.4028235e+38, initial_weight_distribution = c("UniformAdaptive",
#>     "Uniform", "Normal"), initial_weight_scale = 1, initial_weights = NULL,
#>   initial_biases = NULL, loss = c("Automatic", "CrossEntropy",
#>     "Quadratic", "Huber", "Absolute", "Quantile"), distribution = c("AUTO",
#>     "bernelloulli", "multinomial", "gaussian", "poisson", "gamma",
#>     "tweedie", "laplace", "quantile", "huber"), quantile_alpha = 0.5,
#>   tweedie_power = 1.5, huber_alpha = 0.9, score_interval = 5,
#>   score_training_samples = 10000, score_validation_samples = 0,
#>   score_duty_cycle = 0.1, classification_stop = 0, regression_stop = 1e-06,
#>   stopping_rounds = 5, stopping_metric = c("AUTO", "deviance",
#>     "logloss", "MSE", "RMSE", "MAE", "RMSLE", "AUC", "lift_top_group",
#>     "misclassification", "mean_per_class_error", "custom",
#>     "custom_increasing"), stopping_tolerance = 0, max_runtime_secs = 0,
#>   score_validation_sampling = c("Uniform", "Stratified"), diagnostics = TRUE,
#>   fast_mode = TRUE, force_load_balance = TRUE, variable_importances = TRUE,
#>   replicate_training_data = TRUE, single_node_mode = FALSE,
#>   shuffle_training_data = FALSE, missing_values_handling = c("MeanImputation",
#>     "Skip"), quiet_mode = FALSE, autoencoder = FALSE, sparse = FALSE,
#>   col_major = FALSE, average_activation = 0, sparsity_beta = 0,
#>   max_categorical_features = 2147483647, reproducible = FALSE,
#>   export_weights_and_biases = FALSE, mini_batch_size = 1, categorical_encoding = c("AUTO",
#>     "Enum", "OneHotInternal", "OneHotExplicit", "Binary",
#>     "Eigen", "LabelEncoder", "SortByResponse", "EnumLimited"),
#>   elastic_averaging = FALSE, elastic_averaging_moving_rate = 0.9,
#>   elastic_averaging_regularization = 0.001, export_checkpoints_dir = NULL,
#>   verbose = FALSE)
#> NULL
if (interactive()) help(h2o.deeplearning)
example(h2o.deeplearning)
#>
```

```
#> h2.dpl> ## No test:
#> h2.dpl> ##D library(h2o)
#> h2.dpl> ##D h2o.init()
#> h2.dpl> ##D iris_hf <- as.h2o(iris)
#> h2.dpl> ##D iris_dl <- h2o.deeplearning(x = 1:4, y = 5, training_frame = iris_hf, seed=123456)
#> h2.dpl> ##D
#> h2.dpl> ##D # now make a prediction
#> h2.dpl> ##D predictions <- h2o.predict(iris_dl, iris_hf)
#> h2.dpl> ## End(No test)
#> h2.dpl>
#> h2.dpl>
#> h2.dpl>
if (interactive()) demo(h2o.deeplearning) #requires user interaction
```

While **H2O Deep Learning** has many parameters, it was designed to be just as easy to use as the other supervised training methods in H2O. Early stopping, automatic data standardization and handling of categorical variables and missing values and adaptive learning rates (per weight) reduce the amount of parameters the user has to specify. Often, it's just the number and sizes of hidden layers, the number of epochs and the activation function and maybe some regularization techniques.

40.4 Let's have some fun first: Decision Boundaries

We start with a small dataset representing red and black dots on a plane, arranged in the shape of two nested spirals. Then we task H2O's machine learning methods to separate the red and black dots, i.e., recognize each spiral as such by assigning each point in the plane to one of the two spirals.

We visualize the nature of H2O Deep Learning (DL), H2O's tree methods (GBM/DRF) and H2O's generalized linear modeling (GLM) by plotting the decision boundary between the red and black spirals:

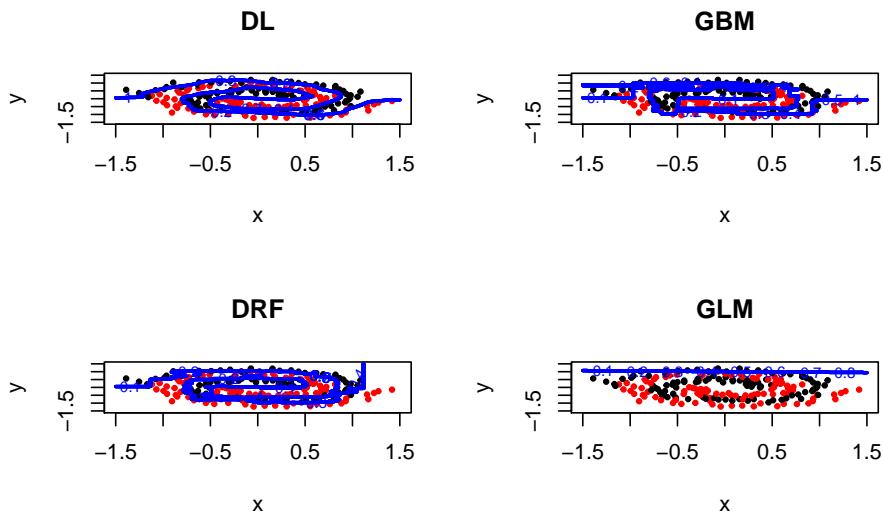
```
# setwd("~/h2o-tutorials/tutorials/deeplearning") ##For RStudio
spiral <- h2o.importFile(path = normalizePath(file.path(data_raw_dir, "spiral.csv")))
#>
|
|                                     | 0%
|
|=====| 100%
grid   <- h2o.importFile(path = normalizePath(file.path(data_raw_dir, "grid.csv")))
#>
|
|                                     | 0%
|
|=====| 16%
|
|=====| 100%

# Define helper to plot contours
plotC <- function(name, model, data=spiral, g=grid) {
  data <- as.data.frame(data) #get data from into R
  pred <- as.data.frame(h2o.predict(model, g))
  n=0.5*(sqrt(nrow(g))-1); d <- 1.5; h <- d*(-n:n)/n
  plot(data[,-3],pch=19,col=data[,3],cex=0.5,
    xlim=c(-d,d),ylim=c(-d,d),main=name)
  contour(h,h,z=array(ifelse(pred[,1]=="Red",0,1),
```

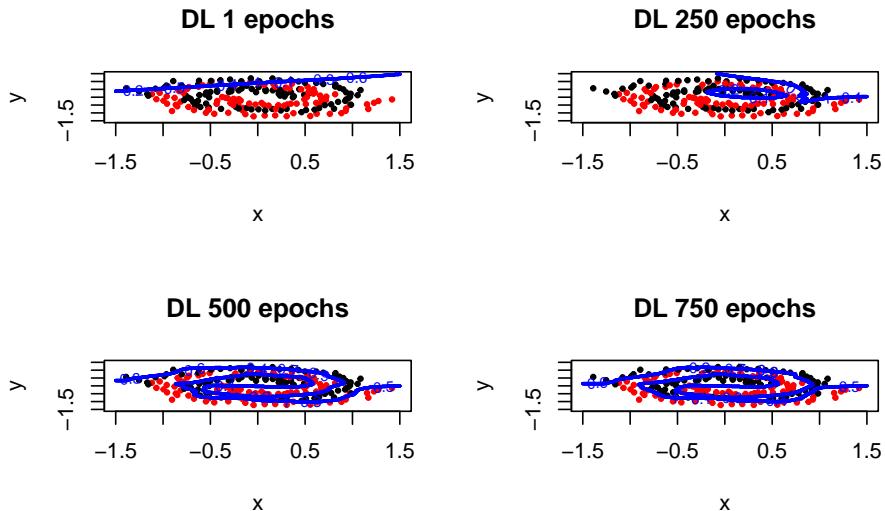
```
    dim=c(2*n+1,2*n+1)),col="blue",lwd=2,add=T)  
}
```

We build a few different models:

```
#dev.new(noRStudioGD=FALSE) #direct plotting output to a new window
par(mfrow=c(2,2)) #set up the canvas for 2x2 plots
plotC("DL", h2o.deeplearning(1:2,3,spiral,epochs=1e3))
plotC("GBM", h2o.gbm      (1:2,3,spiral))
plotC("DRF", h2o.randomForest(1:2,3,spiral))
plotC("GLM", h2o.glm      (1:2,3,spiral,family="binomial"))
```

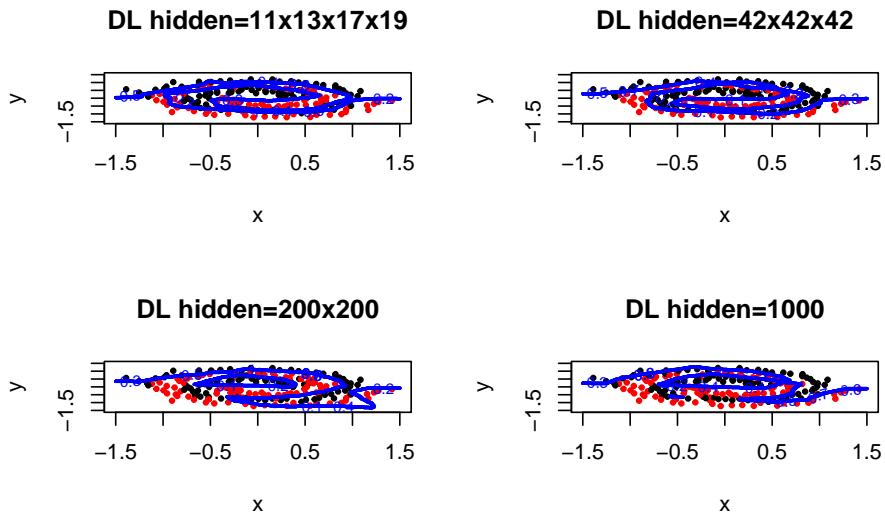


Let's investigate some more Deep Learning models. First, we explore the evolution over training time (number of passes over the data), and we use checkpointing to continue training the same model:



You can see how the network learns the structure of the spirals with enough training time. We explore different network architectures next:

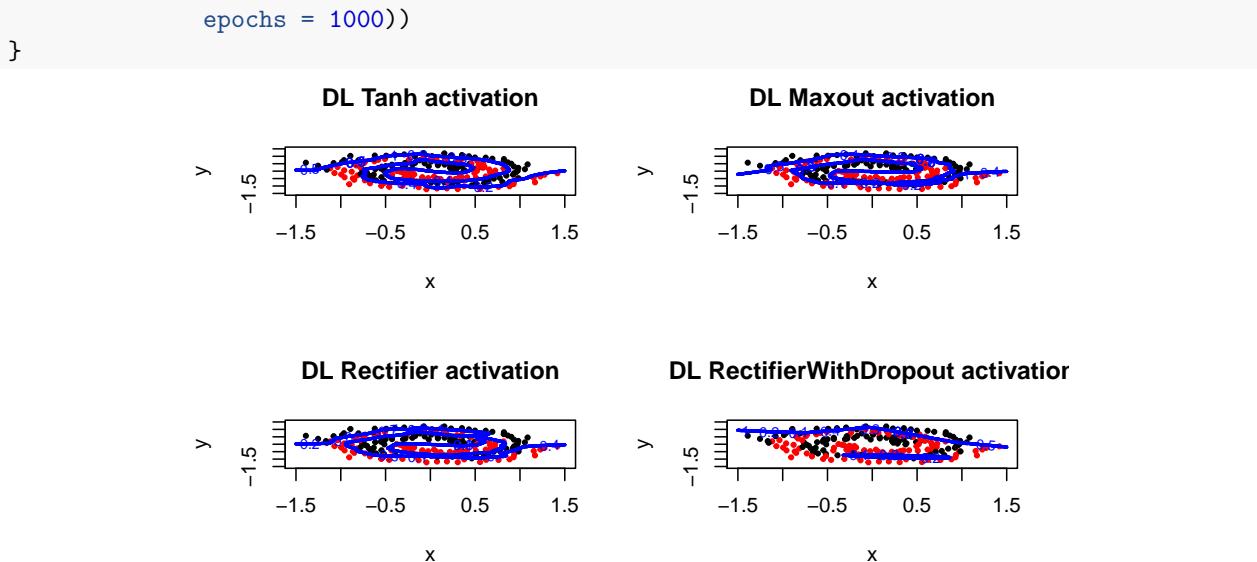
```
#dev.new(noRStudioGD=FALSE) #direct plotting output to a new window
par(mfrow=c(2,2)) #set up the canvas for 2x2 plots
for (hidden in list(c(11,13,17,19),c(42,42,42),c(200,200),c(1000))) {
  plotC(paste0("DL hidden=",paste0(hidden, collapse="x"))),
    h2o.deeplearning(1:2,3 ,spiral, hidden=hidden, epochs=500))
}
```



It is clear that different configurations can achieve similar performance, and that tuning will be required for optimal performance. Next, we compare between different activation functions, including one with 50% dropout regularization in the hidden layers:

```
#dev.new(noRStudioGD=FALSE) #direct plotting output to a new window
par(mfrow=c(2,2)) #set up the canvas for 2x2 plots

for (act in c("Tanh", "Maxout", "Rectifier", "RectifierWithDropout")) {
  plotC(paste0("DL ",act," activation")),
    h2o.deeplearning(1:2,3 ,spiral,
      activation = act,
      hidden = c(100,100),
```



Clearly, the dropout rate was too high or the number of epochs was too low for the last configuration, which often ends up performing the best on larger datasets where generalization is important.

More information about the parameters can be found in the H2O Deep Learning booklet.

40.5 Cover Type Dataset

We import the full cover type dataset (581k rows, 13 columns, 10 numerical, 3 categorical). We also split the data 3 ways: 60% for training, 20% for validation (hyper parameter tuning) and 20% for final testing.

```
df <- h2o.importFile(path = normalizePath(file.path(data_raw_dir, "covtype.full.csv")))
#>
#> [1] 581012      13
df
#>   Elevation Aspect Slope Horizontal_Distance_To_Hydrology
#> 1      3066    124     5                      0
#> 2      3136     32    20                     450
#> 3      2655     28    14                     42
#> 4      3191     45    19                    323
#> 5      3217     80    13                     30
#> 6      3119    293    13                     30
#>   Vertical_Distance_To_Hydrology Horizontal_Distance_To_Roadways
#> 1                      0                         1533
#> 2                     -38                        1290
#> 3                       8                        1890
#> 4                      88                        3932
#> 5                       1                        3901
#> 6                      10                        4810
```

```

#>   Hillshade_9am Hillshade_Noon Hillshade_3pm
#> 1      229        236       141
#> 2      211        193       111
#> 3      214        209       128
#> 4      221        195       100
#> 5      237        217       109
#> 6      182        237       194
#>   Horizontal_Distance_To_Fire_Points Wilderness_Area Soil_Type Cover_Type
#> 1                      459      area_0  type_22  class_1
#> 2                     1112     area_0  type_28  class_1
#> 3                     1001     area_2  type_9   class_2
#> 4                     2919     area_0  type_39  class_2
#> 5                     2859     area_0  type_22  class_7
#> 6                     1200     area_0  type_21  class_1
#>
#> [581012 rows x 13 columns]
splits <- h2o.splitFrame(df, c(0.6, 0.2), seed=1234)
train  <- h2o.assign(splits[[1]], "train.hex") # 60%
valid  <- h2o.assign(splits[[2]], "valid.hex") # 20%
test   <- h2o.assign(splits[[3]], "test.hex") # 20%

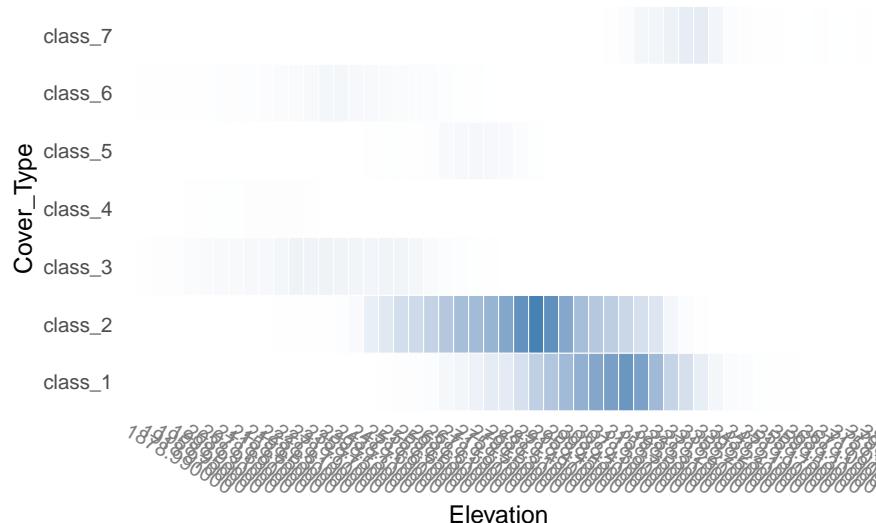
```

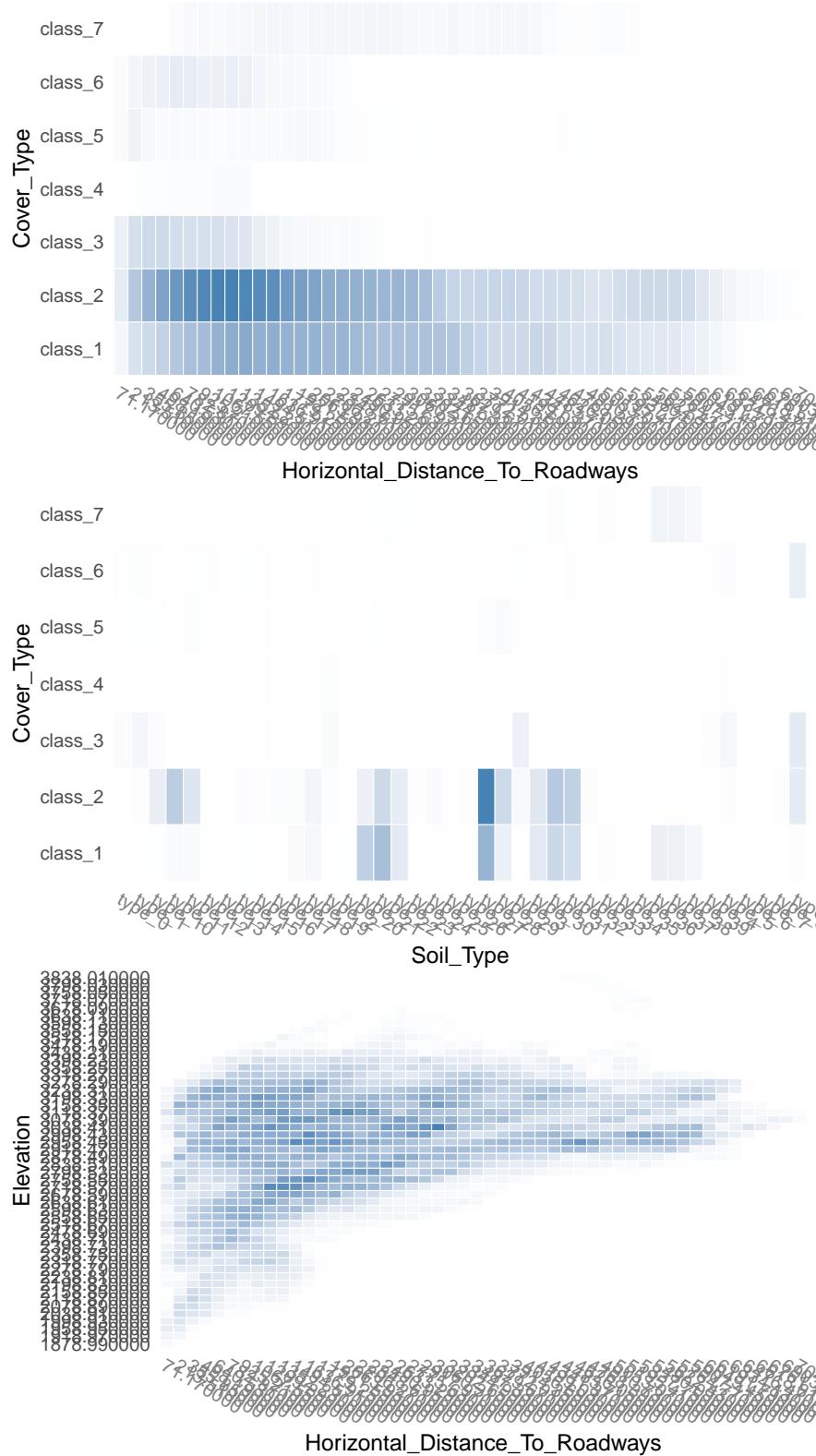
Here's a scalable way to do scatter plots via binning (works for categorical and numeric columns) to get more familiar with the dataset.

```

#dev.new(noRStudioGD=FALSE) #direct plotting output to a new window
par(mfrow=c(1,1)) # reset canvas
plot(h2o.tabulate(df, "Elevation",                                "Cover_Type"))
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures    rlang
#>   c.quosures    rlang
#>   print.quosures rlang
plot(h2o.tabulate(df, "Horizontal_Distance_To_Roadways", "Cover_Type"))
plot(h2o.tabulate(df, "Soil_Type",                           "Cover_Type"))
plot(h2o.tabulate(df, "Horizontal_Distance_To_Roadways", "Elevation" ))

```





40.5.1 First Run of H2O Deep Learning

Let's run our first Deep Learning model on the covtype dataset. We want to predict the `Cover_Type` column, a categorical feature with 7 levels, and the Deep Learning model will be tasked to perform (multi-class) classification. It uses the other 12 predictors of the dataset, of which 10 are numerical, and 2 are categorical with a total of 44 levels. We can expect the Deep Learning model to have 56 input neurons (after automatic one-hot encoding).

```
response <- "Cover_Type"
predictors <- setdiff(names(df), response)
predictors
#> [1] "Elevation"
#> [2] "Aspect"
#> [3] "Slope"
#> [4] "Horizontal_Distance_To_Hydrology"
#> [5] "Vertical_Distance_To_Hydrology"
#> [6] "Horizontal_Distance_To_Roadways"
#> [7] "Hillshade_9am"
#> [8] "Hillshade_Noon"
#> [9] "Hillshade_3pm"
#> [10] "Horizontal_Distance_To_Fire_Points"
#> [11] "Wilderness_Area"
#> [12] "Soil_Type"

train_df <- as.data.frame(train)
str(train_df)
#> 'data.frame': 349015 obs. of 13 variables:
#> $ Elevation : int 3136 3217 3119 2679 3261 2885 3227 2843 2853 2883 ...
#> $ Aspect   : int 32 80 293 48 322 26 32 12 124 177 ...
#> $ Slope    : int 20 13 13 7 13 9 6 18 12 9 ...
#> $ Horizontal_Distance_To_Hydrology : int 450 30 30 150 30 192 108 335 30 426 ...
#> $ Vertical_Distance_To_Hydrology  : int -38 1 10 24 5 38 13 50 -5 126 ...
#> $ Horizontal_Distance_To_Roadways : int 1290 3901 4810 1588 5701 3271 5542 2642 1485 2139 ...
#> $ Hillshade_9am      : int 211 237 182 223 186 216 219 199 240 225 ...
#> $ Hillshade_Noon    : int 193 217 237 224 226 220 227 201 231 246 ...
#> $ Hillshade_3pm     : int 111 109 194 136 180 140 145 135 119 153 ...
#> $ Horizontal_Distance_To_Fire_Points: int 1112 2859 1200 6265 769 2643 765 1719 2497 713 ...
#> $ Wilderness_Area   : Factor w/ 4 levels "area_0","area_1",...: 1 1 1 1 1 1 1 3 3 ...
#> $ Soil_Type          : Factor w/ 40 levels "type_0","type_1",...: 22 16 15 4 15 22 15 ...
#> $ Cover_Type         : Factor w/ 7 levels "class_1","class_2",...: 1 7 1 2 1 2 1 ...

valid_df <- as.data.frame(valid)
str(valid_df)
#> 'data.frame': 116018 obs. of 13 variables:
#> $ Elevation : int 3066 2655 2902 2994 2697 2990 3237 2884 2972 2696 ...
#> $ Aspect   : int 124 28 304 61 93 59 135 71 100 169 ...
#> $ Slope    : int 5 14 22 9 9 12 14 9 4 10 ...
#> $ Horizontal_Distance_To_Hydrology : int 0 42 511 391 306 108 240 459 175 323 ...
#> $ Vertical_Distance_To_Hydrology  : int 0 8 18 57 -2 10 -11 141 13 149 ...
#> $ Horizontal_Distance_To_Roadways : int 1533 1890 1273 4286 553 2190 1189 1214 5031 2452 ...
#> $ Hillshade_9am      : int 229 214 155 227 234 229 241 231 227 228 ...
#> $ Hillshade_Noon    : int 236 209 223 222 227 215 233 222 234 244 ...
#> $ Hillshade_3pm     : int 141 128 206 128 125 117 118 124 142 148 ...
#> $ Horizontal_Distance_To_Fire_Points: int 459 1001 1347 1928 1716 1048 2748 1355 6198 1044 ...
#> $ Wilderness_Area   : Factor w/ 4 levels "area_0","area_1",...: 1 3 3 1 1 3 1 3 1 3 ...
```

```
#> $ Soil_Type : Factor w/ 39 levels "type_0","type_1",...: 15 39 25 4 4 25 14 ...  
#> $ Cover_Type : Factor w/ 7 levels "class_1","class_2",...: 1 2 2 2 2 2 1 2 1 ...
```

To keep it fast, we only run for one epoch (one pass over the training data).

```
m1 <- h2o.deeplearning(  
  model_id="dl_model_first",  
  training_frame = train,  
  validation_frame = valid,    ## validation dataset: used for scoring and early stopping  
  x = predictors,  
  y = response,  
  activation="Rectifier",    ## default  
  #hidden=c(200,200),        ## default: 2 hidden layers with 200 neurons each  
  epochs = 1,  
  variable_importances=T    ## not enabled by default  
)  
#>  
|-----| 0%  
|-----| 10%  
|-----| 20%  
|-----| 30%  
|-----| 40%  
|-----| 50%  
|-----| 60%  
|-----| 70%  
|-----| 80%  
|-----| 90%  
|-----| 100%  
summary(m1)  
#> Model Details:  
#> ======  
#>  
#> H2OMultinomialModel: deeplearning  
#> Model Key: dl_model_first  
#> Status of Neuron Layers: predicting Cover_Type, 7-class classification, multinomial distribution, Cross-Entropy Loss  
#> layer units type dropout l1 l2 mean_rate rate_rms  
#> 1 1 56 Input 0.00 % NA NA NA NA  
#> 2 2 200 Rectifier 0.00 % 0.000000 0.000000 0.052280 0.214135  
#> 3 3 200 Rectifier 0.00 % 0.000000 0.000000 0.009215 0.008708  
#> 4 4 7 Softmax NA 0.000000 0.000000 0.117055 0.287708  
#> momentum mean_weight weight_rms mean_bias bias_rms  
#> 1 NA NA NA NA NA  
#> 2 0.000000 -0.011302 0.119766 0.022118 0.121956
```

```

#> 3 0.000000 -0.026712 0.116801 0.681338 0.397288
#> 4 0.000000 -0.373111 0.512719 -0.598332 0.174839
#>
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on training data. **
#> ** Metrics reported on temporary training frame with 9899 samples **
#>
#> Training Set Metrics:
#> =====
#>
#> MSE: (Extract with `h2o.mse`) 0.13
#> RMSE: (Extract with `h2o.rmse`) 0.361
#> Logloss: (Extract with `h2o.logloss`) 0.417
#> Mean Per-Class Error: 0.328
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, train = TRUE)`
#> =====
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>      class_1 class_2 class_3 class_4 class_5 class_6 class_7 Error
#> class_1    2761     827      2      0      8      0     74 0.2481
#> class_2     338    4425     44      0     20      9      4 0.0857
#> class_3      0      40    509      7      1      8      0 0.0991
#> class_4      0      0     14     26      0      0      0 0.3500
#> class_5      4      90      6      0     53      1      0 0.6558
#> class_6      0     69    152      3      1     73      0 0.7550
#> class_7     27      7      0      0      0      0     296 0.1030
#> Totals     3130    5458    727     36     83     91    374 0.1774
#>          Rate
#> class_1 = 911 / 3,672
#> class_2 = 415 / 4,840
#> class_3 = 56 / 565
#> class_4 = 14 / 40
#> class_5 = 101 / 154
#> class_6 = 225 / 298
#> class_7 = 34 / 330
#> Totals = 1,756 / 9,899
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, train = TRUE)`
#> =====
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1 0.822608
#> 2 2 0.984645
#> 3 3 0.998485
#> 4 4 0.999596
#> 5 5 1.000000
#> 6 6 1.000000
#> 7 7 1.000000
#>
#>
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on validation data. **
#> ** Metrics reported on full validation frame **
#>

```

```

#> Validation Set Metrics:
#> =====
#>
#> Extract validation frame with `h2o.getFrame("valid.hex")` -
#> MSE: (Extract with `h2o.mse`) 0.133
#> RMSE: (Extract with `h2o.rmse`) 0.365
#> Logloss: (Extract with `h2o.logloss`) 0.428
#> Mean Per-Class Error: 0.326
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, valid = TRUE)` -
#> =====
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>          class_1 class_2 class_3 class_4 class_5 class_6 class_7 Error
#> class_1    31963   9459     13      0     63     16    986 0.2479
#> class_2     4390   51058    499      8    244    112     69 0.0944
#> class_3       0     484   6459     55      8    137      0 0.0958
#> class_4       0       0    216    343      0      3      0 0.3897
#> class_5      53     998     59      0    754      6      0 0.5968
#> class_6       1     856   1678     44      4    881      0 0.7457
#> class_7     389      77      3      0      0      0    3630 0.1144
#> Totals     36796   62932   8927     450    1073    1155   4685 0.1804
#>          Rate
#> class_1 = 10,537 / 42,500
#> class_2 = 5,322 / 56,380
#> class_3 = 684 / 7,143
#> class_4 = 219 / 562
#> class_5 = 1,116 / 1,870
#> class_6 = 2,583 / 3,464
#> class_7 = 469 / 4,099
#> Totals = 20,930 / 116,018
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, valid = TRUE)` -
#> =====
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1 0.819597
#> 2 2 0.983882
#> 3 3 0.998052
#> 4 4 0.999578
#> 5 5 1.000000
#> 6 6 1.000000
#> 7 7 1.000000
#>
#>
#>
#> Scoring History:
#>           timestamp duration training_speed epochs iterations
#> 1 2019-09-20 15:12:01 0.000 sec             NA 0.00000         0
#> 2 2019-09-20 15:12:05 6.067 sec    7903 obs/sec 0.09944         1
#> 3 2019-09-20 15:12:21 22.168 sec   10943 obs/sec 0.59907         6
#> 4 2019-09-20 15:12:34 35.323 sec  12379 obs/sec 1.09698        11
#>           samples training_rmse training_logloss training_r2
#> 1      0.000000            NA            NA            NA

```

```

#> 2 34705.000000    0.45370    0.65859    0.89158
#> 3 209084.000000   0.38547    0.46547    0.92173
#> 4 382862.000000   0.36121    0.41746    0.93128
#>   training_classification_error validation_rmse validation_logloss
#> 1                      NA          NA          NA
#> 2                      0.27821    0.46368    0.68261
#> 3                      0.20295    0.39299    0.48243
#> 4                      0.17739    0.36527    0.42791
#>   validation_r2 validation_classification_error
#> 1                      NA          NA
#> 2                      0.88979    0.29197
#> 3                      0.92084    0.21145
#> 4                      0.93161    0.18040
#>
#> Variable Importances: (Extract with `h2o.varimp`)
#> =====
#>
#> Variable Importances:
#>   variable relative_importance scaled_importance
#> 1   Wilderness_Area.area_0      1.000000    1.000000
#> 2   Horizontal_Distance_To_Roadways 0.883062    0.883062
#> 3   Elevation                  0.873007    0.873007
#> 4   Horizontal_Distance_To_Fire_Points 0.852707    0.852707
#> 5   Wilderness_Area.area_2      0.766709    0.766709
#> percentage
#> 1  0.033708
#> 2  0.029766
#> 3  0.029427
#> 4  0.028743
#> 5  0.025844
#>
#> ---
#>   variable relative_importance scaled_importance
#> 51  Soil_Type.type_7        0.409154    0.409154
#> 52  Hillshade_3pm         0.401568    0.401568
#> 53  Slope                   0.395765    0.395765
#> 54  Aspect                  0.279821    0.279821
#> 55  Soil_Type.missing(NA)  0.000000    0.000000
#> 56  Wilderness_Area.missing(NA) 0.000000    0.000000
#> percentage
#> 51  0.013792
#> 52  0.013536
#> 53  0.013340
#> 54  0.009432
#> 55  0.000000
#> 56  0.000000

```

Inspect the model in Flow for more information about model building etc. by issuing a cell with the content `getModel "dl_model_first"`, and pressing Ctrl-Enter.

40.5.2 Variable Importances

Variable importances for Neural Network models are notoriously difficult to compute, and there are many pitfalls. H2O Deep Learning has implemented the method of Gedeon, and returns relative variable importances in descending order of importance.

```
head(as.data.frame(h2o.varimp(m1)))
#>   variable relative_importance scaled_importance
#> 1 Wilderness_Area.area_0          1.000          1.000
#> 2 Horizontal_Distance_To_Roadways 0.883          0.883
#> 3 Elevation                      0.873          0.873
#> 4 Horizontal_Distance_To_Fire_Points 0.853          0.853
#> 5 Wilderness_Area.area_2          0.767          0.767
#> 6 Wilderness_Area.area_3          0.706          0.706
#> percentage
#> 1 0.0337
#> 2 0.0298
#> 3 0.0294
#> 4 0.0287
#> 5 0.0258
#> 6 0.0238
```

40.5.3 Early Stopping

Now we run another, smaller network, and we let it stop automatically once the misclassification rate converges (specifically, if the moving average of length 2 does not improve by at least 1% for 2 consecutive scoring events). We also sample the validation set to 10,000 rows for faster scoring.

```
m2 <- h2o.deeplearning(
  model_id="dl_model_faster",
  training_frame=train,
  validation_frame=valid,
  x=predictors,
  y=response,
  hidden=c(32,32,32),           ## small network, runs faster
  epochs=1000000,               ## hopefully converges earlier...
  score_validation_samples=10000, ## sample the validation dataset (faster)
  stopping_rounds=2,
  stopping_metric="misclassification", ## could be "MSE", "logloss", "r2"
  stopping_tolerance=0.01
)
#>
| | 0%
| |
| |
|=====| 100%
summary(m2)
## Model Details:
## =====
## 
## H2OMultinomialModel: deeplearning
## Model Key: dl_model_faster
## Status of Neuron Layers: predicting Cover_Type, 7-class classification, multinomial distribution, Cr
```

```

#> 1   1   56   Input  0.00 %      NA      NA      NA      NA
#> 2   2   32 Rectifier  0.00 % 0.000000 0.000000 0.044995 0.204363
#> 3   3   32 Rectifier  0.00 % 0.000000 0.000000 0.000295 0.000192
#> 4   4   32 Rectifier  0.00 % 0.000000 0.000000 0.000841 0.001651
#> 5   5   7 Softmax      NA 0.000000 0.000000 0.089028 0.263416
#> momentum mean_weight weight_rms mean_bias bias_rms
#> 1      NA      NA      NA      NA      NA
#> 2 0.000000 -0.018824 0.317631 0.338853 0.373385
#> 3 0.000000 -0.064138 0.382940 0.669714 0.570015
#> 4 0.000000  0.017968 0.640646 0.647194 1.184354
#> 5 0.000000 -4.756408 3.588252 -3.596137 1.350429
#>
#> #> H2OMultinomialMetrics: deeplearning
#> ** Reported on training data. **
#> ** Metrics reported on temporary training frame with 9932 samples **
#>
#> Training Set Metrics:
#> =====
#>
#> MSE: (Extract with `h2o.mse`) 0.108
#> RMSE: (Extract with `h2o.rmse`) 0.329
#> Logloss: (Extract with `h2o.logloss`) 0.369
#> Mean Per-Class Error: 0.263
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, train = TRUE)`
#> =====
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>          class_1 class_2 class_3 class_4 class_5 class_6 class_7 Error
#> class_1    3007    490     1     0     5     3    28 0.1491
#> class_2     412    4397    33     0    33    31     2 0.1041
#> class_3      0     23    518     4     1    50     0 0.1309
#> class_4      0     0    11    40     0     8     0 0.3220
#> class_5      4     89     5     0    85     2     0 0.5405
#> class_6      1     31    56     4     1   185     0 0.3345
#> class_7     86     11     0     0     0     0   275 0.2608
#> Totals     3510    5041   624     48   125   279   305 0.1435
#>          Rate
#> class_1 = 527 / 3,534
#> class_2 = 511 / 4,908
#> class_3 = 78 / 596
#> class_4 = 19 / 59
#> class_5 = 100 / 185
#> class_6 = 93 / 278
#> class_7 = 97 / 372
#> Totals = 1,425 / 9,932
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, train = TRUE)`
#> =====
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1 0.856524
#> 2 2 0.984595
#> 3 3 0.998188
#> 4 4 0.999899

```

```

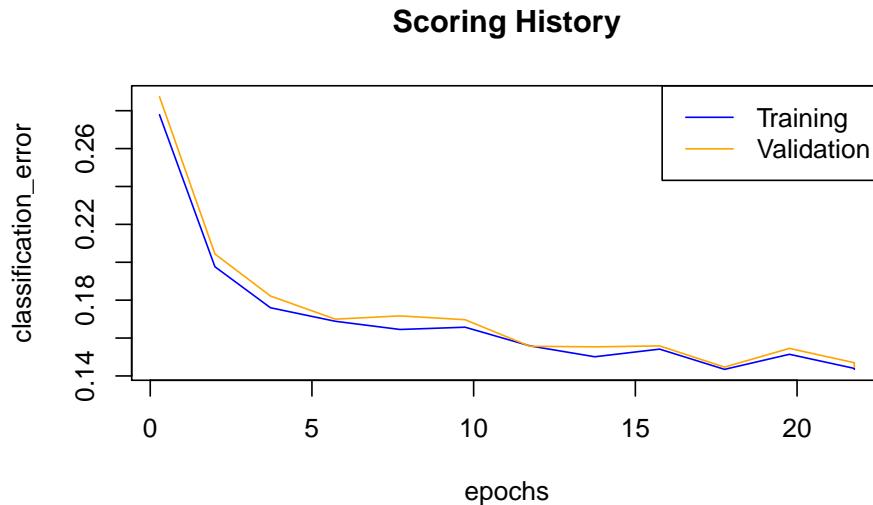
#> 5 5 1.000000
#> 6 6 1.000000
#> 7 7 1.000000
#>
#>
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on validation data. **
#> ** Metrics reported on temporary validation frame with 9913 samples **
#>
#> Validation Set Metrics:
#> =====
#>
#> MSE: (Extract with `h2o.mse`) 0.11
#> RMSE: (Extract with `h2o.rmse`) 0.332
#> Logloss: (Extract with `h2o.logloss`) 0.376
#> Mean Per-Class Error: 0.23
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, valid = TRUE)`
#> =====
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>   class_1 class_2 class_3 class_4 class_5 class_6 class_7 Error
#> class_1    3163    547     1     0     3     0    21 0.1531
#> class_2     390    4207    23     1    27    51     9 0.1064
#> class_3      0     27    521    16     0    55     0 0.1583
#> class_4      0      0     3     41     0     1     0 0.0889
#> class_5     10     62     2     0    65     4     0 0.5455
#> class_6      0     35     53     6     2   202     0 0.3221
#> class_7     80      5     0     0     0     0   280 0.2329
#> Totals     3643    4883    603    64    97   313   310 0.1447
#>           Rate
#> class_1 = 572 / 3,735
#> class_2 = 501 / 4,708
#> class_3 = 98 / 619
#> class_4 = 4 / 45
#> class_5 = 78 / 143
#> class_6 = 96 / 298
#> class_7 = 85 / 365
#> Totals = 1,434 / 9,913
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, valid = TRUE)`
#> =====
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1 0.855341
#> 2 2 0.985373
#> 3 3 0.998083
#> 4 4 0.999899
#> 5 5 1.000000
#> 6 6 1.000000
#> 7 7 1.000000
#>
#>
#>
#>
```

```
#> Scoring History:
#>   timestamp      duration training_speed epochs
#> 1 2019-09-20 15:12:37 0.000 sec        NA 0.00000
#> 2 2019-09-20 15:12:38 1.133 sec 96270 obs/sec 0.28604
#> 3 2019-09-20 15:12:43 6.693 sec 106412 obs/sec 2.00010
#> 4 2019-09-20 15:12:48 11.965 sec 110052 obs/sec 3.71670
#> 5 2019-09-20 15:12:54 17.649 sec 114508 obs/sec 5.72221
#> 6 2019-09-20 15:13:00 23.318 sec 116879 obs/sec 7.72611
#> 7 2019-09-20 15:13:05 28.700 sec 119550 obs/sec 9.73526
#> 8 2019-09-20 15:13:11 34.181 sec 121006 obs/sec 11.74058
#> 9 2019-09-20 15:13:16 39.757 sec 121766 obs/sec 13.74575
#> 10 2019-09-20 15:13:22 45.124 sec 122907 obs/sec 15.75115
#> 11 2019-09-20 15:13:27 50.497 sec 123792 obs/sec 17.75799
#> 12 2019-09-20 15:13:32 55.882 sec 124486 obs/sec 19.76427
#> 13 2019-09-20 15:13:38 1 min 1.196 sec 125201 obs/sec 21.77015
#> 14 2019-09-20 15:13:38 1 min 1.235 sec 125197 obs/sec 21.77015
#> iterations      samples training_rmse training_logloss training_r2
#> 1          0 0.000000        NA        NA        NA
#> 2          1 99833.000000 0.45275 0.62097 0.89680
#> 3          7 698066.000000 0.38388 0.47987 0.92581
#> 4         13 1297185.000000 0.36290 0.42781 0.93370
#> 5         20 1997138.000000 0.35440 0.41127 0.93677
#> 6         27 2696527.000000 0.35010 0.40644 0.93830
#> 7         34 3397751.000000 0.35273 0.40268 0.93736
#> 8         41 4097640.000000 0.34165 0.39531 0.94124
#> 9         48 4797472.000000 0.33552 0.38066 0.94333
#> 10        55 5497388.000000 0.34058 0.40336 0.94160
#> 11        62 6197805.000000 0.32917 0.36920 0.94545
#> 12        69 6898026.000000 0.33564 0.38261 0.94329
#> 13        76 7598110.000000 0.33031 0.37585 0.94507
#> 14        76 7598110.000000 0.32917 0.36920 0.94545
#> training_classification_error validation_rmse validation_logloss
#> 1                  NA        NA        NA
#> 2 0.277789 0.45999 0.63511
#> 3 0.19764 0.38998 0.48931
#> 4 0.17600 0.36883 0.43908
#> 5 0.16885 0.35687 0.41543
#> 6 0.16452 0.35616 0.41901
#> 7 0.16573 0.35551 0.40939
#> 8 0.15586 0.34039 0.38953
#> 9 0.15012 0.33897 0.38881
#> 10 0.15415 0.34215 0.40425
#> 11 0.14348 0.33233 0.37600
#> 12 0.15143 0.33941 0.39483
#> 13 0.14398 0.33497 0.38471
#> 14 0.14348 0.33233 0.37600
#> validation_r2 validation_classification_error
#> 1                  NA        NA
#> 2 0.89351 0.28740
#> 3 0.92346 0.20438
#> 4 0.93153 0.18219
#> 5 0.93590 0.16998
#> 6 0.93616 0.17169
```

```

#> 7      0.93639      0.16968
#> 8      0.94169      0.15565
#> 9      0.94217      0.15535
#> 10     0.94108      0.15586
#> 11     0.94441      0.14466
#> 12     0.94202      0.15454
#> 13     0.94353      0.14698
#> 14     0.94441      0.14466
#>
#> Variable Importances: (Extract with `h2o.varimp`)
#> =====
#>
#> Variable Importances:
#>   variable relative_importance scaled_importance
#> 1   Wilderness_Area.area_3      1.000000  1.000000
#> 2   Elevation                  0.966004  0.966004
#> 3   Horizontal_Distance_To_Roadways 0.965409  0.965409
#> 4   Wilderness_Area.area_1      0.915726  0.915726
#> 5   Wilderness_Area.area_2      0.845369  0.845369
#>   percentage
#> 1  0.032479
#> 2  0.031374
#> 3  0.031355
#> 4  0.029741
#> 5  0.027456
#>
#> ---
#>   variable relative_importance scaled_importance
#> 51  Soil_Type.type_14        0.338649  0.338649
#> 52  Vertical_Distance_To_Hydrology 0.251830  0.251830
#> 53  Slope                    0.226702  0.226702
#> 54  Aspect                   0.045678  0.045678
#> 55  Soil_Type.missing(NA)    0.000000  0.000000
#> 56  Wilderness_Area.missing(NA) 0.000000  0.000000
#>   percentage
#> 51  0.010999
#> 52  0.008179
#> 53  0.007363
#> 54  0.001484
#> 55  0.000000
#> 56  0.000000
plot(m2)

```



40.5.4 Adaptive Learning Rate

By default, H2O Deep Learning uses an adaptive learning rate (ADADELTA) for its stochastic gradient descent optimization. There are only two tuning parameters for this method: rho and epsilon, which balance the global and local search efficiencies. rho is the similarity to prior weight updates (similar to momentum), and epsilon is a parameter that prevents the optimization to get stuck in local optima. Defaults are rho=0.99 and epsilon=1e-8. For cases where convergence speed is very important, it might make sense to perform a few runs to optimize these two parameters (e.g., with rho in c(0.9,0.95,0.99,0.999) and epsilon in c(1e-10,1e-8,1e-6,1e-4)). Of course, as always with grid searches, caution has to be applied when extrapolating grid search results to a different parameter regime (e.g., for more epochs or different layer topologies or activation functions, etc.).

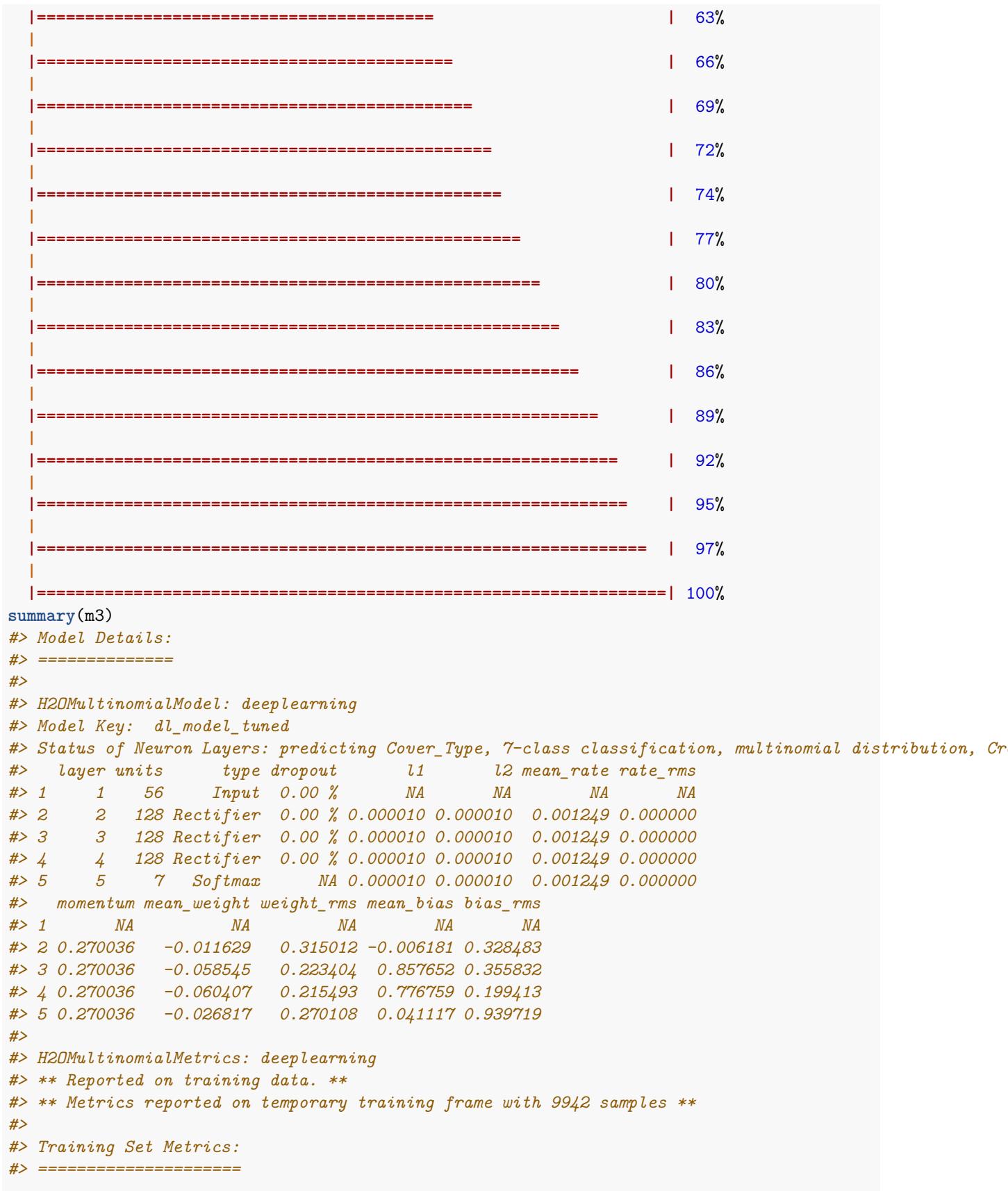
If adaptive_rate is disabled, several manual learning rate parameters become important: rate, rate_annealing, rate_decay, momentum_start, momentum_ramp, momentum_stable and nesterov_accelerated_gradient, the discussion of which we leave to H2O Deep Learning booklet.

40.5.5 Tuning

With some tuning, it is possible to obtain less than 10% test set error rate in about one minute. Error rates of below 5% are possible with larger models. Note that deep tree methods can be more effective for this dataset than Deep Learning, as they directly partition the space into sectors, which seems to be needed here.

```
m3 <- h2o.deeplearning(
  model_id="dl_model_tuned",
  training_frame=train,
  validation_frame=valid,
  x=predictors,
  y=response,
  overwrite_with_best_model=F,      ## Return final model after 10 epochs, even if not the best
  hidden=c(128,128,128),           ## more hidden layers -> more complex interactions
  epochs=10,                      ## to keep it short enough
  score_validation_samples=10000,    ## downsample validation set for faster scoring
  score_duty_cycle=0.025,          ## don't score more than 2.5% of the wall time
  adaptive_rate=F,                ## manually tuned learning rate
  rate=0.01,
  rate_annealing=2e-6,
```

```
momentum_start=0.2, ## manually tuned momentum
momentum_stable=0.4,
momentum_ramp=1e7,
l1=1e-5, ## add some L1/L2 regularization
l2=1e-5,
max_w2=10 ## helps stability for Rectifier
)
#>
| |
| == | 0%
| === | 3%
| ====
| ===- | 6%
| ===-- | 9%
| ===--- | 11%
| ===== | 14%
| ====== | 17%
| =====-- | 20%
| =====--- | 23%
| =====---= | 26%
| =====---== | 29%
| =====---== = | 32%
| =====---== == | 34%
| =====---== == = | 37%
| =====---== == == | 40%
| =====---== == == = | 43%
| =====---== == == == | 46%
| =====---== == == == = | 49%
| =====---== == == == == | 52%
| =====---== == == == == = | 54%
| =====---== == == == == == | 57%
| =====---== == == == == == = | 60%
```



```

#>
#> MSE: (Extract with `h2o.mse`) 0.0542
#> RMSE: (Extract with `h2o.rmse`) 0.233
#> Logloss: (Extract with `h2o.logloss`) 0.178
#> Mean Per-Class Error: 0.109
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, train = TRUE)`
#> =====
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>           class_1 class_2 class_3 class_4 class_5 class_6 class_7 Error
#> class_1      3354     329      0      0      4      0     20 0.0952
#> class_2       152    4547     12      1     25      6      0 0.0413
#> class_3        0     14    564     11      0     31      0 0.0903
#> class_4        0      0      5     45      0      0      0 0.1000
#> class_5        3     23      2      0    140      0      0 0.1667
#> class_6        2     19     37      3      0    218      0 0.2186
#> class_7       18      2      0      0      0      0    355 0.0533
#> Totals       3529    4934     620     60    169    255    375 0.0723
#>           Rate
#> class_1 = 353 / 3,707
#> class_2 = 196 / 4,743
#> class_3 = 56 / 620
#> class_4 = 5 / 50
#> class_5 = 28 / 168
#> class_6 = 61 / 279
#> class_7 = 20 / 375
#> Totals = 719 / 9,942
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, train = TRUE)`
#> =====
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1 0.927681
#> 2 2 0.996982
#> 3 3 0.999698
#> 4 4 1.000000
#> 5 5 1.000000
#> 6 6 1.000000
#> 7 7 1.000000
#>
#>
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on validation data. **
#> ** Metrics reported on temporary validation frame with 9973 samples **
#>
#> Validation Set Metrics:
#> =====
#>
#> MSE: (Extract with `h2o.mse`) 0.0638
#> RMSE: (Extract with `h2o.rmse`) 0.253
#> Logloss: (Extract with `h2o.logloss`) 0.213
#> Mean Per-Class Error: 0.138
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, valid = TRUE)`
#> =====

```

```

#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>           class_1 class_2 class_3 class_4 class_5 class_6 class_7 Error
#> class_1      3275    387     0     0     1     1    15 0.1098
#> class_2      193    4571    16     0    35    11     2 0.0532
#> class_3       0     18    544     6     2    35     0 0.1008
#> class_4       0     0     9    33     0     3     0 0.2667
#> class_5       3     27     4     0   133     0     0 0.2036
#> class_6       2     20    27     4     0   253     0 0.1732
#> class_7      18     3     0     0     0     0    322 0.0612
#> Totals       3491   5026    600    43   171   303   339 0.0844
#>
#>           Rate
#> class_1 = 404 / 3,679
#> class_2 = 257 / 4,828
#> class_3 = 61 / 605
#> class_4 = 12 / 45
#> class_5 = 34 / 167
#> class_6 = 53 / 306
#> class_7 = 21 / 343
#> Totals = 842 / 9,973
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, valid = TRUE)`
#> =====
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1 0.915572
#> 2 2 0.995588
#> 3 3 0.999799
#> 4 4 1.000000
#> 5 5 1.000000
#> 6 6 1.000000
#> 7 7 1.000000
#>
#>
#>
#>
#> Scoring History:
#>           timestamp      duration training_speed epochs
#> 1 2019-09-20 15:13:38 0.000 sec          NA 0.00000
#> 2 2019-09-20 15:13:44 5.558 sec 18903 obs/sec 0.28560
#> 3 2019-09-20 15:13:55 16.864 sec 24391 obs/sec 1.14643
#> 4 2019-09-20 15:14:05 26.830 sec 26703 obs/sec 2.00354
#> 5 2019-09-20 15:14:15 36.575 sec 27964 obs/sec 2.86512
#> 6 2019-09-20 15:14:24 46.047 sec 28855 obs/sec 3.72548
#> 7 2019-09-20 15:14:32 54.163 sec 28317 obs/sec 4.30076
#> 8 2019-09-20 15:14:42 1 min 3.539 sec 28966 obs/sec 5.16261
#> 9 2019-09-20 15:14:51 1 min 13.283 sec 29289 obs/sec 6.02348
#> 10 2019-09-20 15:15:01 1 min 23.203 sec 29472 obs/sec 6.88366
#> 11 2019-09-20 15:15:11 1 min 32.510 sec 29801 obs/sec 7.74054
#> 12 2019-09-20 15:15:20 1 min 41.827 sec 30082 obs/sec 8.60118
#> 13 2019-09-20 15:15:29 1 min 51.334 sec 30256 obs/sec 9.46029
#> 14 2019-09-20 15:15:36 1 min 57.593 sec 30395 obs/sec 10.03336
#>   iterations      samples training_rmse training_logloss training_r2
#> 1          0 0.000000          NA          NA          NA

```

```

#> 2      1  99677.000000  0.42954   0.57028   0.90760
#> 3      4  400123.000000  0.35428   0.39831   0.93714
#> 4      7  699264.000000  0.32503   0.33535   0.94709
#> 5     10  999969.000000  0.30876   0.30346   0.95226
#> 6     13  1300249.000000  0.28809   0.26678   0.95844
#> 7     15  1501031.000000  0.28144   0.25617   0.96033
#> 8     18  1801828.000000  0.26740   0.23221   0.96419
#> 9     21  2102286.000000  0.26200   0.22176   0.96562
#> 10    24  2402502.000000  0.25772   0.21608   0.96674
#> 11    27  2701563.000000  0.25290   0.20629   0.96797
#> 12    30  3001941.000000  0.24526   0.19643   0.96988
#> 13    33  3301782.000000  0.23535   0.18278   0.97226
#> 14    35  3501792.000000  0.23277   0.17826   0.97286
#>   training_classification_error validation_rmse validation_logloss
#> 1           NA          NA          NA
#> 2           0.24814    0.43012   0.57466
#> 3           0.17029    0.36301   0.41871
#> 4           0.14243    0.33326   0.35389
#> 5           0.12945    0.31775   0.32413
#> 6           0.11275    0.29831   0.28753
#> 7           0.10682    0.29290   0.27746
#> 8           0.09747    0.28134   0.25833
#> 9           0.09153    0.27138   0.24210
#> 10          0.09053    0.26804   0.23534
#> 11          0.08892    0.26551   0.23446
#> 12          0.08389    0.25881   0.22341
#> 13          0.07594    0.25030   0.20947
#> 14          0.07232    0.25255   0.21251
#>   validation_r2 validation_classification_error
#> 1           NA          NA
#> 2           0.90498    0.24647
#> 3           0.93232    0.17758
#> 4           0.94296    0.15061
#> 5           0.94814    0.13787
#> 6           0.95429    0.11822
#> 7           0.95594    0.11822
#> 8           0.95935    0.10809
#> 9           0.96217    0.09877
#> 10          0.96310    0.09646
#> 11          0.96379    0.09365
#> 12          0.96560    0.09044
#> 13          0.96782    0.08202
#> 14          0.96724    0.08443
#>
#> Variable Importances: (Extract with `h2o.varimp`)
#> =====
#>
#> Variable Importances:
#>   variable relative_importance scaled_importance
#> 1       Elevation        1.000000        1.000000
#> 2 Horizontal_Distance_To_Roadways  0.937629        0.937629
#> 3 Horizontal_Distance_To_Fire_Points  0.926569        0.926569
#> 4 Wilderness_Area.area_0            0.665309        0.665309

```

```

#> 5           Wilderness_Area.area_2          0.605383    0.605383
#> percentage
#> 1  0.047802
#> 2  0.044821
#> 3  0.044292
#> 4  0.031803
#> 5  0.028939
#>
#> ---
#>           variable relative_importance scaled_importance
#> 51      Soil_Type.type_13      0.165890    0.165890
#> 52      Soil_Type.type_14      0.155661    0.155661
#> 53      Soil_Type.type_6       0.155113    0.155113
#> 54      Soil_Type.type_35      0.152966    0.152966
#> 55      Soil_Type.missing(NA)  0.000000    0.000000
#> 56 Wilderness_Area.missing(NA) 0.000000    0.000000
#> percentage
#> 51  0.007930
#> 52  0.007441
#> 53  0.007415
#> 54  0.007312
#> 55  0.000000
#> 56  0.000000

```

Let's compare the training error with the validation and test set errors

```

h2o.performance(m3, train=T)          ## sampled training data (from model building)
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on training data. **
#> ** Metrics reported on temporary training frame with 9942 samples **
#>
#> Training Set Metrics:
#> =====
#>
#> MSE: (Extract with `h2o.mse`) 0.0542
#> RMSE: (Extract with `h2o.rmse`) 0.233
#> Logloss: (Extract with `h2o.logloss`) 0.178
#> Mean Per-Class Error: 0.109
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, train = TRUE)` )
#> =====
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>           class_1 class_2 class_3 class_4 class_5 class_6 class_7 Error
#> class_1     3354    329     0     0     4     0    20 0.0952
#> class_2     152    4547    12     1    25     6     0 0.0413
#> class_3      0     14    564    11     0    31     0 0.0903
#> class_4      0      0      5    45     0     0     0 0.1000
#> class_5      3     23      2     0   140     0     0 0.1667
#> class_6      2     19     37     3     0   218     0 0.2186
#> class_7     18      2      0     0     0     0   355 0.0533
#> Totals     3529    4934    620    60   169   255   375 0.0723
#>           Rate
#> class_1 = 353 / 3,707
#> class_2 = 196 / 4,743
#> class_3 =  56 / 620

```

```

#> class_4 =      5 / 50
#> class_5 =    28 / 168
#> class_6 =    61 / 279
#> class_7 =    20 / 375
#> Totals = 719 / 9,942
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, train = TRUE)`
#> =====
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1 0.927681
#> 2 2 0.996982
#> 3 3 0.999698
#> 4 4 1.000000
#> 5 5 1.000000
#> 6 6 1.000000
#> 7 7 1.000000
h2o.performance(m3, valid=T)           ## sampled validation data (from model building)
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on validation data. **
#> ** Metrics reported on temporary validation frame with 9973 samples **
#>
#> Validation Set Metrics:
#> =====
#>
#> MSE: (Extract with `h2o.mse`) 0.0638
#> RMSE: (Extract with `h2o.rmse`) 0.253
#> Logloss: (Extract with `h2o.logloss`) 0.213
#> Mean Per-Class Error: 0.138
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, valid = TRUE)`
#> =====
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>   class_1 class_2 class_3 class_4 class_5 class_6 class_7 Error
#> class_1    3275    387     0     0     1     1    15 0.1098
#> class_2     193    4571    16     0    35    11     2 0.0532
#> class_3      0     18    544     6     2    35     0 0.1008
#> class_4      0     0     9    33     0     3     0 0.2667
#> class_5      3     27     4     0   133     0     0 0.2036
#> class_6      2     20    27     4     0   253     0 0.1732
#> class_7     18     3     0     0     0     0    322 0.0612
#> Totals     3491    5026    600    43   171   303   339 0.0844
#>               Rate
#> class_1 = 404 / 3,679
#> class_2 = 257 / 4,828
#> class_3 = 61 / 605
#> class_4 = 12 / 45
#> class_5 = 34 / 167
#> class_6 = 53 / 306
#> class_7 = 21 / 343
#> Totals = 842 / 9,973
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, valid = TRUE)`
#> =====

```

```

#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1  0.915572
#> 2 2  0.995588
#> 3 3  0.999799
#> 4 4  1.000000
#> 5 5  1.000000
#> 6 6  1.000000
#> 7 7  1.000000
h2o.performance(m3, newdata=train)      ## full training data
#> H2OMultinomialMetrics: deeplearning
#>
#> Test Set Metrics:
#> =====
#>
#> MSE: (Extract with `h2o.mse`) 0.0554
#> RMSE: (Extract with `h2o.rmse`) 0.235
#> Logloss: (Extract with `h2o.logloss`) 0.183
#> Mean Per-Class Error: 0.12
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, <data>)`)
#> =====
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>   class_1 class_2 class_3 class_4 class_5 class_6 class_7 Error
#> class_1    115214    11181      0      0     84     14    627 0.0937
#> class_2     6086   162617     512     10    886    193     38 0.0453
#> class_3      10     428   19568    224     34   1178      0 0.0874
#> class_4      0      0    183   1390      0     85      0 0.1616
#> class_5     105    871     84      0    4651      9      0 0.1869
#> class_6      37    721   1222     68      9   8376      0 0.1972
#> class_7     675    150      0      0      3      0   11472 0.0673
#> Totals     122127  175968   21569    1692    5667   9855  12137 0.0737
#>           Rate
#> class_1 = 11,906 / 127,120
#> class_2 = 7,725 / 170,342
#> class_3 = 1,874 / 21,442
#> class_4 = 268 / 1,658
#> class_5 = 1,069 / 5,720
#> class_6 = 2,057 / 10,433
#> class_7 = 828 / 12,300
#> Totals = 25,727 / 349,015
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, <data>)`
#> =====
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1  0.926287
#> 2 2  0.996582
#> 3 3  0.999736
#> 4 4  0.999983
#> 5 5  1.000000
#> 6 6  1.000000
#> 7 7  1.000000
h2o.performance(m3, newdata=valid)      ## full validation data

```

```

#> H2OMultinomialMetrics: deeplearning
#>
#> Test Set Metrics:
#> =====
#>
#> MSE: (Extract with `h2o.mse`) 0.0618
#> RMSE: (Extract with `h2o.rmse`) 0.249
#> Logloss: (Extract with `h2o.logloss`) 0.205
#> Mean Per-Class Error: 0.134
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, <data>)` 
#> =====
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#> class_1 class_2 class_3 class_4 class_5 class_6 class_7 Error
#> class_1 38194 4033 1 0 33 5 234 0.1013
#> class_2 2277 53466 164 4 368 85 16 0.0517
#> class_3 3 170 6463 84 13 410 0 0.0952
#> class_4 0 0 74 451 0 37 0 0.1975
#> class_5 35 300 43 0 1487 5 0 0.2048
#> class_6 12 302 412 22 6 2710 0 0.2177
#> class_7 249 50 0 0 1 0 3799 0.0732
#> Totals 40770 58321 7157 561 1908 3252 4049 0.0814
#> Rate
#> class_1 = 4,306 / 42,500
#> class_2 = 2,914 / 56,380
#> class_3 = 680 / 7,143
#> class_4 = 111 / 562
#> class_5 = 383 / 1,870
#> class_6 = 754 / 3,464
#> class_7 = 300 / 4,099
#> Totals = 9,448 / 116,018
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, <data>)` 
#> =====
#> Top-7 Hit Ratios:
#> k hit_ratio
#> 1 1 0.918564
#> 2 2 0.995656
#> 3 3 0.999664
#> 4 4 0.999957
#> 5 5 1.000000
#> 6 6 1.000000
#> 7 7 1.000000
h2o.performance(m3, newdata=test)      ## full test data
#> H2OMultinomialMetrics: deeplearning
#>
#> Test Set Metrics:
#> =====
#>
#> MSE: (Extract with `h2o.mse`) 0.0614
#> RMSE: (Extract with `h2o.rmse`) 0.248
#> Logloss: (Extract with `h2o.logloss`) 0.204
#> Mean Per-Class Error: 0.134
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, <data>)` 

```

```

#> =====
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>          class_1 class_2 class_3 class_4 class_5 class_6 class_7 Error
#> class_1    37965    3974      0      0     35      1    245 0.1008
#> class_2    2257     53674    191      3    332     91     31 0.0513
#> class_3      2      170    6498     92      9    398      0 0.0936
#> class_4      0      0    70 429      0     28      0 0.1860
#> class_5     31     317     32      0   1513     10      0 0.2049
#> class_6     13     271    425     29      8   2724      0 0.2150
#> class_7    299      49      0      0      1      0   3762 0.0849
#> Totals     40567    58455    7216    553   1898   3252   4038 0.0812
#>
#>           Rate
#> class_1 = 4,255 / 42,220
#> class_2 = 2,905 / 56,579
#> class_3 = 671 / 7,169
#> class_4 = 98 / 527
#> class_5 = 390 / 1,903
#> class_6 = 746 / 3,470
#> class_7 = 349 / 4,111
#> Totals = 9,414 / 115,979
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, <data>)` 
#> =====
#> Top-7 Hit Ratios:
#> k hit_ratio
#> 1 1 0.918830
#> 2 2 0.995525
#> 3 3 0.999595
#> 4 4 0.999948
#> 5 5 1.000000
#> 6 6 1.000000
#> 7 7 1.000000

```

To confirm that the reported confusion matrix on the validation set (here, the test set) was correct, we make a prediction on the test set and compare the confusion matrices explicitly:

```

pred <- h2o.predict(m3, test)
#>
|                               | 0%
|                               |
|                               |
|=====| 100%
pred
#> predict class_1 class_2 class_3 class_4 class_5 class_6 class_7
#> 1 class_2 1.32e-01 0.8683777 3.06e-05 1.13e-06 5.96e-06 2.03e-05 1.49e-05
#> 2 class_1 1.00e+00 0.000337 4.50e-07 7.93e-07 1.06e-07 7.64e-07 2.45e-06
#> 3 class_1 9.98e-01 0.001618 6.70e-09 8.48e-10 1.18e-10 4.16e-09 2.14e-05
#> 4 class_1 9.98e-01 0.002007 1.27e-06 6.85e-07 3.29e-07 2.08e-07 4.33e-06
#> 5 class_2 7.74e-02 0.920466 6.87e-06 8.27e-06 2.10e-03 1.03e-05 1.60e-07
#> 6 class_5 2.79e-05 0.239081 1.37e-07 5.33e-09 7.61e-01 2.93e-06 3.28e-10
#>
#> [115979 rows x 8 columns]
test$Accuracy <- pred$predict == test$Cover_Type
1-mean(test$Accuracy)

```

```
#> [1] 0.0812
```

40.5.6 Hyper-parameter Tuning with Grid Search

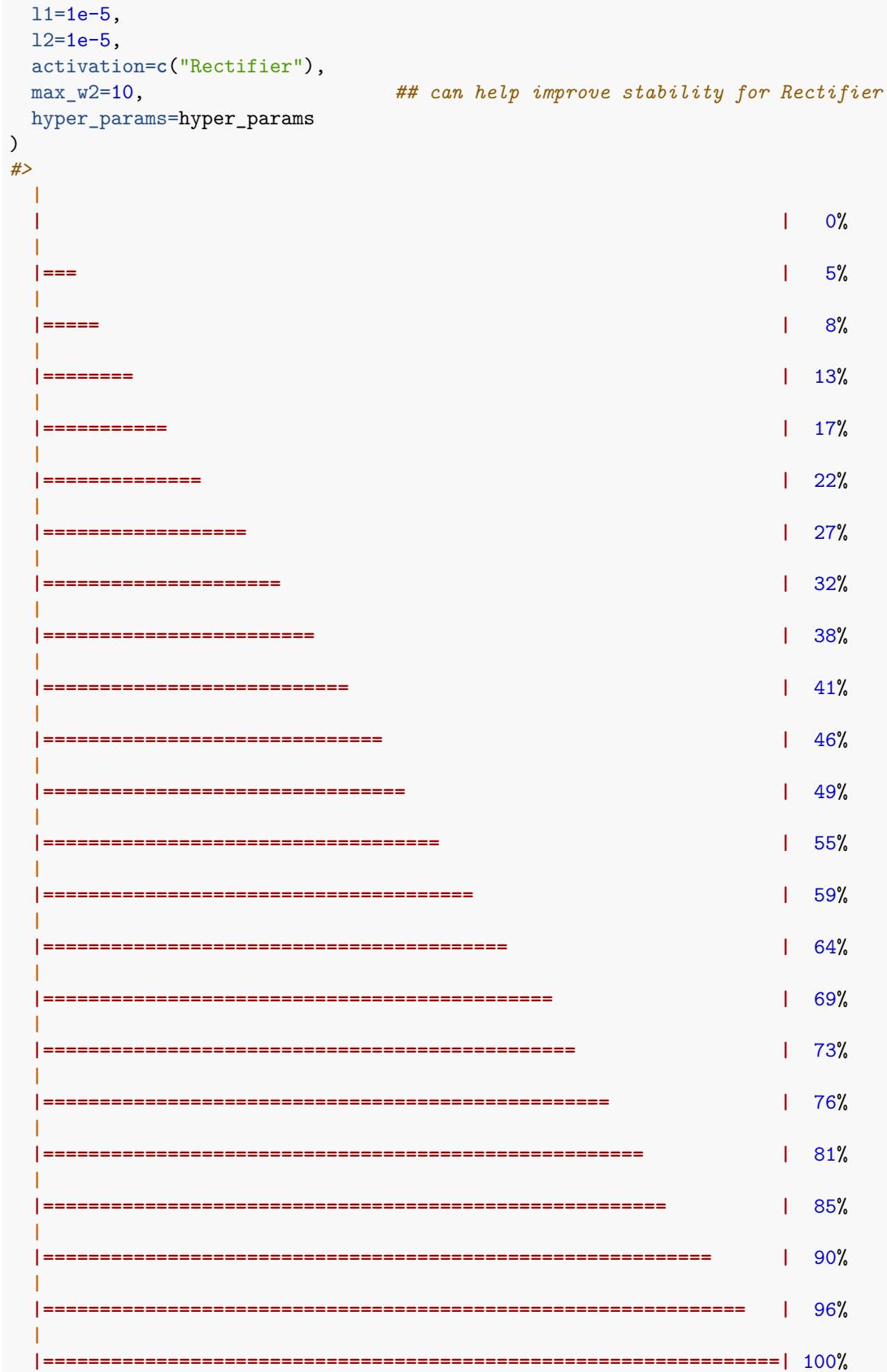
Since there are a lot of parameters that can impact model accuracy, hyper-parameter tuning is especially important for Deep Learning:

For speed, we will only train on the first 10,000 rows of the training dataset:

```
sampled_train=train[1:10000,]
```

The simplest hyperparameter search method is a brute-force scan of the full Cartesian product of all combinations specified by a grid search:

```
hyper_params <- list(
  hidden=list(c(32,32,32),c(64,64)),
  input_dropout_ratio=c(0,0.05),
  rate=c(0.01,0.02),
  rate_annealing=c(1e-8,1e-7,1e-6)
)
hyper_params
#> $hidden
#> $hidden[[1]]
#> [1] 32 32 32
#>
#> $hidden[[2]]
#> [1] 64 64
#>
#>
#> $input_dropout_ratio
#> [1] 0.00 0.05
#>
#> $rate
#> [1] 0.01 0.02
#>
#> $rate_annealing
#> [1] 1e-08 1e-07 1e-06
grid <- h2o.grid(
  algorithm="deeplearning",
  grid_id="dl_grid",
  training_frame=sampled_train,
  validation_frame=valid,
  x=predictors,
  y=response,
  epochs=10,
  stopping_metric="misclassification",
  stopping_tolerance=1e-2,          ## stop when misclassification does not improve by >=1% for 2 scoring
  stopping_rounds=2,
  score_validation_samples=10000,   ## downsample validation set for faster scoring
  score_duty_cycle=0.025,          ## don't score more than 2.5% of the wall time
  adaptive_rate=F,                ## manually tuned learning rate
  momentum_start=0.5,              ## manually tuned momentum
  momentum_stable=0.9,
  momentum_ramp=1e7,
```



```

grid
#> H2O Grid Details
#> =====
#>
#> Grid ID: dl_grid
#> Used hyper parameters:
#>   - hidden
#>   - input_dropout_ratio
#>   - rate
#>   - rate_annealing
#> Number of models: 24
#> Number of failed models: 0
#>
#> Hyper-Parameter Search Summary: ordered by increasing logloss
#>   hidden input_dropout_ratio rate rate_annealing      model_ids
#> 1 [64, 64]          0.0 0.01    1.0E-6 dl_grid_model_18
#> 2 [64, 64]          0.0 0.01    1.0E-7 dl_grid_model_10
#> 3 [64, 64]          0.0 0.01    1.0E-8 dl_grid_model_2
#> 4 [64, 64]          0.05 0.01   1.0E-8 dl_grid_model_4
#> 5 [64, 64]          0.05 0.01   1.0E-7 dl_grid_model_12
#>           logloss
#> 1 0.5655348085079595
#> 2 0.5710151424815758
#> 3 0.5859372209341824
#> 4 0.5884748164131279
#> 5 0.590855031873655
#>
#> ---
#>   hidden input_dropout_ratio rate rate_annealing      model_ids
#> 19 [32, 32, 32]       0.0 0.02    1.0E-8 dl_grid_model_5
#> 20 [64, 64]           0.05 0.02   1.0E-6 dl_grid_model_24
#> 21 [32, 32, 32]       0.0 0.01    1.0E-8 dl_grid_model_1
#> 22 [32, 32, 32]       0.0 0.02    1.0E-7 dl_grid_model_13
#> 23 [32, 32, 32]       0.05 0.01   1.0E-8 dl_grid_model_3
#> 24 [32, 32, 32]       0.0 0.02    1.0E-6 dl_grid_model_21
#>           logloss
#> 19 0.6483325937974176
#> 20 0.6494497620049914
#> 21 0.6643165406615816
#> 22 0.6853226995665436
#> 23 0.6923421232689934
#> 24 0.7942616802375754

```

Let's see which model had the lowest validation error:

```

grid <- h2o.getGrid("dl_grid", sort_by="err", decreasing=FALSE)
grid
#> H2O Grid Details
#> =====
#>
#> Grid ID: dl_grid
#> Used hyper parameters:
#>   - hidden
#>   - input_dropout_ratio

```

```

#>   - rate
#>   - rate_annealing
#> Number of models: 24
#> Number of failed models: 0
#>
#> Hyper-Parameter Search Summary: ordered by increasing err
#>           hidden input_dropout_ratio rate rate_annealing      model_ids
#> 1 [32, 32, 32]                 0.05 0.01      1.0E-7 dl_grid_model_11
#> 2 [64, 64]                   0.05 0.01      1.0E-7 dl_grid_model_12
#> 3 [64, 64]                   0.05 0.01      1.0E-6 dl_grid_model_20
#> 4 [64, 64]                   0.0 0.01       1.0E-6 dl_grid_model_18
#> 5 [64, 64]                   0.0 0.01      1.0E-7 dl_grid_model_10
#>                               err
#> 1 0.25161546873446666
#> 2 0.2533002481389578
#> 3 0.25361300682456844
#> 4 0.25504092633260134
#> 5 0.25569264720633966
#>
#> ---
#>           hidden input_dropout_ratio rate rate_annealing      model_ids
#> 19 [32, 32, 32]                0.05 0.01      1.0E-8 dl_grid_model_3
#> 20 [32, 32, 32]                0.0 0.02       1.0E-8 dl_grid_model_5
#> 21 [32, 32, 32]                0.05 0.02      1.0E-7 dl_grid_model_15
#> 22 [64, 64]                   0.05 0.02      1.0E-6 dl_grid_model_24
#> 23 [32, 32, 32]                0.0 0.02      1.0E-7 dl_grid_model_13
#> 24 [32, 32, 32]                0.0 0.02      1.0E-6 dl_grid_model_21
#>                               err
#> 19 0.27659787617711884
#> 20 0.28005204684215795
#> 21 0.28219620508679855
#> 22 0.28436684728075456
#> 23 0.2872138358492452
#> 24 0.30268698616402645

## To see what other "sort_by" criteria are allowed
#grid <- h2o.getGrid("dl_grid", sort_by="wrong_thing", decreasing=FALSE)

## Sort by logloss
h2o.getGrid("dl_grid", sort_by="logloss", decreasing=FALSE)
#> H2O Grid Details
#> =====
#>
#> Grid ID: dl_grid
#> Used hyper parameters:
#>   - hidden
#>   - input_dropout_ratio
#>   - rate
#>   - rate_annealing
#> Number of models: 24
#> Number of failed models: 0
#>
#> Hyper-Parameter Search Summary: ordered by increasing logloss

```

```

#>      hidden input_dropout_ratio rate_rate_annealing      model_ids
#> 1 [64, 64]          0.0 0.01      1.0E-6 dl_grid_model_18
#> 2 [64, 64]          0.0 0.01      1.0E-7 dl_grid_model_10
#> 3 [64, 64]          0.0 0.01      1.0E-8 dl_grid_model_2
#> 4 [64, 64]          0.05 0.01     1.0E-8 dl_grid_model_4
#> 5 [64, 64]          0.05 0.01     1.0E-7 dl_grid_model_12
#>      logloss
#> 1 0.5655348085079595
#> 2 0.5710151424815758
#> 3 0.5859372209341824
#> 4 0.5884748164131279
#> 5 0.590855031873655
#>
#> ---
#>      hidden input_dropout_ratio rate_rate_annealing      model_ids
#> 19 [32, 32, 32]      0.0 0.02      1.0E-8 dl_grid_model_5
#> 20 [64, 64]          0.05 0.02     1.0E-6 dl_grid_model_24
#> 21 [32, 32, 32]      0.0 0.01      1.0E-8 dl_grid_model_1
#> 22 [32, 32, 32]      0.0 0.02      1.0E-7 dl_grid_model_13
#> 23 [32, 32, 32]      0.05 0.01     1.0E-8 dl_grid_model_3
#> 24 [32, 32, 32]      0.0 0.02      1.0E-6 dl_grid_model_21
#>      logloss
#> 19 0.6483325937974176
#> 20 0.6494497620049914
#> 21 0.6643165406615816
#> 22 0.6853226995665436
#> 23 0.6923421232689934
#> 24 0.7942616802375754

## Find the best model and its full set of parameters
grid@summary_table[1,]
#> Hyper-Parameter Search Summary: ordered by increasing err
#>      hidden input_dropout_ratio rate_rate_annealing      model_ids
#> 1 [32, 32, 32]          0.05 0.01      1.0E-7 dl_grid_model_11
#>      err
#> 1 0.25161546873446666
best_model <- h2o.getModel(grid@model_ids[[1]])
best_model
#> Model Details:
#> =====
#>
#> # H2OMultinomialModel: deeplearning
#> Model ID: dl_grid_model_11
#> Status of Neuron Layers: predicting Cover_Type, 7-class classification, multinomial distribution, Cr
#>      layer units      type dropout      l1      l2 mean_rate rate_rms
#> 1      1    56 Input 5.00 %      NA      NA      NA      NA
#> 2      2     32 Rectifier 0.00 % 0.000010 0.000010 0.009901 0.000000
#> 3      3     32 Rectifier 0.00 % 0.000010 0.000010 0.009901 0.000000
#> 4      4     32 Rectifier 0.00 % 0.000010 0.000010 0.009901 0.000000
#> 5      5      7 Softmax      NA 0.000010 0.000010 0.009901 0.000000
#>      momentum mean_weight weight_rms mean_bias bias_rms
#> 1        NA        NA        NA        NA        NA
#> 2 0.504000 -0.005652  0.248677  0.162806  0.181253

```

```

#> 3 0.504000 -0.075886 0.255297 0.867495 0.168337
#> 4 0.504000 -0.095631 0.228149 0.834338 0.103173
#> 5 0.504000 0.017759 0.531430 -0.009904 0.667407
#>
#>
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on training data. **
#> ** Metrics reported on full training frame **
#>
#> Training Set Metrics:
#> =====
#>
#> Extract training frame with `h2o.getFrame("RTMP_sid_ab5c_9")` 
#> MSE: (Extract with `h2o.mse`) 0.183
#> RMSE: (Extract with `h2o.rmse`) 0.428
#> Logloss: (Extract with `h2o.logloss`) 0.568
#> Mean Per-Class Error: 0.456
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, train = TRUE)` 
#> =====
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#> class_1 class_2 class_3 class_4 class_5 class_6 class_7 Error
#> class_1 2698 934 1 0 9 3 43 0.2684
#> class_2 714 3965 95 0 38 11 12 0.1799
#> class_3 0 37 586 1 0 6 0 0.0698
#> class_4 0 0 33 11 0 0 0 0.7500
#> class_5 1 99 14 0 42 0 0 0.7308
#> class_6 0 68 202 0 0 39 0 0.8738
#> class_7 105 2 1 0 0 0 230 0.3195
#> Totals 3518 5105 932 12 89 59 285 0.2429
#> 
#> Rate
#> class_1 = 990 / 3,688
#> class_2 = 870 / 4,835
#> class_3 = 44 / 630
#> class_4 = 33 / 44
#> class_5 = 114 / 156
#> class_6 = 270 / 309
#> class_7 = 108 / 338
#> Totals = 2,429 / 10,000
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, train = TRUE)` 
#> =====
#> Top-7 Hit Ratios:
#> k hit_ratio
#> 1 1 0.757100
#> 2 2 0.971100
#> 3 3 0.995300
#> 4 4 0.999400
#> 5 5 0.999900
#> 6 6 1.000000
#> 7 7 1.000000
#>
#>
#> H2OMultinomialMetrics: deeplearning

```

```

#> ** Reported on validation data. **
#> ** Metrics reported on temporary validation frame with 10059 samples **
#>
#> Validation Set Metrics:
#> =====
#>
#> MSE: (Extract with `h2o.mse`) 0.19
#> RMSE: (Extract with `h2o.rmse`) 0.435
#> Logloss: (Extract with `h2o.logloss`) 0.593
#> Mean Per-Class Error: 0.472
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, valid = TRUE)` 
#> =====
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>          class_1 class_2 class_3 class_4 class_5 class_6 class_7 Error
#> class_1     2631    993      0      0      3      0     46 0.2837
#> class_2      701   3995    100      0     54     13      7 0.1797
#> class_3       0      51    589      2      1      7      0 0.0938
#> class_4       0      0     36     13      0      0      0 0.7347
#> class_5       2     113     13      0     39      0      0 0.7665
#> class_6       0      67    201      1      1     27      0 0.9091
#> class_7     118      0      1      0      0      0     234 0.3371
#> Totals     3452   5219    940     16     98     47   287 0.2516
#>          Rate
#> class_1 = 1,042 / 3,673
#> class_2 = 875 / 4,870
#> class_3 = 61 / 650
#> class_4 = 36 / 49
#> class_5 = 128 / 167
#> class_6 = 270 / 297
#> class_7 = 119 / 353
#> Totals = 2,531 / 10,059
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, valid = TRUE)` 
#> =====
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1 0.748385
#> 2 2 0.968784
#> 3 3 0.995825
#> 4 4 0.998907
#> 5 5 0.999801
#> 6 6 1.000000
#> 7 7 1.000000

print(best_model@allparameters)
#> $model_id
#> [1] "dl_grid_model_11"
#>
#> $training_frame
#> [1] "RTMP_sid_ab5c_9"
#>
#> $validation_frame
#> [1] "valid.hex"

```

```
#>
#> $nfolds
#> [1] 0
#>
#> $keep_cross_validation_models
#> [1] TRUE
#>
#> $keep_cross_validation_predictions
#> [1] FALSE
#>
#> $keep_cross_validation_fold_assignment
#> [1] FALSE
#>
#> $fold_assignment
#> [1] "AUTO"
#>
#> $ignore_const_cols
#> [1] TRUE
#>
#> $score_each_iteration
#> [1] FALSE
#>
#> $balance_classes
#> [1] FALSE
#>
#> $max_after_balance_size
#> [1] 5
#>
#> $max_confusion_matrix_size
#> [1] 20
#>
#> $max_hit_ratio_k
#> [1] 0
#>
#> $overwrite_with_best_model
#> [1] TRUE
#>
#> $use_all_factor_levels
#> [1] TRUE
#>
#> $standardize
#> [1] TRUE
#>
#> $activation
#> [1] "Rectifier"
#>
#> $hidden
#> [1] 32 32 32
#>
#> $epochs
#> [1] 10
#>
#> $train_samples_per_iteration
```

```
#> [1] -2
#>
#> $target_ratio_comm_to_comp
#> [1] 0.05
#>
#> $seed
#> [1] 8.23e+17
#>
#> $adaptive_rate
#> [1] FALSE
#>
#> $rho
#> [1] 0.99
#>
#> $epsilon
#> [1] 1e-08
#>
#> $rate
#> [1] 0.01
#>
#> $rate_annealing
#> [1] 1e-07
#>
#> $rate_decay
#> [1] 1
#>
#> $momentum_start
#> [1] 0.5
#>
#> $momentum_ramp
#> [1] 1e+07
#>
#> $momentum_stable
#> [1] 0.9
#>
#> $nesterov_accelerated_gradient
#> [1] TRUE
#>
#> $input_dropout_ratio
#> [1] 0.05
#>
#> $l1
#> [1] 1e-05
#>
#> $l2
#> [1] 1e-05
#>
#> $max_w2
#> [1] 10
#>
#> $initial_weight_distribution
#> [1] "UniformAdaptive"
#>
```

```
#> $initial_weight_scale  
#> [1] 1  
#>  
#> $loss  
#> [1] "Automatic"  
#>  
#> $distribution  
#> [1] "AUTO"  
#>  
#> $quantile_alpha  
#> [1] 0.5  
#>  
#> $tweedie_power  
#> [1] 1.5  
#>  
#> $huber_alpha  
#> [1] 0.9  
#>  
#> $score_interval  
#> [1] 5  
#>  
#> $score_training_samples  
#> [1] 10000  
#>  
#> $score_validation_samples  
#> [1] 10000  
#>  
#> $score_duty_cycle  
#> [1] 0.025  
#>  
#> $classification_stop  
#> [1] 0  
#>  
#> $regression_stop  
#> [1] 1e-06  
#>  
#> $stopping_rounds  
#> [1] 2  
#>  
#> $stopping_metric  
#> [1] "misclassification"  
#>  
#> $stopping_tolerance  
#> [1] 0.01  
#>  
#> $max_runtime_secs  
#> [1] 1.8e+308  
#>  
#> $score_validation_sampling  
#> [1] "Uniform"  
#>  
#> $diagnostics  
#> [1] TRUE
```

```
#>
#> $fast_mode
#> [1] TRUE
#>
#> $force_load_balance
#> [1] TRUE
#>
#> $variable_importances
#> [1] TRUE
#>
#> $replicate_training_data
#> [1] TRUE
#>
#> $single_node_mode
#> [1] FALSE
#>
#> $shuffle_training_data
#> [1] FALSE
#>
#> $missing_values_handling
#> [1] "MeanImputation"
#>
#> $quiet_mode
#> [1] FALSE
#>
#> $autoencoder
#> [1] FALSE
#>
#> $sparse
#> [1] FALSE
#>
#> $col_major
#> [1] FALSE
#>
#> $average_activation
#> [1] 0
#>
#> $sparsity_beta
#> [1] 0
#>
#> $max_categorical_features
#> [1] 2147483647
#>
#> $reproducible
#> [1] FALSE
#>
#> $export_weights_and_biases
#> [1] FALSE
#>
#> $mini_batch_size
#> [1] 1
#>
#> $categorical_encoding
```

```

#> [1] "AUTO"
#>
#> $elastic_averaging
#> [1] FALSE
#>
#> $elastic_averaging_moving_rate
#> [1] 0.9
#>
#> $elastic_averaging_regularization
#> [1] 0.001
#>
#> $x
#> [1] "Soil_Type"
#> [2] "Wilderness_Area"
#> [3] "Elevation"
#> [4] "Aspect"
#> [5] "Slope"
#> [6] "Horizontal_Distance_To_Hydrology"
#> [7] "Vertical_Distance_To_Hydrology"
#> [8] "Horizontal_Distance_To_Roadways"
#> [9] "Hillshade_9am"
#> [10] "Hillshade_Noon"
#> [11] "Hillshade_3pm"
#> [12] "Horizontal_Distance_To_Fire_Points"
#>
#> $y
#> [1] "Cover_Type"
print(h2o.performance(best_model, valid=T))
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on validation data. **
#> ** Metrics reported on temporary validation frame with 10059 samples **
#>
#> Validation Set Metrics:
#> =====
#>
#> MSE: (Extract with `h2o.mse`) 0.19
#> RMSE: (Extract with `h2o.rmse`) 0.435
#> Logloss: (Extract with `h2o.logloss`) 0.593
#> Mean Per-Class Error: 0.472
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, valid = TRUE)` -----
#> =====
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>           class_1 class_2 class_3 class_4 class_5 class_6 class_7 Error
#> class_1     2631    993      0      0      3      0     46 0.2837
#> class_2      701   3995    100      0     54     13      7 0.1797
#> class_3       0      51    589      2      1      7      0 0.0938
#> class_4       0      0     36     13      0      0      0 0.7347
#> class_5       2     113     13      0     39      0      0 0.7665
#> class_6       0     67    201      1      1     27      0 0.9091
#> class_7     118      0      1      0      0      0     0 234 0.3371
#> Totals     3452    5219    940     16     98     47    287 0.2516
#>           Rate
#> class_1 = 1,042 / 3,673

```

```

#> class_2 =     875 / 4,870
#> class_3 =       61 / 650
#> class_4 =        36 / 49
#> class_5 =      128 / 167
#> class_6 =      270 / 297
#> class_7 =      119 / 353
#> Totals   = 2,531 / 10,059
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, valid = TRUE)`
#> =====
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1 0.748385
#> 2 2 0.968784
#> 3 3 0.995825
#> 4 4 0.998907
#> 5 5 0.999801
#> 6 6 1.000000
#> 7 7 1.000000
print(h2o.logloss(best_model, valid=T))
#> [1] 0.593

```

40.5.7 Random Hyper-Parameter Search

Often, hyper-parameter search for more than 4 parameters can be done more efficiently with random parameter search than with grid search. Basically, chances are good to find one of many good models in less time than performing an exhaustive grid search. We simply build up to max_models models with parameters drawn randomly from user-specified distributions (here, uniform). For this example, we use the adaptive learning rate and focus on tuning the network architecture and the regularization parameters. We also let the grid search stop automatically once the performance at the top of the leaderboard doesn't change much anymore, i.e., once the search has converged.

```

hyper_params <- list(
  activation=c("Rectifier", "Tanh", "Maxout", "RectifierWithDropout", "TanhWithDropout", "MaxoutWithDropout"),
  hidden=list(c(20,20), c(50,50), c(30,30,30), c(25,25,25,25)),
  input_dropout_ratio=c(0,0.05),
  l1=seq(0,1e-4,1e-6),
  l2=seq(0,1e-4,1e-6)
)
hyper_params

## Stop once the top 5 models are within 1% of each other (i.e., the windowed average varies less than 1%)
search_criteria = list(strategy = "RandomDiscrete", max_runtime_secs = 360, max_models = 100, seed=12345)
dl_random_grid <- h2o.grid(
  algorithm="deeplearning",
  grid_id = "dl_grid_random",
  training_frame=sampled_train,
  validation_frame=valid,
  x=predictors,
  y=response,
  epochs=1,
  stopping_metric="logloss",
  stopping_tolerance=1e-2,      ## stop when logloss does not improve by >=1% for 2 scoring events

```

```

stopping_rounds=2,
score_validation_samples=10000, ## downsample validation set for faster scoring
score_duty_cycle=0.025, ## don't score more than 2.5% of the wall time
max_w2=10, ## can help improve stability for Rectifier
hyper_params = hyper_params,
search_criteria = search_criteria
)
grid <- h2o.getGrid("dl_grid_random",sort_by="logloss",decreasing=FALSE)
grid

grid@summary_table[1,]
best_model <- h2o.getModel(grid@model_ids[[1]]) ## model with lowest logloss
best_model

```

Let's look at the model with the lowest validation misclassification rate:

```

grid <- h2o.getGrid("dl_grid",sort_by="err",decreasing=FALSE)
best_model <- h2o.getModel(grid@model_ids[[1]]) ## model with lowest classification error (on validation)
h2o.confusionMatrix(best_model,valid=T)
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>          class_1 class_2 class_3 class_4 class_5 class_6 class_7 Error
#> class_1    2631     993      0      0      3      0     46 0.2837
#> class_2     701    3995     100      0     54     13      7 0.1797
#> class_3      0      51     589      2      1      7      0 0.0938
#> class_4      0      0     36     13      0      0      0 0.7347
#> class_5      2     113     13      0     39      0      0 0.7665
#> class_6      0     67    201      1      1     27      0 0.9091
#> class_7    118      0      1      0      0      0    234 0.3371
#> Totals     3452    5219     940     16     98     47    287 0.2516
#>          Rate
#> class_1 =  1,042 / 3,673
#> class_2 =   875 / 4,870
#> class_3 =    61 / 650
#> class_4 =    36 / 49
#> class_5 =   128 / 167
#> class_6 =   270 / 297
#> class_7 =   119 / 353
#> Totals = 2,531 / 10,059
best_params <- best_model@allparameters
best_params$activation
#> [1] "Rectifier"
best_params$hidden
#> [1] 32 32 32
best_params$input_dropout_ratio
#> [1] 0.05
best_params$l1
#> [1] 1e-05
best_params$l2
#> [1] 1e-05

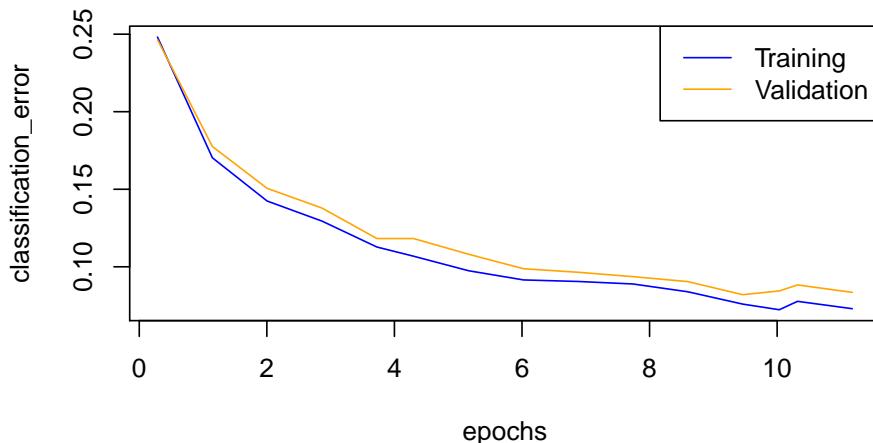
```

40.5.8 Checkpointing

Let's continue training the manually tuned model from before, for 2 more epochs. Note that since many important parameters such as epochs, l1, l2, max_w2, score_interval, train_samples_per_iteration, input_dropout_ratio, hidden_dropout_ratios, score_duty_cycle, classification_stop, regression_stop, variable_importances, force_load_balance can be modified between checkpoint restarts, it is best to specify as many parameters as possible explicitly.

```
max_epochs <- 12 ## Add two more epochs
m_cont <- h2o.deeplearning(
  model_id="dl_model_tuned_continued",
  checkpoint="dl_model_tuned",
  training_frame=train,
  validation_frame=valid,
  x=predictors,
  y=response,
  hidden=c(128,128,128),
  epochs=max_epochs,
  stopping_metric="logloss",
  stopping_tolerance=1e-2,
  stopping_rounds=2,
  score_validation_samples=10000, ## downsample validation set for faster scoring
  score_duty_cycle=0.025,
  adaptive_rate=F,
  rate=0.01,
  rate_annealing=2e-6,
  momentum_start=0.2,
  momentum_stable=0.4,
  momentum_ramp=1e7,
  l1=1e-5,
  l2=1e-5,
  max_w2=10
)
summary(m_cont)
plot(m_cont)
```

Scoring History



Once we are satisfied with the results, we can save the model to disk (on the cluster). In this example, we store the model in a directory called `mybest_deeplearning_covtype_model`, which will be created for us

since force=TRUE.

```
path <- h2o.saveModel(m_cont,
                      path = file.path(data_out_dir, "mybest_deeplearning_covtype_model"), force=TRUE)
```

It can be loaded later with the following command:

```
print(path)
#> [1] "/home/datasience/repos/machine-learning-rsuite/export/mybest_deeplearning_covtype_model/dl_mod
m_loaded <- h2o.loadModel(path)
summary(m_loaded)
#> Model Details:
#> =====
#>
#> #> H2OMultinomialModel: deeplearning
#> Model Key: dl_model_tuned_continued
#> Status of Neuron Layers: predicting Cover_Type, 7-class classification, multinomial distribution, Cr
#>   layer units      type dropout      l1      l2 mean_rate rate_rms
#> 1     1    56 Input 0.00 %      NA      NA      NA      NA
#> 2     2   128 Rectifier 0.00 % 0.000010 0.000010 0.001136 0.000000
#> 3     3   128 Rectifier 0.00 % 0.000010 0.000010 0.001136 0.000000
#> 4     4   128 Rectifier 0.00 % 0.000010 0.000010 0.001136 0.000000
#> 5     5     7 Softmax      NA 0.000010 0.000010 0.001136 0.000000
#>   momentum mean_weight weight_rms mean_bias bias_rms
#> 1       NA         NA        NA        NA        NA
#> 2 0.278029 -0.011299 0.320373 -0.009410 0.334944
#> 3 0.278029 -0.058611 0.226093 0.858487 0.365056
#> 4 0.278029 -0.060740 0.218582 0.776620 0.201415
#> 5 0.278029 -0.027095 0.268963 0.042468 0.944277
#>
#> #> H2OMultinomialMetrics: deeplearning
#> ** Reported on training data. **
#> ** Metrics reported on temporary training frame with 10112 samples **
#>
#> Training Set Metrics:
#> =====
#>
#> MSE: (Extract with `h2o.mse`) 0.0545
#> RMSE: (Extract with `h2o.rmse`) 0.233
#> Logloss: (Extract with `h2o.logloss`) 0.182
#> Mean Per-Class Error: 0.117
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, train = TRUE)`
#> =====
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>   class_1 class_2 class_3 class_4 class_5 class_6 class_7 Error
#> class_1    3447    309     0     0     1     0     12 0.0854
#> class_2     213   4586    14     0    16     8     1 0.0521
#> class_3      0      8   519     7     1    31     0 0.0830
#> class_4      0      0     7    42     0     2     0 0.1765
#> class_5      1     33     4     0   135     2     0 0.2286
#> class_6      0     17    18     1     0   283     0 0.1129
#> class_7     28      4     0     0     0     0   362 0.0812
#> Totals     3689   4957   562    50   153   326   375 0.0730
#>   Rate
#> class_1 = 322 / 3,769
```

```

#> class_2 = 252 / 4,838
#> class_3 = 47 / 566
#> class_4 = 9 / 51
#> class_5 = 40 / 175
#> class_6 = 36 / 319
#> class_7 = 32 / 394
#> Totals = 738 / 10,112
#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, train = TRUE)`
#> =====
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1 0.927017
#> 2 2 0.996934
#> 3 3 0.999901
#> 4 4 1.000000
#> 5 5 1.000000
#> 6 6 1.000000
#> 7 7 1.000000
#>
#>
#> H2OMultinomialMetrics: deeplearning
#> ** Reported on validation data. **
#> ** Metrics reported on temporary validation frame with 10022 samples **
#>
#> Validation Set Metrics:
#> =====
#>
#> MSE: (Extract with `h2o.mse`) 0.0636
#> RMSE: (Extract with `h2o.rmse`) 0.252
#> Logloss: (Extract with `h2o.logloss`) 0.211
#> Mean Per-Class Error: 0.145
#> Confusion Matrix: Extract with `h2o.confusionMatrix(<model>, valid = TRUE)`
#> =====
#> Confusion Matrix: Row labels: Actual class; Column labels: Predicted class
#>   class_1 class_2 class_3 class_4 class_5 class_6 class_7 Error
#> class_1    3322    331     0     0     3     0    19 0.0961
#> class_2     228    4579     9     0    17    17     2 0.0563
#> class_3      0     19    531     7     0    45     0 0.1179
#> class_4      0     0     5    44     0     6     0 0.2000
#> class_5      4     40     6     0   114     2     0 0.3133
#> class_6      1     15    23     0     0   261     0 0.1300
#> class_7     32      6     0     0     0     0    334 0.1022
#> Totals     3587    4990    574    51   134   331   355 0.0835
#>           Rate
#> class_1 = 353 / 3,675
#> class_2 = 273 / 4,852
#> class_3 = 71 / 602
#> class_4 = 11 / 55
#> class_5 = 52 / 166
#> class_6 = 39 / 300
#> class_7 = 38 / 372
#> Totals = 837 / 10,022

```

```

#>
#> Hit Ratio Table: Extract with `h2o.hit_ratio_table(<model>, valid = TRUE)`
#> =====
#> Top-7 Hit Ratios:
#>   k hit_ratio
#> 1 1 0.916484
#> 2 2 0.995111
#> 3 3 0.999701
#> 4 4 0.999900
#> 5 5 1.000000
#> 6 6 1.000000
#> 7 7 1.000000
#>
#>
#>
#>
#> Scoring History:
#>   timestamp duration training_speed epochs
#> 1 2019-09-20 15:13:38 0.000 sec NA 0.00000
#> 2 2019-09-20 15:13:44 5.558 sec 18903 obs/sec 0.28560
#> 3 2019-09-20 15:13:55 16.864 sec 24391 obs/sec 1.14643
#> 4 2019-09-20 15:14:05 26.830 sec 26703 obs/sec 2.00354
#> 5 2019-09-20 15:14:15 36.575 sec 27964 obs/sec 2.86512
#> 6 2019-09-20 15:14:24 46.047 sec 28855 obs/sec 3.72548
#> 7 2019-09-20 15:14:32 54.163 sec 28317 obs/sec 4.30076
#> 8 2019-09-20 15:14:42 1 min 3.539 sec 28966 obs/sec 5.16261
#> 9 2019-09-20 15:14:51 1 min 13.283 sec 29289 obs/sec 6.02348
#> 10 2019-09-20 15:15:01 1 min 23.203 sec 29472 obs/sec 6.88366
#> 11 2019-09-20 15:15:11 1 min 32.510 sec 29801 obs/sec 7.74054
#> 12 2019-09-20 15:15:20 1 min 41.827 sec 30082 obs/sec 8.60118
#> 13 2019-09-20 15:15:29 1 min 51.334 sec 30256 obs/sec 9.46029
#> 14 2019-09-20 15:15:36 1 min 57.593 sec 30395 obs/sec 10.03336
#> 15 2019-09-20 15:16:43 2 min 0.850 sec 30469 obs/sec 10.31960
#> 16 2019-09-20 15:16:53 2 min 9.993 sec 30677 obs/sec 11.17851
#>   iterations samples training_rmse training_logloss training_r2
#> 1          0 0.000000          NA          NA          NA
#> 2          1 99677.000000 0.42954 0.57028 0.90760
#> 3          4 400123.000000 0.35428 0.39831 0.93714
#> 4          7 699264.000000 0.32503 0.33535 0.94709
#> 5         10 999969.000000 0.30876 0.30346 0.95226
#> 6         13 1300249.000000 0.28809 0.26678 0.95844
#> 7         15 1501031.000000 0.28144 0.25617 0.96033
#> 8         18 1801828.000000 0.26740 0.23221 0.96419
#> 9         21 2102286.000000 0.26200 0.22176 0.96562
#> 10        24 2402502.000000 0.25772 0.21608 0.96674
#> 11        27 2701563.000000 0.25290 0.20629 0.96797
#> 12        30 3001941.000000 0.24526 0.19643 0.96988
#> 13        33 3301782.000000 0.23535 0.18278 0.97226
#> 14        35 3501792.000000 0.23277 0.17826 0.97286
#> 15        36 3601696.000000 0.24007 0.19019 0.97228
#> 16        39 3901469.000000 0.23335 0.18201 0.97381
#>   training_classification_error validation_rmse validation_logloss
#> 1                  NA                  NA                  NA

```

```

#> 2          0.24814    0.43012    0.57466
#> 3          0.17029    0.36301    0.41871
#> 4          0.14243    0.33326    0.35389
#> 5          0.12945    0.31775    0.32413
#> 6          0.11275    0.29831    0.28753
#> 7          0.10682    0.29290    0.27746
#> 8          0.09747    0.28134    0.25833
#> 9          0.09153    0.27138    0.24210
#> 10         0.09053    0.26804    0.23534
#> 11         0.08892    0.26551    0.23446
#> 12         0.08389    0.25881    0.22341
#> 13         0.07594    0.25030    0.20947
#> 14         0.07232    0.25255    0.21251
#> 15         0.07773    0.25827    0.22061
#> 16         0.07298    0.25216    0.21114
#> validation_r2 validation_classification_error
#> 1          NA          NA
#> 2          0.90498    0.24647
#> 3          0.93232    0.17758
#> 4          0.94296    0.15061
#> 5          0.94814    0.13787
#> 6          0.95429    0.11822
#> 7          0.95594    0.11822
#> 8          0.95935    0.10809
#> 9          0.96217    0.09877
#> 10         0.96310    0.09646
#> 11         0.96379    0.09365
#> 12         0.96560    0.09044
#> 13         0.96782    0.08202
#> 14         0.96724    0.08443
#> 15         0.96667    0.08831
#> 16         0.96823    0.08352
#>
#> Variable Importances: (Extract with `h2o.varimp`)
#> =====
#>
#> Variable Importances:
#>   variable relative_importance scaled_importance
#> 1           Elevation      1.000000      1.000000
#> 2   Horizontal_Distance_To_Roadways  0.936633  0.936633
#> 3   Horizontal_Distance_To_Fire_Points  0.923248  0.923248
#> 4   Wilderness_Area.area_0      0.661243  0.661243
#> 5   Wilderness_Area.area_2      0.596564  0.596564
#> percentage
#> 1  0.048200
#> 2  0.045145
#> 3  0.044500
#> 4  0.031872
#> 5  0.028754
#>
#> ---
#>   variable relative_importance scaled_importance
#> 51      Soil_Type.type_13      0.163302  0.163302

```

```
#> 52      Soil_Type.type_14      0.152124    0.152124
#> 53      Soil_Type.type_6      0.151641    0.151641
#> 54      Soil_Type.type_35      0.149299    0.149299
#> 55      Soil_Type.missing(NA) 0.000000    0.000000
#> 56 Wilderness_Area.missing(NA) 0.000000    0.000000
#>   percentage
#> 51  0.007871
#> 52  0.007332
#> 53  0.007309
#> 54  0.007196
#> 55  0.000000
#> 56  0.000000
```

This model is fully functional and can be inspected, restarted, or used to score a dataset, etc. Note that binary compatibility between H2O versions is currently not guaranteed.

40.5.9 Cross-Validation

For N-fold cross-validation, specify `nfolds>1` instead of (or in addition to) a validation frame, and $N+1$ models will be built: 1 model on the full training data, and N models with each $1/N$ -th of the data held out (there are different holdout strategies). Those N models then score on the held out data, and their combined predictions on the full training data are scored to get the cross-validation metrics.

```
dlmodel <- h2o.deeplearning(
  x=predictors,
  y=response,
  training_frame=train,
  hidden=c(10,10),
  epochs=1,
  nfolds=5,
  fold_assignment="Modulo" # can be "AUTO", "Modulo", "Random" or "Stratified"
)
dlmodel
```

N-fold cross-validation is especially useful with early stopping, as the main model will pick the ideal number of epochs from the convergence behavior of the cross-validation models.

40.6 Regression and Binary Classification

Assume we want to turn the multi-class problem above into a binary classification problem. We create a binary response as follows:

```
train$bin_response <- ifelse(train[,response] == "class_1", 0, 1)
```

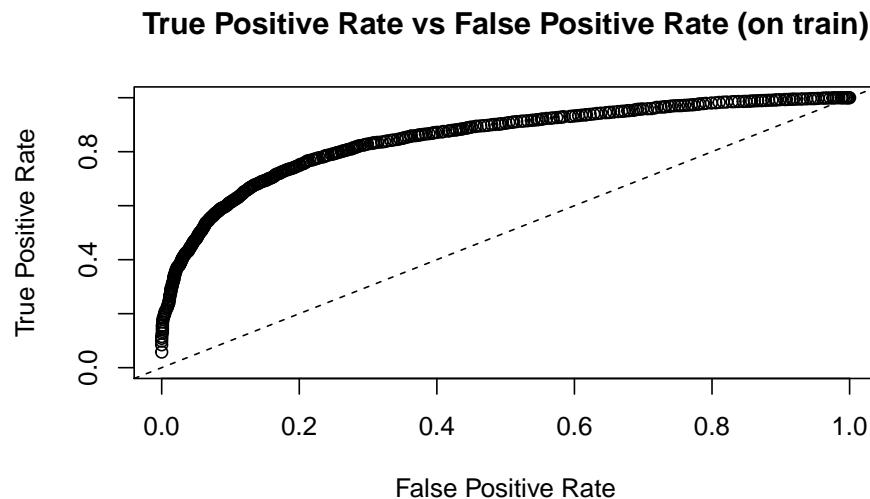
Let's build a quick model and inspect the model:

```
dlmodel <- h2o.deeplearning(
  x=predictors,
  y="bin_response",
  training_frame=train,
  hidden=c(10,10),
  epochs=0.1
)
summary(dlmodel)
```

Instead of a binary classification model, we find a regression model (H2OResponseModel) that contains only 1 output neuron (instead of 2). The reason is that the response was a numerical feature (ordinal numbers 0 and 1), and H2O Deep Learning was run with distribution=AUTO, which defaulted to a Gaussian regression problem for a real-valued response. H2O Deep Learning supports regression for distributions other than Gaussian such as Poisson, Gamma, Tweedie, Laplace. It also supports Huber loss and per-row offsets specified via an offset_column. We refer to our H2O Deep Learning regression code examples for more information.

To perform classification, the response must first be turned into a categorical (factor) feature:

```
train$bin_response <- as.factor(train$bin_response) ##make categorical
dlmodel <- h2o.deeplearning(
  x=predictors,
  y="bin_response",
  training_frame=train,
  hidden=c(10,10),
  epochs=0.1
  #balance_classes=T    ## enable this for high class imbalance
)
summary(dlmodel) ## Now the model metrics contain AUC for binary classification
plot(h2o.performance(dlmodel)) ## display ROC curve
```



Now the model performs (binary) classification, and has multiple (2) output neurons.

40.7 Unsupervised Anomaly detection

For instructions on how to build unsupervised models with H2O Deep Learning, we refer to our previous Tutorial on Anomaly Detection with H2O Deep Learning and our MNIST Anomaly detection code example, as well as our Stacked AutoEncoder R code example and another one for Unsupervised Pretraining with an AutoEncoder R code example.

40.8 H2O Deep Learning Tips & Tricks

40.8.1 Performance Tuning

The Definitive H2O Deep Learning Performance Tuning blog post covers many of the following points that affect the computational efficiency, so it's highly recommended.

40.8.2 Activation Functions

While sigmoids have been used historically for neural networks, H2O Deep Learning implements Tanh, a scaled and shifted variant of the sigmoid which is symmetric around 0. Since its output values are bounded by -1..1, the stability of the neural network is rarely endangered. However, the derivative of the tanh function is always non-zero and back-propagation (training) of the weights is more computationally expensive than for rectified linear units, or Rectifier, which is $\max(0, x)$ and has vanishing gradient for $x <= 0$, leading to much faster training speed for large networks and is often the fastest path to accuracy on larger problems. In case you encounter instabilities with the Rectifier (in which case model building is automatically aborted), try a limited value to re-scale the weights: `max_w2=10`. The Maxout activation function is computationally more expensive, but can lead to higher accuracy. It is a generalized version of the Rectifier with two non-zero channels. In practice, the Rectifier (and RectifierWithDropout, see below) is the most versatile and performant option for most problems.

40.8.3 Generalization Techniques

L1 and L2 penalties can be applied by specifying the `l1` and `l2` parameters. Intuition: L1 lets only strong weights survive (constant pulling force towards zero), while L2 prevents any single weight from getting too big. Dropout has recently been introduced as a powerful generalization technique, and is available as a parameter per layer, including the input layer. `input_dropout_ratio` controls the amount of input layer neurons that are randomly dropped (set to zero), while `hidden_dropout_ratios` are specified for each hidden layer. The former controls overfitting with respect to the input data (useful for high-dimensional noisy data), while the latter controls overfitting of the learned features. Note that `hidden_dropout_ratios` require the activation function to end with ...WithDropout.

40.8.4 Early stopping and optimizing for lowest validation error

By default, Deep Learning training stops when the `stopping_metric` does not improve by at least `stopping_tolerance` (0.01 means 1% improvement) for `stopping_rounds` consecutive scoring events on the training (or validation) data. By default, `overwrite_with_best_model` is enabled and the model returned after training for the specified number of epochs (or after stopping early due to convergence) is the model that has the best training set error (according to the metric specified by `stopping_metric`), or, if a validation set is provided, the lowest validation set error. Note that the training or validation set errors can be based on a subset of the training or validation data, depending on the values for `score_validation_samples` or `score_training_samples`, see below. For early stopping on a predefined error rate on the training data (accuracy for classification or MSE for regression), specify `classification_stop` or `regression_stop`.

40.8.5 Training Samples per MapReduce Iteration

The parameter `train_samples_per_iteration` matters especially in multi-node operation. It controls the number of rows trained on for each MapReduce iteration. Depending on the value selected, one MapReduce pass can sample observations, and multiple such passes are needed to train for one epoch. All H2O compute nodes then communicate to agree on the best model coefficients (weights/biases) so far, and the

model may then be scored (controlled by other parameters below). The default value of -2 indicates auto-tuning, which attempts to keep the communication overhead at 5% of the total runtime. The parameter `target_ratio_comm_to_comp` controls this ratio. This parameter is explained in more detail in the H2O Deep Learning booklet,

40.8.6 Categorical Data

For categorical data, a feature with K factor levels is automatically one-hot encoded (horizontalized) into K-1 input neurons. Hence, the input neuron layer can grow substantially for datasets with high factor counts. In these cases, it might make sense to reduce the number of hidden neurons in the first hidden layer, such that large numbers of factor levels can be handled. In the limit of 1 neuron in the first hidden layer, the resulting model is similar to logistic regression with stochastic gradient descent, except that for classification problems, there's still a softmax output layer, and that the activation function is not necessarily a sigmoid (Tanh). If variable importances are computed, it is recommended to turn on `use_all_factor_levels` (K input neurons for K levels). The experimental option `max_categorical_features` uses feature hashing to reduce the number of input neurons via the hash trick at the expense of hash collisions and reduced accuracy. Another way to reduce the dimensionality of the (categorical) features is to use `h2o.glm()`, we refer to the GLRM tutorial for more details.

40.8.7 Sparse Data

If the input data is sparse (many zeros), then it might make sense to enable the sparse option. This will result in the input not being standardized (0 mean, 1 variance), but only de-scaled (1 variance) and 0 values remain 0, leading to more efficient back-propagation. Sparsity is also a reason why CPU implementations can be faster than GPU implementations, because they can take advantage of if/else statements more effectively.

40.8.8 Missing Values

H2O Deep Learning automatically does mean imputation for missing values during training (leaving the input layer activation at 0 after standardizing the values). For testing, missing test set values are also treated the same way by default. See the `h2o.impute` function to do your own mean imputation.

40.8.9 Loss functions, Distributions, Offsets, Observation Weights

H2O Deep Learning supports advanced statistical features such as multiple loss functions, non-Gaussian distributions, per-row offsets and observation weights. In addition to Gaussian distributions and Squared loss, H2O Deep Learning supports Poisson, Gamma, Tweedie and Laplace distributions. It also supports Absolute and Huber loss and per-row offsets specified via an `offset_column`. Observation weights are supported via a user-specified `weights` column.

We refer to our H2O Deep Learning R test code examples for more information.

40.8.10 Exporting Weights and Biases

The model parameters (weights connecting two adjacent layers and per-neuron bias terms) can be stored as H2O Frames (like a dataset) by enabling `export_weights_and_biases`, and they can be accessed as follows:

```

|          | 0%
|-----| 100%
#>
|          | 0%
|-----| 100%
h2o.weights(iris_dl, matrix_id=1)
#> Sepal.Length Sepal.Width Petal.Length Petal.Width
#> 1 0.06976 0.09806 -0.0707 0.1154
#> 2 0.08570 0.16861 0.1448 -0.0958
#> 3 0.00658 0.05867 0.1243 0.1650
#> 4 -0.09349 -0.09891 0.0570 -0.1430
#> 5 0.14242 0.09565 -0.1405 -0.1648
#> 6 0.04994 0.00555 0.0950 -0.1218
#>
#> [200 rows x 4 columns]
h2o.weights(iris_dl, matrix_id=2)
#> C1      C2      C3      C4      C5      C6      C7      C8
#> 1 -0.00377 -0.04174 -0.0131 -0.11203 -0.10178 -0.117465 -0.11005 0.0344
#> 2 -0.08431 -0.08589 -0.0572 0.10628 0.00912 -0.038027 -0.05764 0.0590
#> 3 0.10611 0.00562 -0.0576 -0.01460 0.07344 -0.026180 0.00163 0.0257
#> 4 -0.06380 -0.09231 -0.0759 -0.02595 -0.00754 0.018112 0.11962 0.0780
#> 5 -0.11794 -0.03649 -0.0553 0.10858 -0.10098 -0.062671 0.05638 0.0626
#> 6 -0.04597 0.11525 0.0407 0.00922 0.08802 -0.000307 -0.02639 -0.0731
#> C9      C10     C11     C12     C13     C14     C15     C16
#> 1 -0.0719 -0.0156 0.08596 -0.08704 -0.0714 0.0415 -0.0745 0.1037
#> 2 -0.0154 0.0699 -0.03310 0.09892 -0.0858 -0.0330 0.0861 -0.0398
#> 3 0.0275 0.0296 -0.05978 -0.07290 -0.0615 0.0363 0.0955 0.0890
#> 4 -0.1055 0.0597 0.03338 -0.01178 0.0603 0.0960 0.0301 0.1220
#> 5 -0.0567 -0.1251 -0.01381 -0.00642 -0.0288 0.1130 -0.1244 -0.1071
#> 6 0.1005 0.1172 0.00832 0.08649 -0.0327 0.0391 -0.0455 -0.0942
#> C17     C18     C19     C20     C21     C22     C23     C24
#> 1 -0.1051 -0.0525 0.07253 0.0787 -0.0465 0.04187 0.0279 0.0800
#> 2 0.0855 0.0679 -0.00729 0.0417 0.0978 0.10999 -0.1012 -0.0305
#> 3 -0.0704 -0.0866 0.03215 0.0869 -0.1067 0.02024 -0.0973 0.0783
#> 4 0.0330 -0.0194 -0.05555 0.0428 -0.0954 0.05979 0.0573 0.0539
#> 5 -0.1090 -0.0831 -0.03706 -0.0530 -0.0235 -0.12264 0.0427 0.1037
#> 6 -0.0308 0.0310 -0.11167 -0.0129 -0.1004 -0.00906 0.0633 0.0602
#> C25     C26     C27     C28     C29     C30     C31     C32
#> 1 0.0167 -0.0499 -0.0347779 -0.0710 0.0180 -0.0595 0.0946 -0.0316
#> 2 0.0839 0.0187 0.000504 -0.0454 -0.0916 0.0967 -0.0901 0.0432
#> 3 0.0721 0.0379 0.102506 0.1121 0.1058 -0.1099 0.0628 -0.0910
#> 4 0.0641 0.0237 0.077027 -0.1125 0.0620 -0.0906 -0.0512 0.0147
#> 5 0.0941 -0.1066 0.107501 0.0827 -0.0724 -0.0843 -0.0773 0.0129
#> 6 -0.1061 -0.0969 -0.101528 -0.0073 0.0169 -0.0509 0.0177 -0.0679
#> C33     C34     C35     C36     C37     C38     C39     C40
#> 1 -0.0570 0.0888 -0.01204 0.06730 -0.06546 -0.0266 -0.1184 0.01854
#> 2 -0.0677 -0.1186 -0.08013 -0.00764 -0.11780 -0.0494 -0.0620 -0.05213
#> 3 0.0960 0.0788 -0.04687 0.07302 0.12392 0.0815 0.0560 -0.05292
#> 4 0.1125 0.0517 -0.00872 0.04115 -0.00382 0.0824 0.0392 -0.05460

```

```

#> 5  0.0139  0.0277 -0.02410  0.00903 -0.08751 -0.0754  0.0790  0.02989
#> 6  0.0363  0.1400  0.02019  0.00907  0.09339  0.0097 -0.0141 -0.00437
#>    C41     C42     C43     C44     C45     C46     C47     C48
#> 1  0.09932 -0.02824  0.0970  0.00485  0.00835  0.0440  0.0810  0.0598
#> 2  0.08903  0.08798  0.0588 -0.03962  0.10481  0.0551  0.0389  0.0122
#> 3  0.02747 -0.00184 -0.0964  0.05819  0.02018  0.0520 -0.0526 -0.0421
#> 4  0.04287 -0.01656 -0.1153  0.10227 -0.11010  0.0783 -0.0314 -0.1167
#> 5 -0.11639  0.09970 -0.1193 -0.06725  0.04144 -0.0547 -0.1028 -0.0169
#> 6 -0.00681 -0.07943  0.0253  0.10209 -0.01227  0.0629 -0.0904 -0.0914
#>    C49     C50     C51     C52     C53     C54     C55     C56
#> 1  0.0587 -0.05867  0.0337  0.03401  0.0675 -0.05641 -0.0141  0.1214
#> 2  0.0477  0.06714 -0.0233  0.05875  0.0935 -0.00106 -0.0182 -0.0347
#> 3  0.0222 -0.07296 -0.0826 -0.10406 -0.0464 -0.06580 -0.0774  0.0391
#> 4 -0.1055 -0.06009  0.1103  0.08981 -0.0563 -0.04074 -0.0624  0.0853
#> 5 -0.0229 -0.00823  0.0932  0.10623 -0.0812  0.04683  0.0532 -0.0178
#> 6 -0.0802 -0.00374  0.0684  0.00065  0.0636  0.10308 -0.0117  0.0543
#>    C57     C58     C59     C60     C61     C62     C63     C64
#> 1  0.06464 -0.0875 -0.12534 -0.1017 -0.0806 -0.08198 -0.00313  0.0640
#> 2  0.01718 -0.0647 -0.07944 -0.0268  0.0500 -0.02901 -0.00279 -0.0398
#> 3 -0.03813 -0.0395  0.08025  0.0453 -0.0255  0.00533 -0.07721  0.0114
#> 4  0.06052  0.0573  0.02448 -0.0264 -0.0363 -0.04703  0.08987 -0.0535
#> 5  0.00484 -0.0460  0.00258 -0.0911  0.0399 -0.12054 -0.07178  0.0154
#> 6  0.11060 -0.0931 -0.06056  0.0570  0.0529  0.12127 -0.10266 -0.0140
#>    C65     C66     C67     C68     C69     C70     C71     C72
#> 1  0.0216  0.035300  0.0281  0.0426  0.0322 -0.00837 -0.1069  0.0608
#> 2  0.1090 -0.000995  0.0376  0.0732 -0.0203  0.10940  0.1067 -0.0273
#> 3 -0.0977 -0.097374 -0.0208  0.0587  0.0490 -0.08462 -0.0824 -0.0486
#> 4  0.0378  0.065128  0.1088 -0.1001 -0.0971  0.08386  0.0326 -0.0570
#> 5  0.0185  0.091271  0.1000  0.1158 -0.1202 -0.05090  0.0707  0.0180
#> 6  0.0663 -0.111192  0.0964  0.1116 -0.0160 -0.00503  0.0740  0.0854
#>    C73     C74     C75     C76     C77     C78     C79     C80
#> 1  0.0353  0.118306 -0.09539 -0.0605 -0.00387  0.0236 -0.0154  0.1069
#> 2 -0.0631  0.035527  0.09136 -0.0700 -0.02152 -0.1022 -0.0560  0.0686
#> 3 -0.1112 -0.030857 -0.11710 -0.0237  0.04397 -0.0840  0.0929  0.0863
#> 4  0.0863  0.000256  0.05556  0.0333 -0.08189  0.0688 -0.0824  0.0224
#> 5 -0.0287  0.068762 -0.06296 -0.0755 -0.03693  0.0176  0.0614 -0.1055
#> 6 -0.0627  0.110967 -0.00217 -0.0892  0.05248 -0.0705 -0.1093 -0.0446
#>    C81     C82     C83     C84     C85     C86     C87     C88
#> 1  0.0542  0.1078  0.1213 -0.115944  0.03729  0.02979 -0.0250  0.0880
#> 2  0.0299  0.0497  0.0816  0.019713  0.00425  0.03484  0.0315 -0.0258
#> 3  0.1043  0.0244 -0.0530  0.000481  0.00506  0.06324  0.0825  0.0691
#> 4 -0.0484  0.0901  0.0515  0.015654  0.09953 -0.00674 -0.0828  0.1057
#> 5 -0.0716  0.0471  0.0958 -0.019437 -0.08262 -0.02651  0.0679  0.0807
#> 6 -0.0816 -0.0300 -0.0924  0.070163 -0.01492  0.09377  0.0246  0.0571
#>    C89     C90     C91     C92     C93     C94     C95     C96
#> 1  0.0498 -0.01297 -0.0171 -0.0204 -0.0764 -0.0347 -0.0546 -0.10222
#> 2 -0.0144 -0.07934  0.0909  0.0488 -0.0961  0.0903  0.1013  0.07162
#> 3 -0.0360 -0.10760  0.1190  0.0394 -0.0290  0.0266 -0.1170 -0.00287
#> 4 -0.0562 -0.00604 -0.1202 -0.0297  0.0437  0.1212 -0.1156 -0.05647
#> 5 -0.0703  0.07844 -0.0807  0.0149 -0.0989  0.1154 -0.0481 -0.01692
#> 6 -0.1082  0.05962 -0.0859 -0.0700 -0.1046 -0.0455  0.1083 -0.09186
#>    C97     C98     C99     C100    C101    C102    C103    C104
#> 1  0.0733  0.1105 -0.1022 -0.0742 -0.06339  0.07266 -0.08954  0.00428

```

```

#> 2  0.0581  0.0659  0.1192  0.0168 -0.00323 -0.03385 -0.03187  0.07758
#> 3 -0.0562  0.0710 -0.0384  0.0429 -0.01505  0.06142  0.11854 -0.01580
#> 4  0.0773 -0.1137  0.0731  0.1095 -0.11849 -0.00227  0.07316  0.12024
#> 5 -0.1221 -0.0893  0.0254  0.0702 -0.07969 -0.11174  0.08665  0.00380
#> 6  0.0301 -0.0726  0.0146 -0.1029  0.12458  0.10706 -0.00359 -0.11074
#>   C105    C106    C107    C108    C109    C110    C111    C112    C113
#> 1  0.0739  0.0101  0.0206  0.122   0.0517  0.0607  0.1118 -0.00823  0.0900
#> 2 -0.0223  0.1158 -0.1012  0.113   0.0891 -0.0750 -0.1070 -0.11625 -0.0659
#> 3  0.1099 -0.0961 -0.0473  0.126   -0.0257 -0.0269 -0.0307 -0.11603  0.0415
#> 4  0.0359 -0.1226  0.0507  0.067   -0.0836 -0.0749  0.0667  0.05283  0.0727
#> 5  0.0359 -0.0923  0.0783  0.082   0.0589  0.0298  0.1002  0.07960 -0.0505
#> 6  0.0223  0.0962 -0.1091  0.100   0.0874 -0.0147  0.0208  0.03547  0.0917
#>   C114    C115    C116    C117    C118    C119    C120    C121
#> 1 -0.0337 -0.10484 -0.06381 -0.1035  0.04573  0.01921 -0.0926 -0.02637
#> 2  0.0839  0.00837 -0.01622 -0.0107  0.02460 -0.05904 -0.0931  0.00474
#> 3 -0.0407  0.01208  0.02241  0.0879 -0.07836 -0.00733  0.0962 -0.06328
#> 4  0.0674  0.09862 -0.00846 -0.1039  0.10639 -0.02641 -0.0770  0.07318
#> 5  0.0974 -0.08817  0.11588 -0.1085  0.00712  0.01767 -0.1234 -0.01964
#> 6 -0.1139 -0.00424 -0.08683 -0.0363 -0.07593  0.00190 -0.0578 -0.05446
#>   C122    C123    C124    C125    C126    C127    C128    C129
#> 1  0.03674  0.1237 -0.04520 -0.0306 -0.0006  0.0956  0.0549 -0.0365
#> 2 -0.09247 -0.1032  0.00108 -0.0504 -0.1196  0.0696 -0.0168  0.1106
#> 3 -0.00997  0.0917 -0.04865 -0.0474 -0.0735  0.0632 -0.0634 -0.0212
#> 4  0.07298 -0.0675 -0.01196 -0.0786  0.0490 -0.1089  0.0633  0.1110
#> 5 -0.09434 -0.0185  0.03656  0.0810 -0.0625  0.0937  0.0646  0.0848
#> 6 -0.08288  0.0630  0.01502 -0.0381 -0.0436 -0.1136  0.0679  0.0128
#>   C130    C131    C132    C133    C134    C135    C136    C137
#> 1  0.1094  0.03566 -0.00416  0.0793 -0.0716  0.0522 -0.06937 -0.05811
#> 2 -0.0294  0.01689 -0.07275  0.0711  0.0746  0.0742  0.06364  0.07468
#> 3  0.1089  0.00647  0.10485  0.0229  0.1020  0.1055  0.09774  0.12125
#> 4 -0.0522 -0.12090  0.10145  0.0843  0.0789  0.0595 -0.00789 -0.11945
#> 5 -0.0271 -0.11823  0.11296  0.0848 -0.0895 -0.0453  0.04522  0.00928
#> 6  0.0828  0.12114 -0.05873 -0.0632 -0.1148 -0.1225 -0.12078  0.11636
#>   C138    C139    C140    C141    C142    C143    C144    C145
#> 1 -0.0779 -0.0631  0.07777 -0.11201  0.00311 -0.1156  0.1226  0.0405
#> 2 -0.1163 -0.0163 -0.02705 -0.00984  0.04533  0.0614  0.0769 -0.0502
#> 3  0.0634 -0.0833 -0.05700  0.01440  0.04347  0.0483  0.0436 -0.0310
#> 4  0.0866  0.0292 -0.09940 -0.00437 -0.09841  0.0676  0.1093  0.1174
#> 5 -0.0982 -0.1104  0.00545 -0.09842  0.05746 -0.0560 -0.0477 -0.0236
#> 6  0.1167  0.0919 -0.04769  0.00281 -0.07250 -0.0707  0.0811  0.0252
#>   C146    C147    C148    C149    C150    C151    C152    C153
#> 1 -0.0446  0.051756  0.0204 -0.10556 -0.09319  0.1024 -0.1053  0.0131
#> 2  0.0175  0.002490  0.1199 -0.10575 -0.00637 -0.0945  0.1138  0.1064
#> 3  0.0315 -0.000557 -0.0663  0.02473  0.00801 -0.0731 -0.0099 -0.0533
#> 4  0.0363  0.110414  0.0865 -0.02161  0.02591  0.0187  0.0214  0.1200
#> 5  0.0353 -0.116950  0.0295 -0.00867  0.05730  0.1190 -0.0641 -0.0743
#> 6 -0.0363 -0.065581 -0.0586 -0.06431  0.01773  0.0332  0.0032 -0.1068
#>   C154    C155    C156    C157    C158    C159    C160    C161    C162
#> 1 -0.0742  0.0638 -0.1176 -0.0519 -0.0896 -0.11058  0.0315 -0.0246  0.0725
#> 2 -0.0546 -0.0351  0.0822  0.0386 -0.0407  0.10167  0.0842  0.0155  0.0232
#> 3  0.0185  0.0925 -0.1085  0.0350  0.0998  0.06522  0.0380  0.1071 -0.1132
#> 4 -0.1138 -0.1047 -0.1144  0.1047 -0.1101  0.00760  0.0680 -0.0235  0.0597
#> 5  0.0410  0.1011  0.0186 -0.1134  0.0829 -0.00998  0.1049 -0.0620  0.0325

```

```

#> 6 -0.1006 -0.1060 -0.0507  0.0699 -0.0810  0.03061 0.0830  0.0700  0.0897
#>   C163   C164   C165   C166   C167   C168   C169   C170
#> 1  0.0884 -0.1224 -0.0567 -0.0511  0.03312 0.06697 0.04004 -0.1166
#> 2  0.1050  0.0641 -0.0712 -0.0185 -0.01521 -0.04217 0.05469 -0.0865
#> 3 -0.1100 -0.0946  0.0529  0.0629  0.07493 0.00368 0.03301 0.0927
#> 4 -0.0668 -0.1182 -0.0306 -0.0694  0.00644 0.11020 0.04253 0.0938
#> 5 -0.1021  0.0284  0.0737 -0.0791 -0.06571 -0.06951 0.00477 0.0553
#> 6 -0.0951 -0.0398 -0.0636 -0.0611  0.09139 0.06308 -0.09140 -0.1021
#>   C171   C172   C173   C174   C175   C176   C177   C178
#> 1 -0.097466 -0.0892  0.0621 -0.0225  0.0276 -0.02481 -0.02909 0.1071
#> 2 -0.000841  0.0400 -0.1109  0.0279 -0.0747 -0.03990 0.07722 -0.0270
#> 3  0.025534  0.0963 -0.0939  0.0458 -0.0116  0.00887 0.05721 -0.0356
#> 4 -0.018872  0.0186 -0.0737  0.0114 -0.1107 -0.06354 -0.03085 0.0275
#> 5 -0.072176  0.0960 -0.0872  0.1130 -0.0462  0.04666 -0.05809 -0.0400
#> 6  0.045227 -0.0873 -0.1229  0.0859  0.1141  0.01828 0.00256 0.0851
#>   C179   C180   C181   C182   C183   C184   C185   C186   C187
#> 1  0.1158  0.0483 -0.1087  0.0204 -0.0801  0.1150 -0.00764 -0.0996 -0.0280
#> 2  0.0255 -0.0242 -0.0147 -0.0687  0.0958  0.0179 -0.11765 -0.0254 0.0715
#> 3  0.0127  0.0543  0.0099 -0.0356  0.0879  0.0908  0.11311 -0.0268 -0.0218
#> 4  0.0850 -0.0154 -0.0685 -0.0515 -0.0645 -0.0459  0.01678 0.0616 -0.0374
#> 5 -0.1246 -0.0758  0.0271 -0.0225 -0.0116 -0.0200  0.00247 -0.0165 -0.0770
#> 6  0.1152  0.0252  0.1172 -0.0365  0.0659  0.0993 -0.09316 -0.0675 0.0548
#>   C188   C189   C190   C191   C192   C193   C194   C195
#> 1  0.0850 -0.0344 -0.00327 -0.1018  0.0448  0.099353 0.0685 0.10706
#> 2  0.0154 -0.1148 -0.09576 -0.0554 -0.0829  0.090553 0.0120 0.12670
#> 3  0.0797 -0.1154 -0.06220  0.1137  0.0873 -0.000183 -0.1110 -0.08733
#> 4 -0.0701 -0.0110 -0.05203 -0.1060  0.0894  0.053941 0.0608 0.03834
#> 5  0.0018 -0.0970 -0.12031 -0.0553 -0.0341 -0.090961 0.0222 0.00542
#> 6 -0.0941  0.1250  0.10445 -0.0366 -0.0664 -0.073620 0.0849 0.11440
#>   C196   C197   C198   C199   C200
#> 1  0.0139 -0.000847  0.0977  0.0292 -0.0211
#> 2 -0.0577  0.054134 -0.0411  0.0177 -0.1136
#> 3 -0.1171  0.084142 -0.0184 -0.0742  0.0254
#> 4 -0.0309 -0.071778  0.0967  0.0785  0.0418
#> 5  0.0604 -0.061921 -0.0995  0.0962 -0.1252
#> 6  0.0905  0.112109 -0.0810  0.0379 -0.0375
#>
#> [200 rows x 200 columns]
h2o.weights(iris_dl, matrix_id=3)
#>   C1    C2    C3    C4    C5    C6    C7    C8    C9    C10   C11
#> 1  0.342 -0.368 0.324 -0.364 -0.592 -0.203 0.585 0.516 0.0173 0.035 0.568
#> 2  0.482  0.293 0.359  0.228 -0.175  0.425 0.406 0.172 0.4495 0.650 0.113
#> 3  0.183 -0.666 0.292 -0.424  0.683  0.490 0.562 0.151 -0.0929 0.306 0.609
#>   C12   C13   C14   C15   C16   C17   C18   C19   C20   C21
#> 1 -0.00214 -0.589 -0.397 -0.598  0.660  0.230 -0.195 0.0347 0.141 0.2215
#> 2  0.26457 -0.236 0.672 -0.518  0.209 -0.540 -0.302 -0.5254 -0.612 0.0479
#> 3 -0.24274 0.248 -0.533  0.372 -0.144  0.421 -0.613 -0.1978 0.519 0.6486
#>   C22   C23   C24   C25   C26   C27   C28   C29   C30   C31
#> 1 -0.197 -0.580 0.357  0.0808 -0.188 0.533 0.365 -0.322 -0.153 0.0178
#> 2 -0.540  0.640 0.313 -0.2145  0.429 0.138 -0.625 -0.202 0.121 -0.1841
#> 3  0.649 -0.488 -0.197 -0.5021 -0.389 0.672 -0.364 0.287 -0.640 0.2210
#>   C32   C33   C34   C35   C36   C37   C38   C39   C40   C41
#> 1  0.332 0.460 -0.32759 -0.108 -0.445 0.657 -0.1124 0.558 -0.448 -0.513

```

```

#> 2 -0.294 0.391 -0.58829 0.647 -0.184 -0.274 0.2036 0.405 -0.446 -0.220
#> 3 0.313 0.673 0.00415 -0.196 -0.390 0.482 -0.0456 -0.159 -0.205 -0.430
#> C42 C43 C44 C45 C46 C47 C48 C49 C50
#> 1 -0.532 -0.295 -0.496 -0.381 0.362 -0.511 -0.0958 -0.4112 -0.6348
#> 2 -0.115 -0.612 -0.192 0.484 0.217 0.154 -0.2259 0.0227 -0.0495
#> 3 -0.614 0.607 -0.378 0.505 -0.280 0.566 0.0986 -0.2556 0.5727
#> C51 C52 C53 C54 C55 C56 C57 C58 C59 C60
#> 1 0.3852 0.274 -0.272 -0.441 0.559 0.667 -0.2140 -0.3457 0.137 -0.180
#> 2 -0.0425 -0.391 -0.156 -0.407 0.212 -0.189 -0.5148 0.5274 -0.507 0.418
#> 3 0.6496 -0.256 0.397 0.273 -0.523 0.115 -0.0471 0.0869 0.164 -0.484
#> C61 C62 C63 C64 C65 C66 C67 C68 C69
#> 1 -0.43399 0.236 -0.627 -0.1996 -0.337 0.491 -0.1129 0.0639 0.0326
#> 2 0.00354 0.294 -0.630 0.1598 -0.201 -0.305 0.0646 -0.3334 0.3836
#> 3 -0.04962 0.484 -0.261 0.0953 0.284 0.171 -0.2263 0.2749 -0.6304
#> C70 C71 C72 C73 C74 C75 C76 C77 C78 C79
#> 1 -0.350 0.563 0.0541 0.6424 0.246 -0.496 -0.114 -0.2509 0.138 0.406
#> 2 0.480 0.534 -0.4180 -0.0151 -0.185 -0.595 0.566 -0.0383 0.325 0.149
#> 3 -0.416 -0.120 -0.5839 -0.6403 0.532 -0.340 0.214 0.2159 -0.498 0.116
#> C80 C81 C82 C83 C84 C85 C86 C87 C88 C89
#> 1 -0.674 -0.663 -0.206 -0.634 -0.413 -0.3357 -0.231 -0.435 -0.164 0.0199
#> 2 0.522 -0.618 0.681 0.433 -0.486 0.1022 -0.576 0.314 -0.435 0.1380
#> 3 -0.443 -0.498 0.464 0.125 0.413 -0.0961 -0.424 0.589 -0.349 -0.4965
#> C90 C91 C92 C93 C94 C95 C96 C97 C98 C99
#> 1 -0.239 -0.407 -0.3047 -0.664 -0.108 0.504 -0.384 0.0026 0.602 0.0933
#> 2 0.186 0.485 0.0146 -0.573 -0.129 -0.324 0.227 -0.3302 0.296 -0.0159
#> 3 0.155 -0.443 -0.4446 0.127 -0.490 0.452 -0.555 0.1478 -0.272 0.2311
#> C100 C101 C102 C103 C104 C105 C106 C107 C108 C109
#> 1 0.668 0.196 -0.5353 0.687 -0.155 0.233 0.145 0.43366 -0.260 -0.561
#> 2 0.114 -0.476 -0.0474 -0.202 0.489 -0.293 0.415 -0.07614 0.222 -0.161
#> 3 0.633 0.217 0.3521 0.181 0.406 0.245 0.401 0.00776 -0.276 -0.382
#> C110 C111 C112 C113 C114 C115 C116 C117 C118 C119
#> 1 -0.547 -0.559 0.142 0.202 -0.6393 -0.235 0.325 0.384 0.661 -0.450
#> 2 -0.610 0.414 0.169 -0.313 0.0841 0.172 0.525 0.438 -0.668 -0.164
#> 3 0.620 -0.486 0.345 0.111 0.6790 -0.371 0.287 0.410 0.494 -0.357
#> C120 C121 C122 C123 C124 C125 C126 C127 C128 C129
#> 1 -0.488 0.354 0.266 -0.6560 -0.684 0.495 -0.577 -0.172 0.33653 -0.632
#> 2 0.204 0.273 -0.454 -0.0724 -0.280 0.455 0.265 0.320 -0.00751 0.188
#> 3 -0.043 -0.108 0.675 -0.1612 0.648 0.376 0.178 -0.435 -0.60044 0.164
#> C130 C131 C132 C133 C134 C135 C136 C137 C138 C139
#> 1 0.196 0.1892 0.1882 0.575 -0.1521 0.0399 0.603 0.371 -0.5318 0.660
#> 2 -0.175 0.6168 0.0231 -0.680 0.0942 0.2942 -0.512 0.223 -0.1942 0.504
#> 3 0.506 0.0157 -0.5676 -0.630 -0.6126 0.0769 -0.505 -0.221 -0.0263 0.100
#> C140 C141 C142 C143 C144 C145 C146 C147 C148
#> 1 0.0922 -0.0641 -0.391 0.614 0.230 -0.0131 -0.558 0.632 -0.3995
#> 2 -0.5530 0.4508 -0.119 -0.570 -0.332 0.4490 0.452 -0.353 -0.0845
#> 3 -0.0205 -0.5195 -0.475 0.377 0.579 0.6530 0.682 -0.541 -0.6666
#> C149 C150 C151 C152 C153 C154 C155 C156 C157
#> 1 0.652 -0.255 -0.662 -0.29231 0.5346 -0.6735 0.410 0.362 0.3094
#> 2 -0.620 -0.369 -0.114 0.00293 0.0844 0.3824 0.200 -0.159 -0.6263
#> 3 0.342 -0.397 0.513 -0.43492 -0.4018 0.0382 0.227 0.245 -0.0769
#> C158 C159 C160 C161 C162 C163 C164 C165 C166
#> 1 -0.6355 -0.0757 0.00464 0.477 0.469 0.392 0.2368 -0.36293 0.261
#> 2 -0.0662 -0.1898 0.29747 -0.368 -0.570 0.149 0.0281 0.00839 -0.408

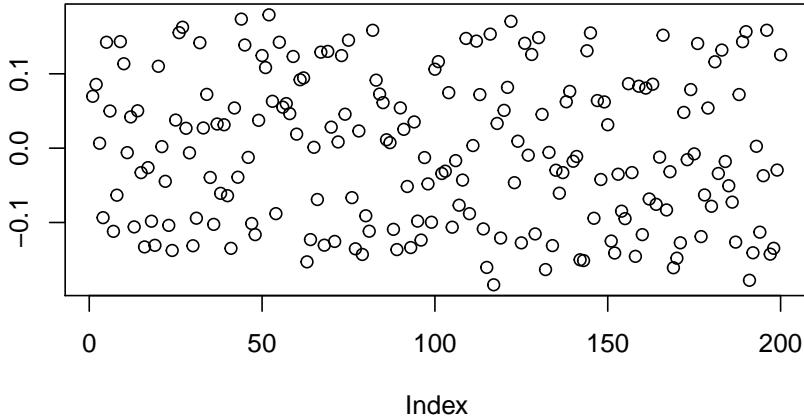
```

```

#> 3 -0.5992 -0.0279 0.47702 0.247 0.120 0.327 0.0274 0.49503 0.527
#> C167 C168 C169 C170 C171 C172 C173 C174 C175 C176
#> 1 -0.0771 0.618 -0.587 0.299 0.4908 -0.2277 0.376 -0.648 -0.546 -0.314
#> 2 0.6645 0.602 0.546 -0.158 -0.2703 -0.3597 0.241 -0.202 0.609 -0.520
#> 3 -0.6659 0.404 0.161 0.453 -0.0947 0.0698 -0.179 -0.388 -0.160 0.631
#> C177 C178 C179 C180 C181 C182 C183 C184 C185
#> 1 0.0109 -0.447 -0.428 -0.213 0.0703 0.624 0.0255 0.5505 -0.2055
#> 2 -0.1831 -0.352 0.331 0.301 0.4346 -0.659 0.6451 -0.6585 0.2983
#> 3 0.0763 -0.288 -0.631 0.383 0.3603 0.554 -0.0247 0.0292 -0.0477
#> C186 C187 C188 C189 C190 C191 C192 C193 C194 C195
#> 1 0.0703 -0.440 -0.328 0.5271 0.231 0.671 -0.296 0.242 0.598 -0.479
#> 2 -0.3119 0.604 0.159 -0.2755 0.466 -0.222 0.590 -0.620 0.185 -0.301
#> 3 -0.5587 -0.168 -0.497 0.0699 0.283 -0.245 0.613 -0.644 -0.301 0.579
#> C196 C197 C198 C199 C200
#> 1 0.0273 -0.00461 0.150 -0.628 0.379
#> 2 -0.5469 -0.09958 -0.492 -0.239 -0.595
#> 3 0.3862 -0.48212 0.444 0.135 0.178
#>
#> [3 rows x 200 columns]
h2o.biases(iris_dl, vector_id=1)
#> C1
#> 1 0.490
#> 2 0.498
#> 3 0.492
#> 4 0.493
#> 5 0.498
#> 6 0.496
#>
#> [200 rows x 1 column]
h2o.biases(iris_dl, vector_id=2)
#> C1
#> 1 1.003
#> 2 1.003
#> 3 1.002
#> 4 1.003
#> 5 0.994
#> 6 0.996
#>
#> [200 rows x 1 column]
h2o.biases(iris_dl, vector_id=3)
#> C1
#> 1 1.33e-03
#> 2 -6.48e-05
#> 3 -2.62e-03
#>
#> [3 rows x 1 column]
#plot weights connecting `Sepal.Length` to first hidden neurons
plot(as.data.frame(h2o.weights(iris_dl, matrix_id=1))[,1])

```

```
as.data.frame(h2o.weights(iris_dl, matrix_id = 1))[,
```



40.8.11 Reproducibility

Every run of DeepLearning results in different results since multithreading is done via Hogwild! that benefits from intentional lock-free race conditions between threads. To get reproducible results for small datasets and testing purposes, set `reproducible=T` and set `seed=1337` (pick any integer). This will not work for big data for technical reasons, and is probably also not desired because of the significant slowdown (runs on 1 core only).

40.8.12 Scoring on Training/Validation Sets During Training

The training and/or validation set errors can be based on a subset of the training or validation data, depending on the values for `score_validation_samples` (defaults to 0: all) or `score_training_samples` (defaults to 10,000 rows, since the training error is only used for early stopping and monitoring). For large datasets, Deep Learning can automatically sample the validation set to avoid spending too much time in scoring during training, especially since scoring results are not currently displayed in the model returned to R.

Note that the default value of `score_duty_cycle=0.1` limits the amount of time spent in scoring to 10%, so a large number of scoring samples won't slow down overall training progress too much, but it will always score once after the first MapReduce iteration, and once at the end of training.

Stratified sampling of the validation dataset can help with scoring on datasets with class imbalance. Note that this option also requires `balance_classes` to be enabled (used to over/under-sample the training dataset, based on the max. relative size of the resulting training dataset, `max_after_balance_size`):

More information can be found in the H2O Deep Learning booklet, in our H2O SlideShare Presentations, our H2O YouTube channel, as well as on our H2O Github Repository, especially in our H2O Deep Learning R tests, and H2O Deep Learning Python tests.

40.9 All done, shutdown H2O

```
h2o.shutdown(prompt=FALSE)
#> [1] TRUE
```

Chapter 41

Regression with ANN - Yacht Hydrodynamics

41.1 Introduction

Regression ANNs predict an output variable as a function of the inputs. The input features (independent variables) can be categorical or numeric types, however, for regression ANNs, we require a numeric dependent variable. If the output variable is a categorical variable (or binary) the ANN will function as a classifier (see next tutorial).

Source: http://uc-r.github.io/ann_regression

In this tutorial we introduce a neural network used for numeric predictions and cover:

- Replication requirements: What you'll need to reproduce the analysis in this tutorial.
- Data Preparation: Preparing our data.
- 1st Regression ANN: Constructing a 1-hidden layer ANN with 1 neuron.
- Regression Hyperparameters: Tuning the model.
- Wrapping Up: Final comments and some exercises to test your skills.

41.2 Replication Requirements

We require the following packages for the analysis.

```
library(tidyverse)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method           from
#>   [.quosures     rlang
#>   c.quosures     rlang
#>   print.quosures rlang
#> Registered S3 method overwritten by 'rvest':
#>   method           from
#>   read_xml.response xml2
#> -- Attaching packages ----- tidyverse 1.2.1 --
#> #> v ggplot2 3.1.1      v purrrr  0.3.2
#> #> v tibble  2.1.1      v dplyr    0.8.0.1
#> #> v tidyverse 0.8.3    v stringr  1.4.0
#> #> v readr   1.3.1      v forcats 0.4.0
```

```
#> -- Conflicts ----- tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
library(neuralnet)
#>
#> Attaching package: 'neuralnet'
#> The following object is masked from 'package:dplyr':
#>
#>     compute
library(GGally)
#> Registered S3 method overwritten by 'GGally':
#>   method from
#>   +.gg   ggplot2
#>
#> Attaching package: 'GGally'
#> The following object is masked from 'package:dplyr':
#>
#>     nasa
```

41.3 Data Preparation

Our regression ANN will use the **Yacht Hydrodynamics** data set from UCI's Machine Learning Repository. The yacht data was provided by Dr. Roberto Lopez email. This data set contains data contains results from 308 full-scale experiments performed at the Delft Ship Hydromechanics Laboratory where they test 22 different hull forms. Their experiment tested the effect of variations in the hull geometry and the ship's Froude number on the craft's residuary resistance per unit weight of displacement.

To begin we download the data from UCI.

```
url <- 'http://archive.ics.uci.edu/ml/machine-learning-databases/00243/yacht_hydrodynamics.data'

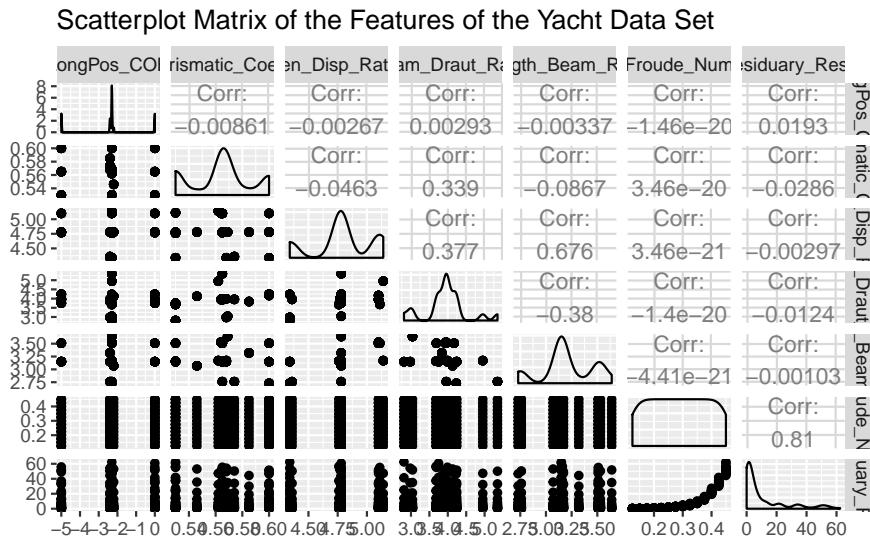
Yacht_Data <- read_table(file = url,
                         col_names = c('LongPos_COB', 'Prismatic_Coeff',
                                      'Len_Displ_Ratio', 'Beam_Draut_Ratio',
                                      'Length_Beam_Ratio', 'Froude_Num',
                                      'Residuary_Resist')) %>%
  na.omit()

#> Parsed with column specification:
#> cols(
#>   LongPos_COB = col_double(),
#>   Prismatic_Coeff = col_double(),
#>   Len_Displ_Ratio = col_double(),
#>   Beam_Draut_Ratio = col_double(),
#>   Length_Beam_Ratio = col_double(),
#>   Froude_Num = col_double(),
#>   Residuary_Resist = col_double()
#> )

dplyr::glimpse(Yacht_Data)
#> Observations: 308
#> Variables: 7
#> $ LongPos_COB      <dbl> -2.3, -2.3, -2.3, -2.3, -2.3, -2.3, -2...
#> $ Prismatic_Coeff <dbl> 0.568, 0.568, 0.568, 0.568, 0.568, 0.568, 0....
```

Prior to any data analysis lets take a look at the data set.

```
ggpairs(Yacht_Data, title = "Scatterplot Matrix of the Features of the Yacht Data Set")
```



Here we see an excellent summary of the variation of each feature in our data set. Draw your attention to the bottom-most strip of scatter-plots. This shows the residuary resistance as a function of the other data set features (independent experimental values). The greatest variation appears with the Froude Number feature. It will be interesting to see how this pattern appears in the subsequent regression ANNs.

Prior to regression ANN construction we first must split the Yacht data set into test and training data sets. Before we split, first scale each feature to fall in the $[0, 1]$ interval.

```
# Scale the Data
scale01 <- function(x){
  (x - min(x)) / (max(x) - min(x))
}

Yacht_Data <- Yacht_Data %>%
  mutate_all(scale01)

# Split into test and train sets
set.seed(12345)
Yacht_Data_Train <- sample_frac(tbl = Yacht_Data, replace = FALSE, size = 0.80)
Yacht_Data_Test <- anti_join(Yacht_Data, Yacht_Data_Train)
#> Joining, by = c("LongPos_COB", "Prismatic_Coeff", "Len_Displ_Ratio", "Beam_Draut_Ratio", "Length_Beam")
```

The `scale01()` function maps each data observation onto the `[0,1]` interval as called in the `dplyr` `mutate_all()` function. We then provided a seed for reproducible results and randomly extracted (without replacement) 80% of the observations to build the `Yacht_Data_Train` data set. Using `dplyr`'s `anti_join()` function we extracted all the observations not within the `Yacht Data Train` data set as our test data set.

in `Yacht_Data_Test`.

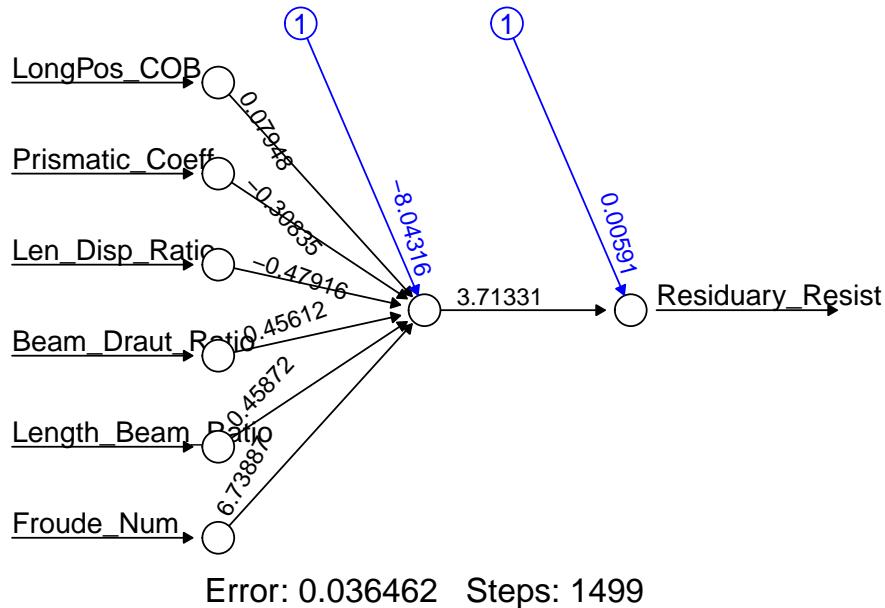
41.4 1st Regression ANN

To begin we construct a 1-hidden layer ANN with 1 neuron, the simplest of all neural networks.

```
set.seed(12321)
Yacht_NN1 <- neuralnet(Residuary_Resist ~ LongPos_COB + Prismatic_Coeff +
                        Len_Disp_Ratio + Beam_Draut_Ratio + Length_Beam_Ratio +
                        Froude_Num, data = Yacht_Data_Train)
```

The `Yacht_NN1` is a list containing all parameters of the regression ANN as well as the results of the neural network on the test data set. To view a diagram of the `Yacht_NN1` use the `plot()` function.

```
plot(Yacht_NN1, rep = 'best')
```



This plot shows the weights learned by the `Yacht_NN1` neural network, and displays the number of iterations before convergence, as well as the SSE of the training data set. To manually compute the SSE you can use the following:

```
NN1_Train_SSE <- sum((Yacht_NN1$net.result - Yacht_Data_Train[, 7])^2)/2
paste("SSE: ", round(NN1_Train_SSE, 4))
#> [1] "SSE: 0.0365"
## [1] "SSE: 0.0361"
```

This SSE is the error associated with the training data set. A superior metric for estimating the generalization capability of the ANN would be the SSE of the test data set. Recall, the test data set contains observations not used to train the `Yacht_NN1` ANN. To calculate the test error, we first must run our test observations through the `Yacht_NN1` ANN. This is accomplished with the `neuralnet` package `compute()` function, which takes as its first input the desired neural network object created by the `neuralnet()` function, and the second argument the test data set feature (independent variable(s)) values.

```
Test_NN1_Output <- compute(Yacht_NN1, Yacht_Data_Test[, 1:6])$net.result
NN1_Test_SSE <- sum((Test_NN1_Output - Yacht_Data_Test[, 7])^2)/2
NN1_Test_SSE
```

```
#> [1] 0.0139
## [1] 0.008417631461
```

The `compute()` function outputs the response variable, in our case the `Residuary_Resist`, as estimated by the neural network. Once we have the ANN estimated response we can compute the test SSE. Comparing the test error of 0.0084 to the training error of 0.0361 we see that in our case our test error is smaller than our training error.

41.5 Regression Hyperparameters

We have constructed the most basic of regression ANNs without modifying any of the default hyperparameters associated with the `neuralnet()` function. We should try and improve the network by modifying its basic structure and hyperparameter modification. To begin we will add depth to the hidden layer of the network, then we will change the activation function from the logistic to the tangent hyperbolicus (`tanh`) to determine if these modifications can improve the test data set SSE. When using the `tanh` activation function, we first must rescale the data from $[0, 1]$ to $[-1, 1]$ using the `rescale` package. For the purposes of this exercise we will use the same random seed for reproducible results, generally this is not a best practice.

```
# 2-Hidden Layers, Layer-1 4-neurons, Layer-2, 1-neuron, logistic activation
# function
set.seed(12321)
Yacht_NN2 <- neuralnet(Residuary_Resist ~ LongPos_COB + Prismatic_Coeff + Len_Disp_Ratio + Beam_Draut_Ra
                        data = Yacht_Data_Train,
                        hidden = c(4, 1),
                        act.fct = "logistic")

## Training Error
NN2_Train_SSE <- sum((Yacht_NN2$net.result - Yacht_Data_Train[, 7])^2)/2

## Test Error
Test_NN2_Output <- compute(Yacht_NN2, Yacht_Data_Test[, 1:6])$net.result
NN2_Test_SSE <- sum((Test_NN2_Output - Yacht_Data_Test[, 7])^2)/2

# Rescale for tanh activation function
scale11 <- function(x) {
  (2 * ((x - min(x))/(max(x) - min(x)))) - 1
}
Yacht_Data_Train <- Yacht_Data_Train %>% mutate_all(scale11)
Yacht_Data_Test <- Yacht_Data_Test %>% mutate_all(scale11)

# 2-Hidden Layers, Layer-1 4-neurons, Layer-2, 1-neuron, tanh activation
# function
set.seed(12321)
Yacht_NN3 <- neuralnet(Residuary_Resist ~ LongPos_COB + Prismatic_Coeff + Len_Disp_Ratio + Beam_Draut_Ra
                        data = Yacht_Data_Train,
                        hidden = c(4, 1),
                        act.fct = "tanh")

## Training Error
NN3_Train_SSE <- sum((Yacht_NN3$net.result - Yacht_Data_Train[, 7])^2)/2

## Test Error
Test_NN3_Output <- compute(Yacht_NN3, Yacht_Data_Test[, 1:6])$net.result
```

```

NN3_Test_SSE <- sum((Test_NN3_Output - Yacht_Data_Test[, 7])^2)/2

# 1-Hidden Layer, 1-neuron, tanh activation function
set.seed(12321)
Yacht_NN4 <- neuralnet(Residuary_Resist ~ LongPos_COB + Prismatic_Coeff + Len_Displ_Ratio + Beam_Draut_Ra
                         data = Yacht_Data_Train,
                         act.fct = "tanh")

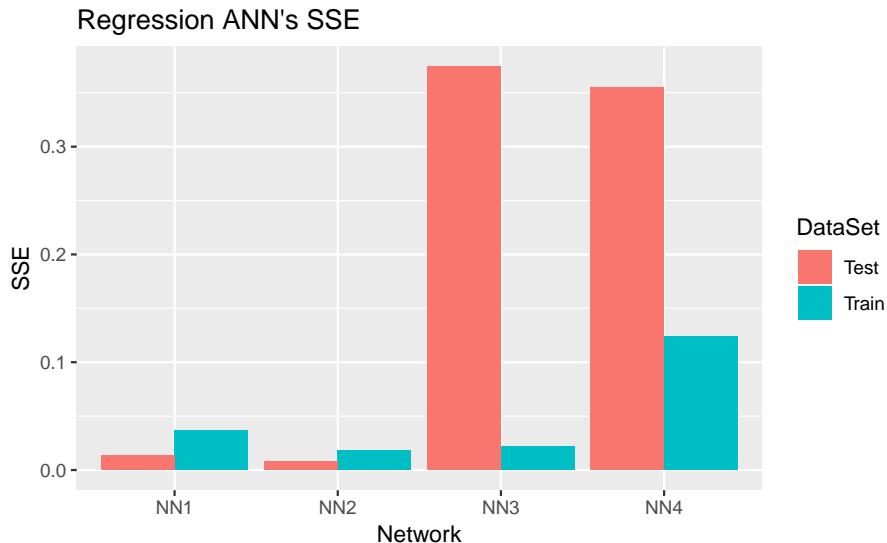
## Training Error
NN4_Train_SSE <- sum((Yacht_NN4$net.result - Yacht_Data_Train[, 7])^2)/2

## Test Error
Test_NN4_Output <- compute(Yacht_NN4, Yacht_Data_Test[, 1:6])$net.result
NN4_Test_SSE <- sum((Test_NN4_Output - Yacht_Data_Test[, 7])^2)/2

# Bar plot of results
Regression_NN_Errors <- tibble(Network = rep(c("NN1", "NN2", "NN3", "NN4"), each = 2),
                                  DataSet = rep(c("Train", "Test"), time = 4),
                                  SSE = c(NN1_Train_SSE, NN1_Test_SSE,
                                          NN2_Train_SSE, NN2_Test_SSE,
                                          NN3_Train_SSE, NN3_Test_SSE,
                                          NN4_Train_SSE, NN4_Test_SSE))

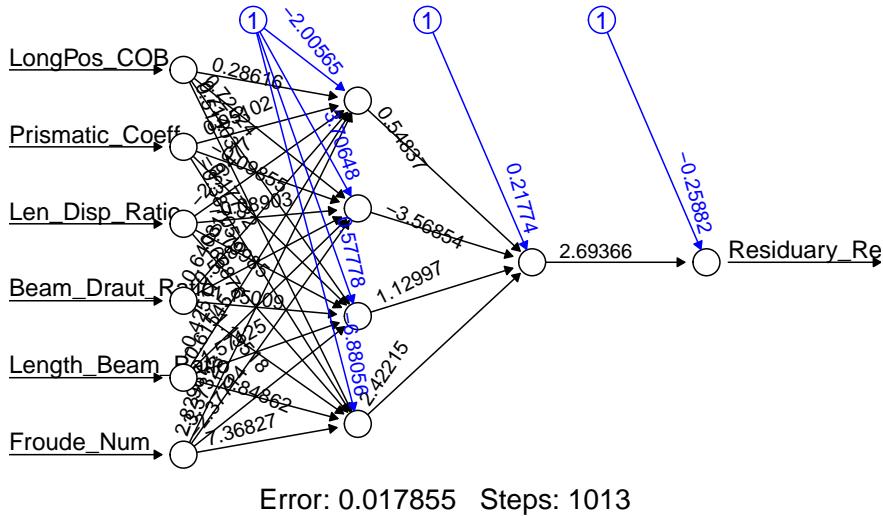
Regression_NN_Errors %>%
  ggplot(aes(Network, SSE, fill = DataSet)) +
  geom_col(position = "dodge") +
  ggtitle("Regression ANN's SSE")

```



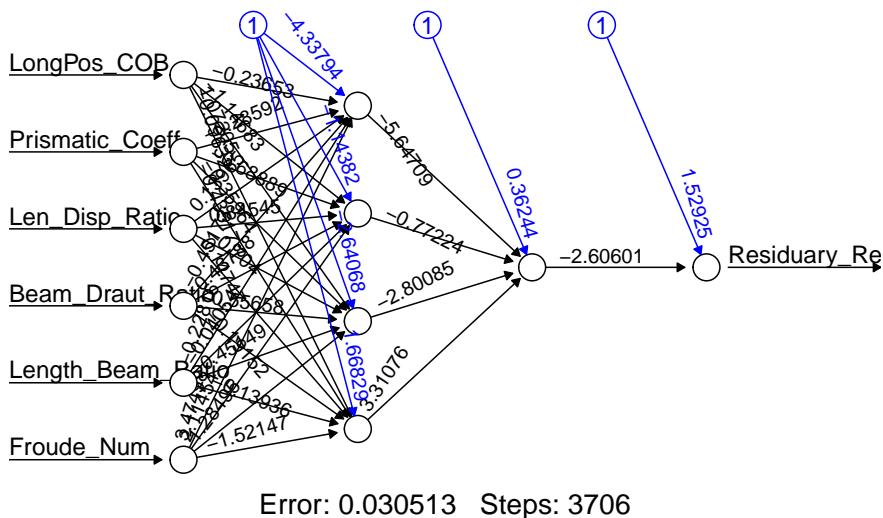
As evident from the plot, we see that the best regression ANN we found was `Yacht_NN2` with a training and test SSE of 0.0188 and 0.0057. We make this determination by the value of the training and test SSEs only. `Yacht_NN2`'s structure is presented here:

```
plot(Yacht_NN2, rep = "best")
```



```
set.seed(12321)
Yacht_NN2 <- neuralnet(Residuary_Resist ~ LongPos_COB + Prismatic_Coeff + Len_Disp_Ratio + Beam_Draut_R,
                        data = Yacht_Data_Train,
                        hidden = c(4, 1),
                        act.fct = "logistic",
                        rep = 10)

plot(Yacht_NN2, rep = "best")
```



By setting the same seed, prior to running the 10 repetitions of ANNs, we force the software to reproduce the exact same Yacht_NN2 ANN for the first replication. The subsequent 9 generated ANNs, use a different random set of starting weights. Comparing the ‘best’ of the 10 repetitions, to the Yacht_NN2, we observe a decrease in training set error indicating we have a superior set of weights.

41.6 Wrapping Up

We have briefly covered regression ANNs in this tutorial. In the next tutorial we will cover classification ANNs. The neuralnet package used in this tutorial is one of many tools available for ANN implementation in R. Others include:

- nnet
- autoencoder
- caret
- RSNNS
- h2o

Before you move on to the next tutorial, test your new knowledge on the exercises that follow.

1. Why do we split the yacht data into a training and test data sets?
2. Re-load the Yacht Data from the UCI Machine learning repository yacht data without scaling. Run any regression ANN. What happens? Why do you think this happens?
3. After completing exercise question 1, re-scale the yacht data. Perform a simple linear regression fitting `Residuary_Resist` as a function of all other features. Now run a regression neural network (see 1st Regression ANN section). Plot the regression ANN and compare the weights on the features in the ANN to the p-values for the regressors.
4. Build your own regression ANN using the scaled yacht data modifying one hyperparameter. Use `?neuralnet` to see the function options. Plot your ANN.

Chapter 42

Regression - cereals dataset

42.1 Introduction

Source: <https://www.analyticsvidhya.com/blog/2017/09/creating-visualizing-neural-network-in-r/>

Neural network is an information-processing machine and can be viewed as analogous to human nervous system. Just like human nervous system, which is made up of interconnected neurons, a neural network is made up of interconnected information processing units. The information processing units do not work in a linear manner. In fact, neural network draws its strength from parallel processing of information, which allows it to deal with non-linearity. Neural network becomes handy to infer meaning and detect patterns from complex data sets.

Neural network is considered as one of the most useful technique in the world of data analytics. However, it is complex and is often regarded as a black box, i.e. users view the input and output of a neural network but remain clueless about the knowledge generating process. We hope that the article will help readers learn about the internal mechanism of a neural network and get hands-on experience to implement it in R.

42.2 The Basics of Neural Networks

A neural network is a model characterized by an activation function, which is used by interconnected information processing units to transform input into output. A neural network has always been compared to human nervous system. Information is passed through interconnected units analogous to information passage through neurons in humans. The first layer of the neural network receives the raw input, processes it and passes the processed information to the hidden layers. The hidden layer passes the information to the last layer, which produces the output. The advantage of neural network is that it is adaptive in nature. It learns from the information provided, i.e. trains itself from the data, which has a known outcome and optimizes its weights for a better prediction in situations with unknown outcome.

A perceptron, viz. single layer neural network, is the most basic form of a neural network. A perceptron receives multidimensional input and processes it using a weighted summation and an activation function. It is trained using a labeled data and learning algorithm that optimize the weights in the summation processor. A major limitation of perceptron model is its inability to deal with non-linearity. A multilayered neural network overcomes this limitation and helps solve non-linear problems. The input layer connects with hidden layer, which in turn connects to the output layer. The connections are weighted and weights are optimized using a learning rule.

There are many learning rules that are used with neural network:

- a) least mean square;

- b) gradient descent;
- c) newton's rule;
- d) conjugate gradient etc.

The learning rules can be used in conjunction with backpropagation error method. The learning rule is used to calculate the error at the output unit. This error is backpropagated to all the units such that the error at each unit is proportional to the contribution of that unit towards total error at the output unit. The errors at each unit are then used to optimize the weight at each connection. Figure 1 displays the structure of a simple neural network model for better understanding.

42.3 Fitting a Neural Network in R

Now we will fit a neural network model in R. In this article, we use a subset of cereal dataset shared by Carnegie Mellon University (CMU). The details of the dataset are on the following link: <http://lib.stat.cmu.edu/DASL/Datafiles/Cereals.html>. The objective is to predict rating of the cereals variables such as calories, proteins, fat etc. The R script is provided side by side and is commented for better understanding of the user. . The data is in .csv format and can be downloaded by clicking: cereals.

Please set working directory in R using `setwd()` function, and keep cereal.csv in the working directory. We use rating as the dependent variable and calories, proteins, fat, sodium and fiber as the independent variables. We divide the data into training and test set. Training set is used to find the relationship between dependent and independent variables while the test set assesses the performance of the model. We use 60% of the dataset as training set. The assignment of the data to training and test set is done using random sampling. We perform random sampling on R using `sample()` function. We have used `set.seed()` to generate same random sample everytime and maintain consistency. We will use the index variable while fitting neural network to create training and test data sets. The R script is as follows:

```
## Creating index variable

# Read the Data
data = read.csv(file.path(data_raw_dir, "cereals.csv"), header=T)

# Random sampling
samplesize = 0.60 * nrow(data)
set.seed(80)
index = sample( seq_len ( nrow ( data ) ), size = samplesize )

# Create training and test set
datatrain = data[ index, ]
datatest = data[ -index, ]

dplyr::glimpse(data)
#> Observations: 75
#> Variables: 6
#> $ calories <int> 70, 120, 70, 50, 110, 110, 130, 90, 90, 120, 110, 120...
#> $ protein <int> 4, 3, 4, 4, 2, 2, 3, 2, 3, 1, 6, 1, 3, 1, 2, 2, 1, 1, ...
#> $ fat <int> 1, 5, 1, 0, 2, 0, 2, 1, 0, 2, 2, 3, 2, 1, 0, 0, 0, 1, ...
#> $ sodium <int> 130, 15, 260, 140, 180, 125, 210, 200, 210, 220, 290, ...
#> $ fiber <dbl> 10.0, 2.0, 9.0, 14.0, 1.5, 1.0, 2.0, 4.0, 5.0, 0.0, 2...
#> $ rating <dbl> 68.4, 34.0, 59.4, 93.7, 29.5, 33.2, 37.0, 49.1, 53.3, ...
```

Now we fit a neural network on our data. We use `neuralnet` library for the analysis. The first step is to scale the cereal dataset. The scaling of data is essential because otherwise a variable may have large impact on the prediction variable only because of its scale. Using unscaled may lead to meaningless results. The

common techniques to scale data are: min-max normalization, Z-score normalization, median and MAD, and tan-h estimators. The min-max normalization transforms the data into a common range, thus removing the scaling effect from all the variables. Unlike Z-score normalization and median and MAD method, the min-max method retains the original distribution of the variables. We use min-max normalization to scale the data. The R script for scaling the data is as follows.

```
## Scale data for neural network

max = apply(data , 2 , max)
min = apply(data, 2 , min)
scaled = as.data.frame(scale(data, center = min, scale = max - min))

## Fit neural network

# install library
# install.packages("neuralnet ")

# load library
library(neuralnet)

# creating training and test set
trainNN = scaled[index , ]
testNN = scaled[-index , ]

# fit neural network
set.seed(2)
NN = neuralnet(rating ~ calories + protein + fat + sodium + fiber,
               trainNN, hidden = 3 , linear.output = T )

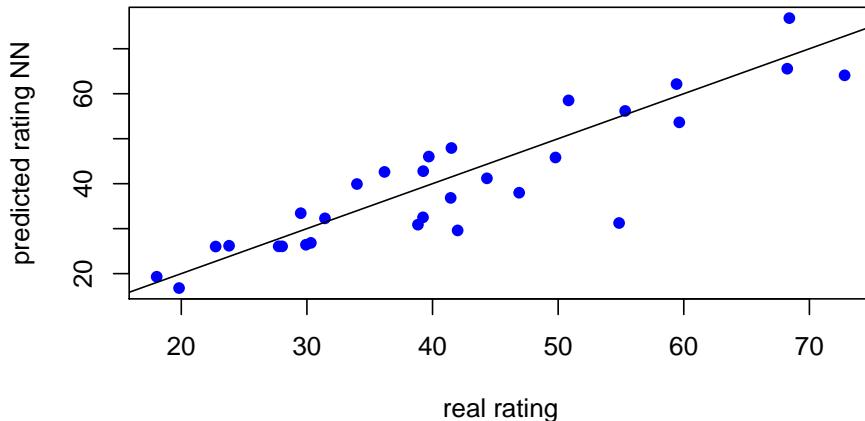
# plot neural network
plot(NN)

## Prediction using neural network

predict_testNN = compute(NN, testNN[,c(1:5)])
predict_testNN = (predict_testNN$net.result * (max(data$rating) - min(data$rating))) + min(data$rating)

plot(datatest$rating, predict_testNN, col='blue', pch=16, ylab = "predicted rating NN", xlab = "real rating")
abline(0,1)

# Calculate Root Mean Square Error (RMSE)
RMSE.NN = (sum((datatest$rating - predict_testNN)^2) / nrow(datatest)) ^ 0.5
```



```
## Cross validation of neural network model

# install relevant libraries
# install.packages("boot")
# install.packages("plyr")

# Load libraries
library(boot)
library(plyr)

# Initialize variables
set.seed(50)
k = 100
RMSE.NN = NULL

List = list( )

# Fit neural network model within nested for loop
for(j in 10:65){
  for (i in 1:k) {
    index = sample(1:nrow(data),j )

    trainNN = scaled[index,]
    testNN = scaled[-index,]
    datatest = data[-index,]

    NN = neuralnet(rating ~ calories + protein + fat + sodium + fiber, trainNN, hidden = 3, linear...
    predict_testNN = compute(NN,testNN[,c(1:5)])
    predict_testNN = (predict_testNN$net.result*(max(data$rating)-min(data$rating)))+min(data$rating)

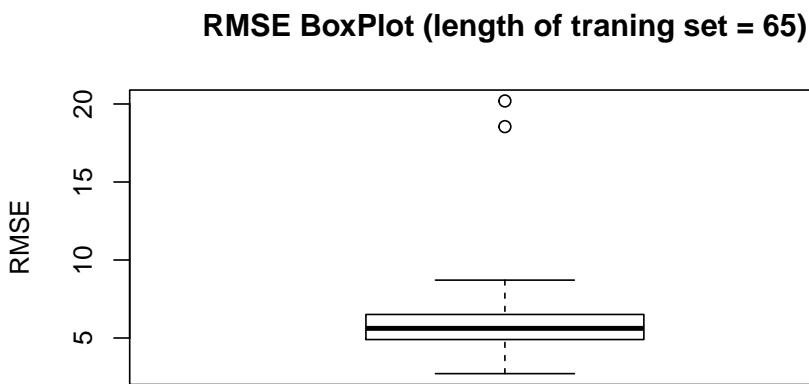
    RMSE.NN [i]<- (sum((datatest$rating - predict_testNN)^2)/nrow(datatest))^0.5
  }
  List[[j]] = RMSE.NN
}

Matrix.RMSE = do.call(cbind, List)

## Prepare boxplot
boxplot(Matrix.RMSE[,56], ylab = "RMSE", main = "RMSE BoxPlot (length of traning set = 65)")
```



Figure 42.1: Variation of RMSE



```

## Variation of median RMSE
# install.packages("matrixStats")
library(matrixStats)
#>
#> Attaching package: 'matrixStats'
#> The following object is masked from 'package:plyr':
#>
#>     count

med = colMedians(Matrix.RMSE)

X = seq(10,65)

plot (med~X, type = "l", xlab = "length of training set", ylab = "median RMSE", main = "Variation of RMSE with length of training set")

```

Figure 42.1) shows that the median RMSE of our model decreases as the length of the training the set. This is an important result. The reader must remember that the model accuracy is dependent on the length of training set. The performance of neural network model is sensitive to training-test split.

42.4 End Notes

The article discusses the theoretical aspects of a neural network, its implementation in R and post training evaluation. Neural network is inspired from biological nervous system. Similar to nervous system the information is passed through layers of processors. The significance of variables is represented by weights of each connection. The article provides basic understanding of back propagation algorithm, which is used to assign these weights. In this article we also implement neural network on R. We use a publically available dataset shared by CMU. The aim is to predict the rating of cereals using information such as calories, fat, protein etc. After constructing the neural network we evaluate the model for accuracy and robustness. We compute RMSE and perform cross-validation analysis. In cross validation, we check the variation in model accuracy as the length of training set is changed. We consider training sets with length 10 to 65. For each length a 100 samples are random picked and median RMSE is calculated. We show that model accuracy increases when training set is large. Before using the model for prediction, it is important to check the robustness of performance through cross validation.

The article provides a quick review neural network and is a useful reference for data enthusiasts. We have provided commented R code throughout the article to help readers with hands on experience of using neural networks.

Chapter 43

Fitting a neural network

43.1 Introduction

<https://www.r-bloggers.com/fitting-a-neural-network-in-r-neuralnet-package/>

<https://datascienceplus.com/fitting-neural-network-in-r/>

Neural networks have always been one of the fascinating machine learning models in my opinion, not only because of the fancy backpropagation algorithm but also because of their complexity (think of deep learning with many hidden layers) and structure inspired by the brain.

Neural networks have not always been popular, partly because they were, and still are in some cases, computationally expensive and partly because they did not seem to yield better results when compared with simpler methods such as support vector machines (SVMs). Nevertheless, Neural Networks have, once again, raised attention and become popular.

Update: We published another post about Network analysis at DataScience+ Network analysis of Game of Thrones

In this post, we are going to fit a simple neural network using the neuralnet package and fit a linear model as a comparison.

43.2 The dataset

We are going to use the Boston dataset in the MASS package. The Boston dataset is a collection of data about housing values in the suburbs of Boston. Our goal is to predict the median value of owner-occupied homes (medv) using all the other continuous variables available.

```
set.seed(500)
library(MASS)
data <- Boston

dplyr::glimpse(data)
#> Observations: 506
#> Variables: 14
#> $ crim    <dbl> 0.00632, 0.02731, 0.02729, 0.03237, 0.06905, 0.02985, ...
#> $ zn      <dbl> 18.0, 0.0, 0.0, 0.0, 0.0, 12.5, 12.5, 12.5, 12.5, ...
#> $ indus   <dbl> 2.31, 7.07, 7.07, 2.18, 2.18, 2.18, 7.87, 7.87, 7.87, ...
#> $ chas    <int> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
#> $ nox     <dbl> 0.538, 0.469, 0.469, 0.458, 0.458, 0.458, 0.524, 0.524, ...
```

```
#> $ rm      <dbl> 6.58, 6.42, 7.18, 7.00, 7.15, 6.43, 6.01, 6.17, 5.63, ...
#> $ age     <dbl> 65.2, 78.9, 61.1, 45.8, 54.2, 58.7, 66.6, 96.1, 100.0, ...
#> $ dis     <dbl> 4.09, 4.97, 4.97, 6.06, 6.06, 6.06, 5.56, 5.95, 6.08, ...
#> $ rad     <int> 1, 2, 2, 3, 3, 3, 5, 5, 5, 5, 5, 5, 4, 4, 4, 4, ...
#> $ tax     <dbl> 296, 242, 242, 222, 222, 311, 311, 311, 311, ...
#> $ ptratio <dbl> 15.3, 17.8, 17.8, 18.7, 18.7, 18.7, 15.2, 15.2, 15.2, ...
#> $ black   <dbl> 397, 397, 393, 395, 397, 394, 396, 397, 387, 387, 393, ...
#> $ lstat   <dbl> 4.98, 9.14, 4.03, 2.94, 5.33, 5.21, 12.43, 19.15, 29.9...
#> $ medv   <dbl> 24.0, 21.6, 34.7, 33.4, 36.2, 28.7, 22.9, 27.1, 16.5, ...
```

First we need to check that no datapoint is missing, otherwise we need to fix the dataset.

```
apply(data,2,function(x) sum(is.na(x)))
#>    crim      zn    indus    chas      nox      rm     age     dis     rad
#>      0        0      0        0        0        0        0        0        0
#>    tax  ptratio    black    lstat    medv
#>      0        0      0        0        0
```

There is no missing data, good. We proceed by randomly splitting the data into a train and a test set, then we fit a linear regression model and test it on the test set. Note that I am using the `gml()` function instead of the `lm()` this will become useful later when cross validating the linear model.

```
index <- sample(1:nrow(data),round(0.75*nrow(data)))
train <- data[index,]
test <- data[-index,]
lm.fit <- gml(medv~., data=train)
summary(lm.fit)
#>
#> Call:
#> gml(formula = medv ~ ., data = train)
#>
#> Deviance Residuals:
#>    Min      1Q  Median      3Q     Max
#> -15.211  -2.559  -0.655   1.828  29.711
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)    
#> (Intercept) 31.11170  5.45981   5.70  2.5e-08 ***
#> crim        -0.11137  0.03326  -3.35  0.00090 ***
#> zn          0.04263  0.01431   2.98  0.00308 ** 
#> indus       0.00148  0.06745   0.02  0.98247    
#> chas        1.75684  0.98109   1.79  0.07417 .  
#> nox        -18.18485 4.47157  -4.07  5.8e-05 ***
#> rm          4.76034  0.48047   9.91 < 2e-16 ***
#> age        -0.01344  0.01410  -0.95  0.34119    
#> dis        -1.55375  0.21893  -7.10  6.7e-12 ***
#> rad         0.28818  0.07202   4.00  7.6e-05 ***
#> tax        -0.01374  0.00406  -3.38  0.00079 ***
#> ptratio     -0.94755  0.14012  -6.76  5.4e-11 ***
#> black       0.00950  0.00290   3.28  0.00115 ** 
#> lstat      -0.38890  0.05973  -6.51  2.5e-10 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> (Dispersion parameter for gaussian family taken to be 20.2)
```

```
#>
#>     Null deviance: 32463.5  on 379  degrees of freedom
#> Residual deviance:  7407.1  on 366  degrees of freedom
#> AIC: 2237
#>
#> Number of Fisher Scoring iterations: 2
pr.lm <- predict(lm.fit,test)
MSE.lm <- sum((pr.lm - test$medv)^2)/nrow(test)
```

The sample(x,size) function simply outputs a vector of the specified size of randomly selected samples from the vector x. By default the sampling is without replacement: index is essentially a random vector of indeces. Since we are dealing with a regression problem, we are going to use the mean squared error (MSE) as a measure of how much our predictions are far away from the real data.

43.3 Preparing to fit the neural network

Before fitting a neural network, some preparation need to be done. Neural networks are not that easy to train and tune.

As a first step, we are going to address data preprocessing. It is good practice to normalize your data before training a neural network. I cannot emphasize enough how important this step is: depending on your dataset, avoiding normalization may lead to useless results or to a very difficult training process (most of the times the algorithm will not converge before the number of maximum iterations allowed). You can choose different methods to scale the data (z-normalization, min-max scale, etc...). I chose to use the min-max method and scale the data in the interval [0,1]. Usually scaling in the intervals [0,1] or [-1,1] tends to give better results. We therefore scale and split the data before moving on:

```
maxs <- apply(data, 2, max)
mins <- apply(data, 2, min)

scaled <- as.data.frame(scale(data, center = mins, scale = maxs - mins))

train_ <- scaled[index,]
test_ <- scaled[-index,]
```

Note that scale returns a matrix that needs to be coerced into a data.frame.

43.4 Parameters

As far as I know there is no fixed rule as to how many layers and neurons to use although there are several more or less accepted rules of thumb. Usually, if at all necessary, one hidden layer is enough for a vast numbers of applications. As far as the number of neurons is concerned, it should be between the input layer size and the output layer size, usually 2/3 of the input size. At least in my brief experience testing again and again is the best solution since there is no guarantee that any of these rules will fit your model best. Since this is a toy example, we are going to use 2 hidden layers with this configuration: 13:5:3:1. The input layer has 13 inputs, the two hidden layers have 5 and 3 neurons and the output layer has, of course, a single output since we are doing regression. Let's fit the net:

```
library(neuralnet)
n <- names(train_)
f <- as.formula(paste("medv ~", paste(n[!n %in% "medv"], collapse = " + ")))
nn <- neuralnet(f,data=train_,hidden=c(5,3),linear.output=T)
```

A couple of notes:

- For some reason the formula $y\sim.$ is not accepted in the `neuralnet()` function. You need to first write the formula and then pass it as an argument in the fitting function.
- The `hidden` argument accepts a vector with the number of neurons for each hidden layer, while the argument `linear.output` is used to specify whether we want to do regression `linear.output=TRUE` or classification `linear.output=FALSE`

The `neuralnet` package provides a nice tool to plot the model:

This is the graphical representation of the model with the weights on each connection:

```
plot(nn)
```

The black lines show the connections between each layer and the weights on each connection while the blue lines show the bias term added in each step. The bias can be thought as the intercept of a linear model. The net is essentially a black box so we cannot say that much about the fitting, the weights and the model. Suffice to say that the training algorithm has converged and therefore the model is ready to be used.

43.5 Predicting medv using the neural network

Now we can try to predict the values for the test set and calculate the MSE. Remember that the net will output a normalized prediction, so we need to scale it back in order to make a meaningful comparison (or just a simple prediction).

```
pr.nn <- compute(nn,test_[,1:13])

pr.nn_ <- pr.nn$net.result*(max(data$medv)-min(data$medv))+min(data$medv)
test.r <- (test_$medv)*(max(data$medv)-min(data$medv))+min(data$medv)

MSE.nn <- sum((test.r - pr.nn_)^2)/nrow(test_)
```

we then compare the two MSEs

```
print(paste(MSE.lm,MSE.nn))
#> [1] "31.2630222372615 16.4595537665717"
```

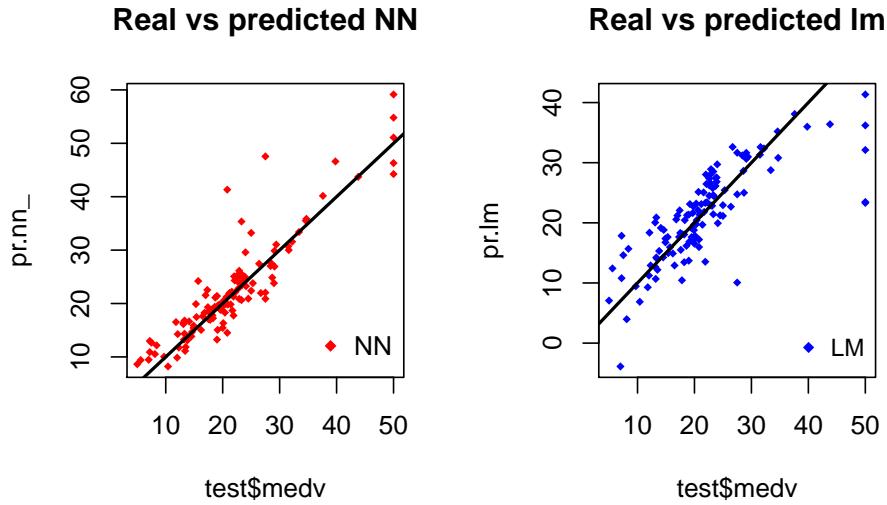
Apparently, the net is doing a better work than the linear model at predicting medv. Once again, be careful because this result depends on the train-test split performed above. Below, after the visual plot, we are going to perform a fast cross validation in order to be more confident about the results.

A first visual approach to the performance of the network and the linear model on the test set is plotted below

```
par(mfrow=c(1,2))

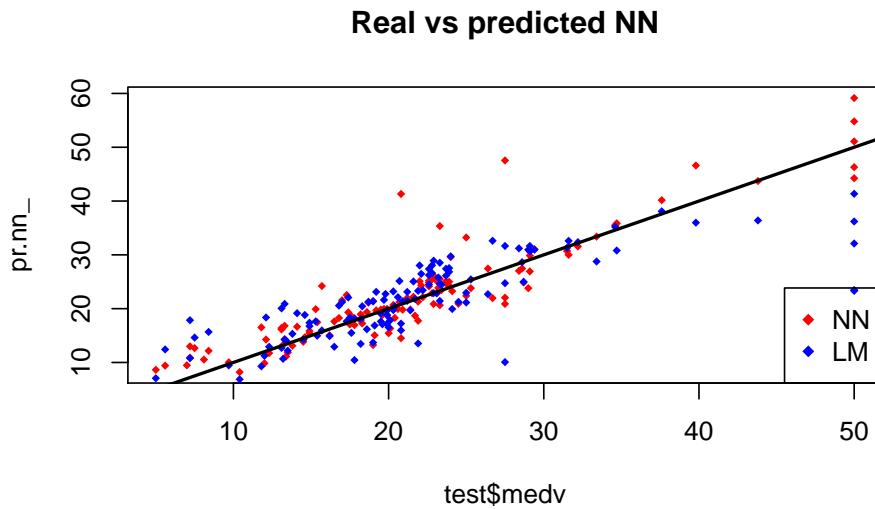
plot(test$medv,pr.nn_,col='red',main='Real vs predicted NN',pch=18,cex=0.7)
abline(0,1,lwd=2)
legend('bottomright',legend='NN',pch=18,col='red', bty='n')

plot(test$medv,pr.lm,col='blue',main='Real vs predicted lm',pch=18, cex=0.7)
abline(0,1,lwd=2)
legend('bottomright',legend='LM',pch=18,col='blue', bty='n', cex=.95)
```



By visually inspecting the plot we can see that the predictions made by the neural network are (in general) more concentrated around the line (a perfect alignment with the line would indicate a MSE of 0 and thus an ideal perfect prediction) than those made by the linear model.

```
plot(test$medv,pr.nn_,col='red',main='Real vs predicted NN',pch=18,cex=0.7)
points(test$medv,pr.lm,col='blue',pch=18,cex=0.7)
abline(0,1,lwd=2)
legend('bottomright',legend=c('NN','LM'),pch=18,col=c('red','blue'))
```



43.6 A (fast) cross validation

Cross validation is another very important step of building predictive models. While there are different kind of cross validation methods, the basic idea is repeating the following process a number of time:

train-test split

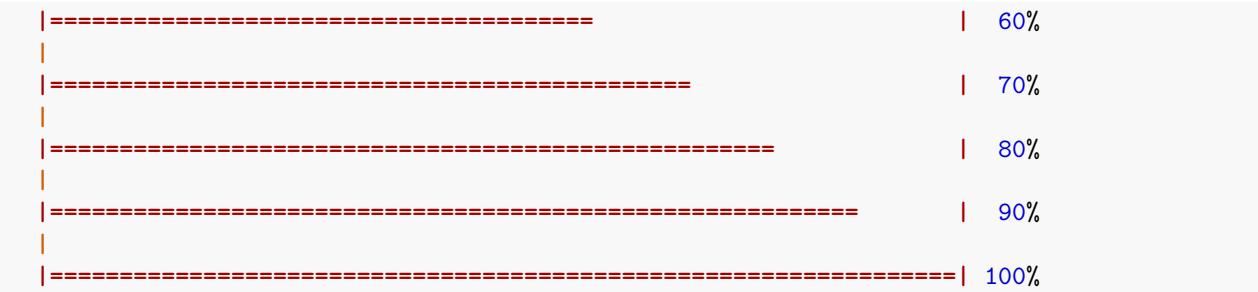
- Do the train-test split
- Fit the model to the train set
- Test the model on the test set
- Calculate the prediction error
- Repeat the process K times

Then by calculating the average error we can get a grasp of how the model is doing.

We are going to implement a fast cross validation using a for loop for the neural network and the `cv.glm()` function in the `boot` package for the linear model. As far as I know, there is no built-in function in R to perform cross-validation on this kind of neural network, if you do know such a function, please let me know in the comments. Here is the 10 fold cross-validated MSE for the linear model:

```
library(boot)
set.seed(200)
lm.fit <- glm(medv~.,data=data)
cv.glm(data,lm.fit,K=10)$delta[1]
#> [1] 23.2
```

Now the net. Note that I am splitting the data in this way: 90% train set and 10% test set in a random way for 10 times. I am also initializing a progress bar using the `plyr` library because I want to keep an eye on the status of the process since the fitting of the neural network may take a while.



After a while, the process is done, we calculate the average MSE and plot the results as a boxplot

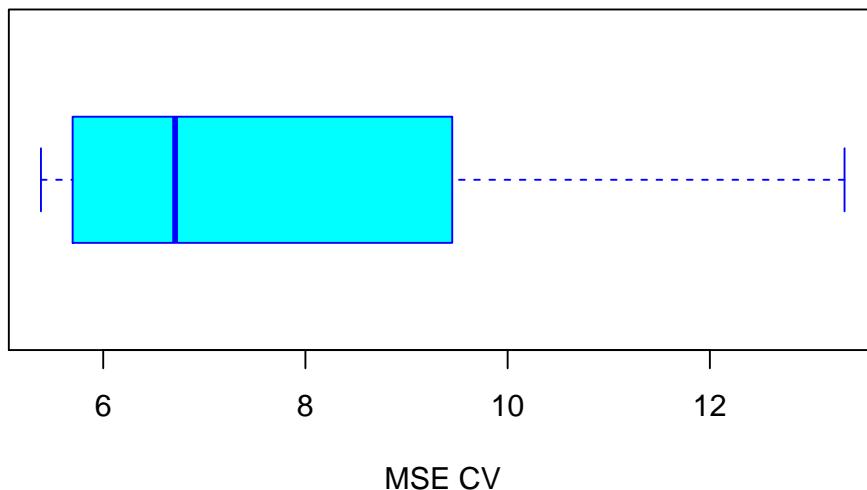
```
mean(cv.error)
#> [1] 7.64

cv.error
#> [1] 13.33 7.10 6.58 5.70 6.84 5.77 10.75 5.38 9.45 5.50
```

The code for the box plot: The code above outputs the following boxplot:

```
boxplot(cv.error,xlab='MSE CV',col='cyan',
        border='blue',names='CV error (MSE)',
        main='CV error (MSE) for NN',horizontal=TRUE)
```

CV error (MSE) for NN



As you can see, the average MSE for the neural network (10.33) is lower than the one of the linear model although there seems to be a certain degree of variation in the MSEs of the cross validation. This may depend on the splitting of the data or the random initialization of the weights in the net. By running the simulation different times with different seeds you can get a more precise point estimate for the average MSE.

43.7 A final note on model interpretability

Neural networks resemble black boxes a lot: explaining their outcome is much more difficult than explaining the outcome of simpler model such as a linear model. Therefore, depending on the kind of application you need, you might want to take into account this factor too. Furthermore, as you have seen above, extra care is needed to fit a neural network and small changes can lead to different results.

A gist with the full code for this post can be found [here](#).

Thank you for reading this post, leave a comment below if you have any question.

Chapter 44

Visualization of neural networks

<https://beckmw.wordpress.com/tag/neuralnet/>

In my last post I said I wasn't going to write anymore about neural networks (i.e., multilayer feedforward perceptron, supervised ANN, etc.). That was a lie. I've received several requests to update the neural network plotting function described in the original post. As previously explained, R does not provide a lot of options for visualizing neural networks. The only option I know of is a plotting method for objects from the neuralnet package. This may be my opinion, but I think this plot leaves much to be desired (see below). Also, no plotting methods exist for neural networks created in other packages, i.e., nnet and RSNNS. These packages are the only ones listed on the CRAN task view, so I've updated my original plotting function to work with all three. Additionally, I've added a new option for plotting a raw weight vector to allow use with neural networks created elsewhere. This blog describes these changes, as well as some new arguments added to the original function.

As usual, I'll simulate some data to use for creating the neural networks. The dataset contains eight input variables and two output variables. The final dataset is a data frame with all variables, as well as separate data frames for the input and output variables. I've retained separate datasets based on the syntax for each package.

```
library(clusterGeneration)
#> Loading required package: MASS
library(tictoc)

seed.val<- 12345
set.seed(seed.val)

num.vars<-8
num.obs<-1000

# input variables
cov.mat <-genPositiveDefMat(num.vars,covMethod=c("unifcorrmat"))$Sigma
rand.vars <-mvrnorm(num.obs,rep(0,num.vars),Sigma=cov.mat)

# output variables
parms <-runif(num.vars,-10,10)
y1 <- rand.vars %*% matrix(parms) + rnorm(num.obs, sd=20)
parms2 <- runif(num.vars,-10,10)
y2 <- rand.vars %*% matrix(parms2) + rnorm(num.obs, sd=20)

# final datasets
```

```

rand.vars <- data.frame(rand.vars)
resp <- data.frame(y1,y2)
names(resp) <- c('Y1','Y2')
dat.in <- data.frame(resp, rand.vars)

dplyr::glimpse(dat.in)
#> Observations: 1,000
#> Variables: 10
#> $ Y1 <dbl> 25.442, -14.578, -36.214, 15.216, -6.393, -20.849, -28.665, ...
#> $ Y2 <dbl> 16.9, 38.8, 31.2, -31.2, 93.3, 11.7, 59.7, -103.5, -49.8, 5...
#> $ X1 <dbl> 3.138, -0.705, -4.373, 0.837, 0.787, 1.923, -1.419, 1.121, ...
#> $ X2 <dbl> 0.195, -0.302, 0.773, 1.311, 3.506, 1.245, 3.800, -0.165, 0...
#> $ X3 <dbl> -1.795, -2.596, 2.308, 4.081, -3.921, 1.473, -0.926, 7.101, ...
#> $ X4 <dbl> -2.7216, 3.0589, 1.2455, 3.4607, 2.3775, -2.9833, 2.6669, -...
#> $ X5 <dbl> 0.0407, 0.7602, -3.0217, -4.2799, 2.0859, 1.4765, 0.0561, 2...
#> $ X6 <dbl> -1.4820, -0.5014, 0.0603, -1.8551, 2.2817, 1.7386, 1.7450, ...
#> $ X7 <dbl> -0.7169, -0.3618, -1.5283, 4.2026, -6.1548, -0.3545, -6.028...
#> $ X8 <dbl> 1.152, 1.810, -1.357, 0.598, -1.425, -1.210, -1.004, 2.494, ...

```

The various neural network packages are used to create separate models for plotting.

```

# first model with nnet
#nnet function from nnet package
library(nnet)
set.seed(seed.val)
tic()
mod1 <- nnet(rand.vars, resp, data = dat.in, size = 10, linout = T)
#> # weights: 112
#> initial value 4784162.893260
#> iter 10 value 1794537.980652
#> iter 20 value 1577753.498759
#> iter 30 value 1485254.945755
#> iter 40 value 1449238.248788
#> iter 50 value 1427720.291804
#> iter 60 value 1416977.236373
#> iter 70 value 1405167.753521
#> iter 80 value 1395046.792257
#> iter 90 value 1370522.267277
#> iter 100 value 1363709.540981
#> final value 1363709.540981
#> stopped after 100 iterations
toc()
#> 0.186 sec elapsed

# nn <- neuralnet(form.in,
#                   data = dat.sc,
#                   # hidden = c(13, 10, 3),
#                   hidden = c(5),
#                   act.fct = "tanh",
#                   linear.output = FALSE,
#                   lifesign = "minimal")

# 2nd model with neuralnet
# neuralnet function from neuralnet package, notice use of only one response
library(neuralnet)

```

```

softplus <- function(x) log(1 + exp(x))
sigmoid <- function(x) log(1 + exp(-x))

dat.sc <- scale(dat.in)
form.in <- as.formula('Y1 ~ X1+X2+X3+X4+X5+X6+X7+X8')
set.seed(seed.val)
tic()
mod2 <- neuralnet(form.in, data = dat.sc, hidden = 10, lifesign = "minimal",
                    linear.output = FALSE,
                    act.fct = "tanh")
#> hidden: 10    thresh: 0.01    rep: 1/1    steps: 26361 error: 160.06372    time: 36.42 secs
toc()
#> 36.424 sec elapsed

# third model with RSNNS
# mlp function from RSNNS package
library(RSNNS)
#> Loading required package: Rcpp
set.seed(seed.val)
tic()
mod3 <- mlp(rand.vars, resp, size = 10, linOut = T)
toc()
#> 0.348 sec elapsed

```

I've noticed some differences between the functions that could lead to some confusion. For simplicity, the above code represents my interpretation of the most direct way to create a neural network in each package. Be very aware that direct comparison of results is not advised given that the default arguments differ between the packages. A few key differences are as follows, although many others should be noted. First, the functions differ in the methods for passing the primary input variables.

The `nnet` function can take separate (or combined) x and y inputs as data frames or as a formula, the `neuralnet` function can only use a formula as input, and the `mlp` function can only take a data frame as combined or separate variables as input. As far as I know, the `neuralnet` function is not capable of modelling multiple response variables, unless the response is a categorical variable that uses one node for each outcome. Additionally, the default output for the `neuralnet` function is linear, whereas the opposite is true for the other two functions.

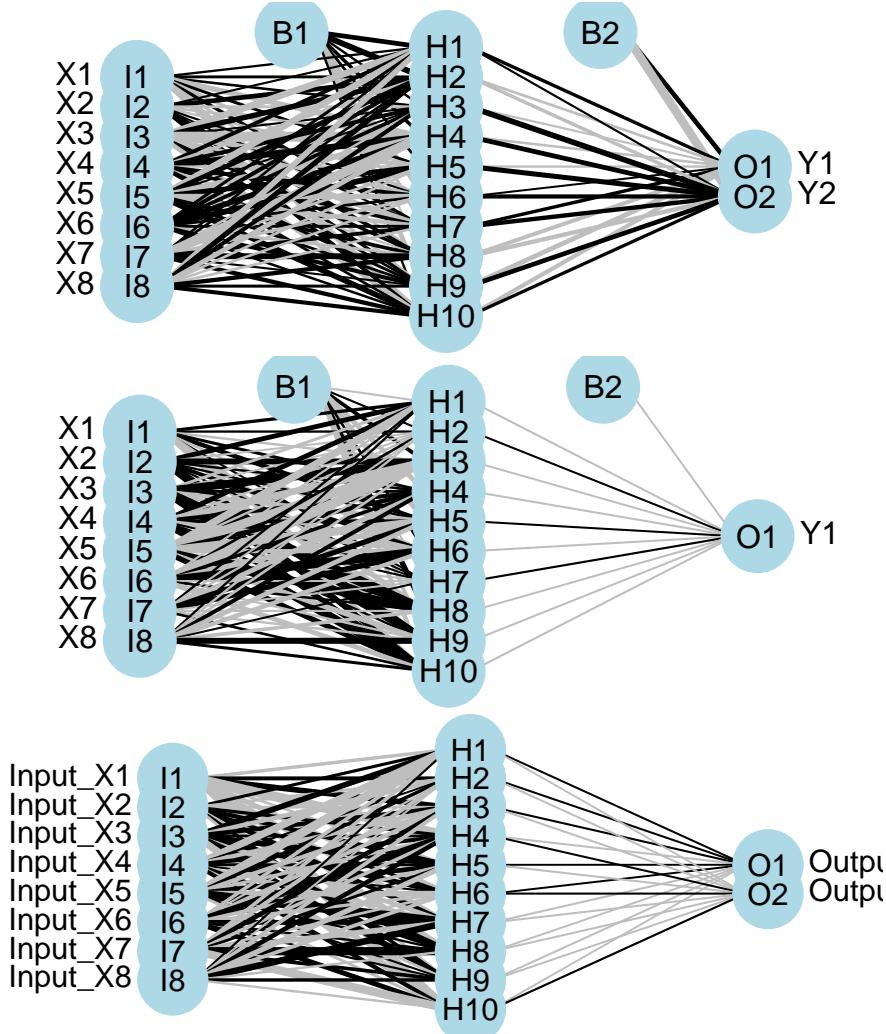
Specifics aside, here's how to use the updated plot function. Note that the same syntax is used to plot each model

```

# import the function from Github
library(devtools)
source_url('https://gist.github.com/fawda123/7471137/raw/466c1474d0a505ff044412703516c34f1a4'
#> SHA-1 hash of file is 74c80bd5ddbc17ab3ae5ece9c0ed9beb612e87ef

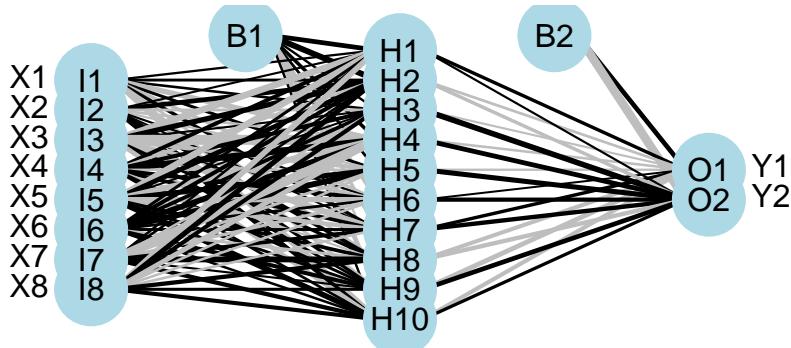
# plot each model
plot.nnet(mod1)
#> Loading required package: scales
#> Loading required package: reshape
plot.nnet(mod2)
plot.nnet(mod3)
#> Warning in plot.nnet(mod3): Bias layer not applicable for rsnns object

```



The plotting function can also now be used with an arbitrary weight vector, rather than a specific model object. The struct argument must also be included if this option is used. I thought the easiest way to use the plotting function with your own weights was to have the input weights as a numeric vector, including bias layers. I've shown how this can be done using the weights directly from mod1 for simplicity.

```
wts.in <- mod1$wts
struct <- mod1$struct
plot.nnet(wts.in, struct=struct)
```



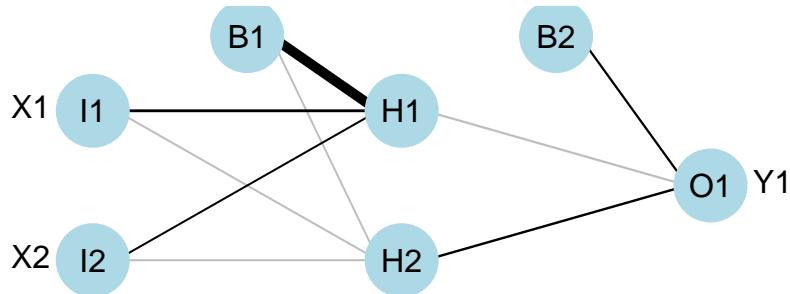
Note that `wts.in` is a numeric vector with length equal to the expected given the architecture (i.e., for 8 10

2 network, 100 connection weights plus 12 bias weights). The plot should look the same as the plot for the neural network from nnet.

The weights in the input vector need to be in a specific order for correct plotting. I realize this is not clear by looking directly at wt.in but this was the simplest approach I could think of. The weight vector shows the weights for each hidden node in sequence, starting with the bias input for each node, then the weights for each output node in sequence, starting with the bias input for each output node. Note that the bias layer has to be included even if the network was not created with biases. If this is the case, simply input a random number where the bias values should go and use the argument bias=F. I'll show the correct order of the weights using an example with plot.nn from the neuralnet package since the weights are included directly on the plot.

If we pretend that the above figure wasn't created in R, we would input the mod.in argument for the updated plotting function as follows. Also note that struct must be included if using this approach.

```
mod.in<-c(13.12,1.49,0.16,-0.11,-0.19,-0.16,0.56,-0.52,0.81)
struct<-c(2,2,1) #two inputs, two hidden, one output
plot.nnet(mod.in, struct=struct)
```



Note the comparability with the figure created using the neuralnet package. That is, larger weights have thicker lines and color indicates sign (+ black, - grey).

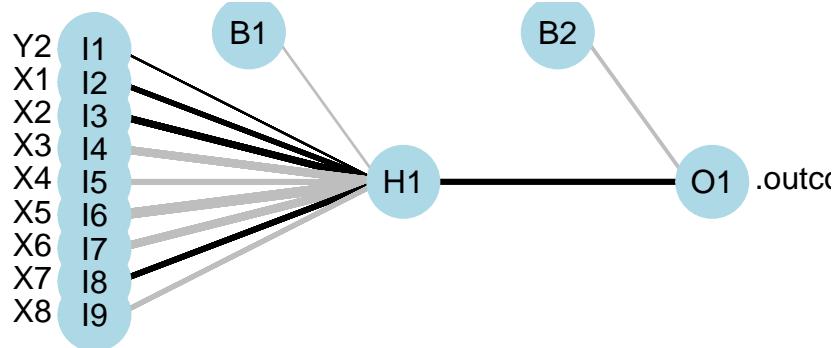
One of these days I'll actually put these functions in a package. In the meantime, please let me know if any bugs are encountered.

44.1 caret and plot NN

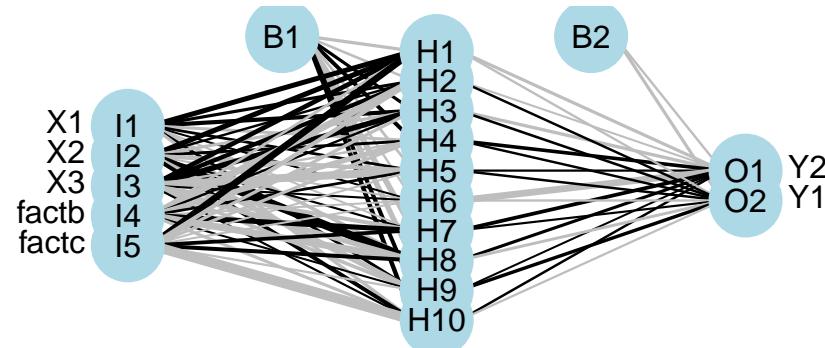
I've changed the function to work with neural networks created using the train function from the caret package. The link above is updated but you can also grab it here.

```
library(caret)
#> Loading required package: lattice
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures    rlang
#>   c.quosures    rlang
#>   print.quosures rlang
#>
#> Attaching package: 'caret'
#> The following objects are masked from 'package:RSNNS':
#>
#>   confusionMatrix, train
mod4 <- train(Y1 ~., method='nnet', data=dat.in, linout=T)
```

```
plot.nnet(mod4,nid=T)
#> Warning in plot.nnet(mod4, nid = T): Using best nnet model from train
#> output
```



```
fact<-factor(sample(c('a','b','c'),size=num.obs,replace=T))
form.in<-formula('cbind(Y2,Y1)~X1+X2+X3+fact')
mod5<-nnet(form.in,data=cbind(dat.in,fact),size=10,linout=T)
#> # weights:  82
#> initial value 4799569.423556
#> iter  10 value 2864553.218126
#> iter  20 value 2595828.194160
#> iter  30 value 2517965.483941
#> iter  40 value 2464882.178217
#> iter  50 value 2444238.700834
#> iter  60 value 2424302.290643
#> iter  70 value 2395226.949866
#> iter  80 value 2375558.751266
#> iter  90 value 2343011.050867
#> iter 100 value 2298860.593948
#> final  value 2298860.593948
#> stopped after 100 iterations
plot.nnet(mod5,nid=T)
```



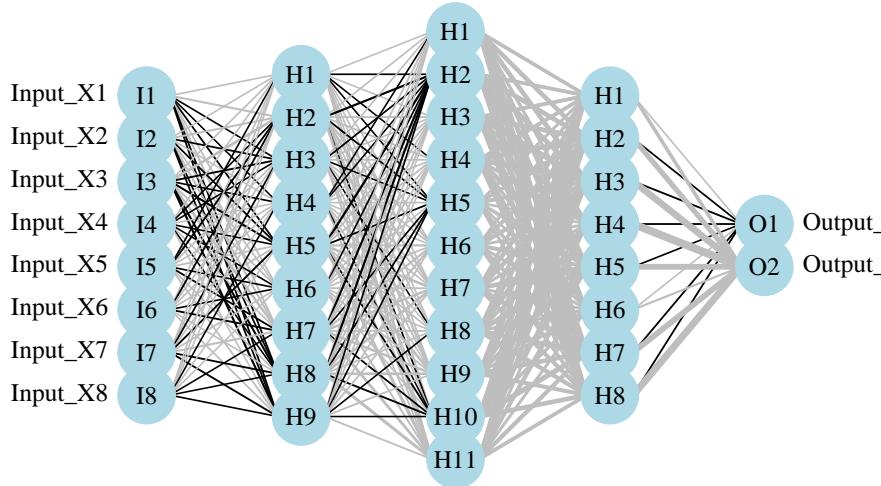
44.2 Multiple hidden layers

More updates... I've now modified the function to plot multiple hidden layers for networks created using the `mlp` function in the `RSNNS` package and `neuralnet` in the `neuralnet` package. As far as I know, these are the only neural network functions in R that can create multiple hidden layers. All others use a single hidden layer. I have not tested the plotting function using manual input for the weight vectors with multiple hidden layers.

My guess is it won't work but I can't be bothered to change the function unless it's specifically requested. The updated function can be grabbed here (all above links to the function have also been changed).

```
library(RSNNS)

# neural net with three hidden layers, 9, 11, and 8 nodes in each
tic()
mod <- mlp(rand.vars, resp,
            size = c(9,11,8),
            linOut = T)
toc()
#> 0.586 sec elapsed
par(mar=numeric(4),family='serif')
plot.nnet(mod)
#> Warning in plot.nnet(mod): Bias layer not applicable for rsnns object
```



44.3 Binary predictors

Here's an example using the `neuralnet` function with binary predictors and categorical outputs (credit to Tao Ma for the model code).

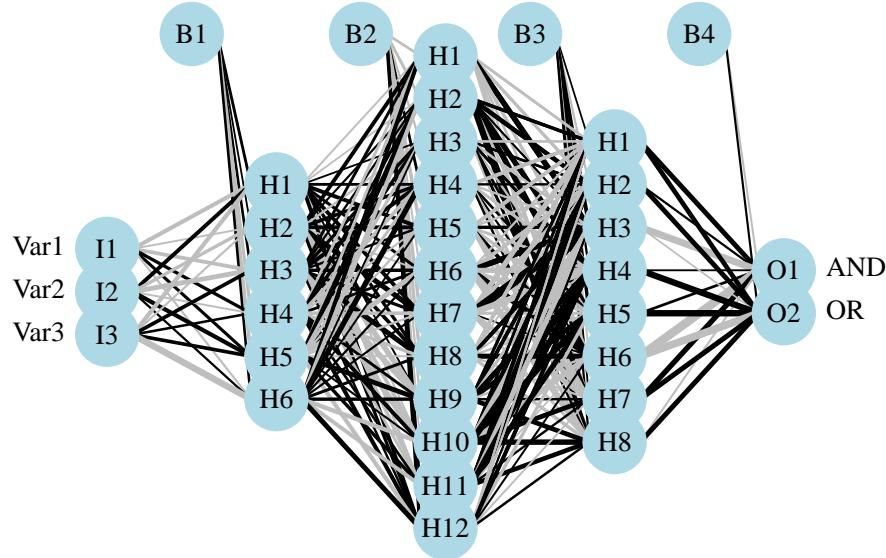
```
library(neuralnet)

#response
AND<-c(rep(0,7),1)
OR<-c(0,rep(1,7))

# response with predictors
binary.data <- data.frame(expand.grid(c(0,1), c(0,1), c(0,1)), AND, OR)

#model
tic()
net <- neuralnet(AND+OR ~ Var1+Var2+Var3,
                  binary.data, hidden =c(6,12,8),
                  rep = 10,
                  err.fct="ce",
                  linear.output=FALSE)
```

```
toc()
#> 0.183 sec elapsed
#plot output
par(mar=numeric(4),family='serif')
plot.nnet(net)
```



44.4 color coding the input layer

The color vector argument (circle.col) for the nodes was changed to allow a separate color vector for the input layer.

The following example shows how this can be done using relative importance of the input variables to color-code the first layer.

```
# example showing use of separate colors for input layer
# color based on relative importance using 'gar.fun'

##
#create input data
seed.val<-3
set.seed(seed.val)

num.vars<-8
num.obs<-1000

#input variables
library(clusterGeneration)
cov.mat<-genPositiveDefMat(num.vars,covMethod=c("unifcorrmat"))$Sigma
rand.vars<-mvrnorm(num.obs,rep(0,num.vars),Sigma=cov.mat)

# output variables
parms<-runif(num.vars,-10,10)
y1<-rand.vars %*% matrix(parms) + rnorm(num.obs, sd=20)
```

```

# final datasets
rand.vars<-data.frame(rand.vars)
resp<-data.frame(y1)
names(resp)<-'Y1'
dat.in <- data.frame(resp,rand.vars)

##
# create model
library(nnet)
mod1 <- nnet(rand.vars,resp,data=dat.in,size=10,linout=T)
#> # weights: 101
#> initial value 844959.580478
#> iter 10 value 543616.101824
#> iter 20 value 479986.887846
#> iter 30 value 465607.784054
#> iter 40 value 454237.073298
#> iter 50 value 445032.412421
#> iter 60 value 433191.158624
#> iter 70 value 426321.161292
#> iter 80 value 424900.966883
#> iter 90 value 423816.437605
#> iter 100 value 422064.114812
#> final value 422064.114812
#> stopped after 100 iterations

##
# relative importance function
library(devtools)
source_url('https://gist.github.com/fawda123/6206737/raw/2e1bc9cbc48d1a56d2a79dd1d33f414213f5f1b1/gar_f'
#> SHA-1 hash of file is 9faa58824c46956c3ff78081696290d9b32d845f

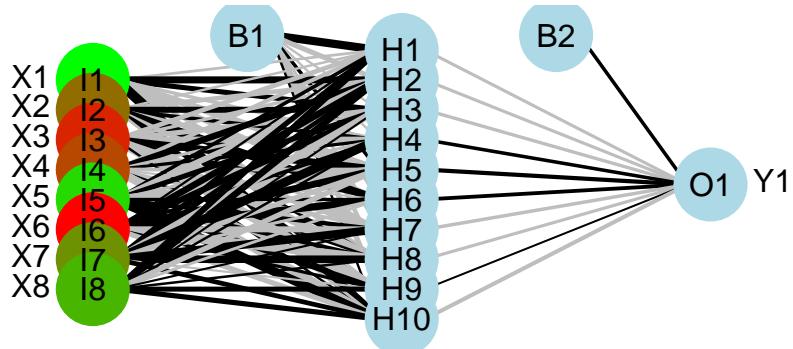
# relative importance of input variables for Y1
rel.imp <- gar.fun('Y1',mod1,bar.plot=F)$rel.imp

#color vector based on relative importance of input values
cols<-colorRampPalette(c('green','red'))(num.vars)[rank(rel.imp)]

##
#plotting function
source_url('https://gist.githubusercontent.com/fawda123/7471137/raw/466c1474d0a505ff044412703516c34f1a4'
#> SHA-1 hash of file is 74c80bd5ddbc17ab3ae5ece9c0ed9beb612e87ef

#plot model with new color vector
#separate colors for input vectors using a list for 'circle.col'
plot(mod1,circle.col=list(cols,'lightblue'))

```



Part VI

Linear Regression

Chapter 45

Temperature modeling using nested dataframes

45.1 Prepare the data

[http://ijlyttle.github.io/isugg_purrr/presentation.html#\(1\)](http://ijlyttle.github.io/isugg_purrr/presentation.html#(1))

45.1.1 Packages to run this presentation

```
library("readr")
library("tibble")
library("dplyr")
library("tidyverse")
library("stringr")
library("ggplot2")
library("purrr")
library("broom")
```

45.1.2 Motivation

As you know, purrr is a recent package from Hadley Wickham, focused on lists and functional programming, like dplyr is focused on data-frames.

I figure a good way to learn a new package is to try to solve a problem, so we have a dataset:

- you can view or download
- you can download the source of this presentation
- these are three temperatures recorded simultaneously in a piece of electronics
- it will be very valuable to be able to characterize the transient temperature for each sensor
- we want to apply the same set of models across all three sensors
- it will be easier to show using pictures

45.1.3 Let's get the data into shape

Using the `readr` package

```
temperature_wide <-
  read_csv(file.path(data_raw_dir, "temperature.csv")) %>%
  print()
#> Parsed with column specification:
#> cols(
#>   instant = col_datetime(format = ""),
#>   temperature_a = col_double(),
#>   temperature_b = col_double(),
#>   temperature_c = col_double()
#> )
#> # A tibble: 327 x 4
#>   instant      temperature_a temperature_b temperature_c
#>   <dttm>        <dbl>       <dbl>        <dbl>
#> 1 2015-11-13 06:10:19     116.       91.7       84.2
#> 2 2015-11-13 06:10:23     116.       91.7       84.2
#> 3 2015-11-13 06:10:27     116.       91.6       84.2
#> 4 2015-11-13 06:10:31     116.       91.7       84.2
#> 5 2015-11-13 06:10:36     116.       91.7       84.2
#> 6 2015-11-13 06:10:41     116.       91.6       84.2
#> # ... with 321 more rows
```

45.1.4 Is `temperature_wide` “tidy”?

```
#> # A tibble: 327 x 4
#>   instant      temperature_a temperature_b temperature_c
#>   <dttm>        <dbl>       <dbl>        <dbl>
#> 1 2015-11-13 06:10:19     116.       91.7       84.2
#> 2 2015-11-13 06:10:23     116.       91.7       84.2
#> 3 2015-11-13 06:10:27     116.       91.6       84.2
#> 4 2015-11-13 06:10:31     116.       91.7       84.2
#> 5 2015-11-13 06:10:36     116.       91.7       84.2
#> 6 2015-11-13 06:10:41     116.       91.6       84.2
#> # ... with 321 more rows
```

Why or why not?

45.1.5 Tidy data

1. Each column is a variable
2. Each row is an observation
3. Each cell is a value

(<http://www.jstatsoft.org/v59/i10/paper>)

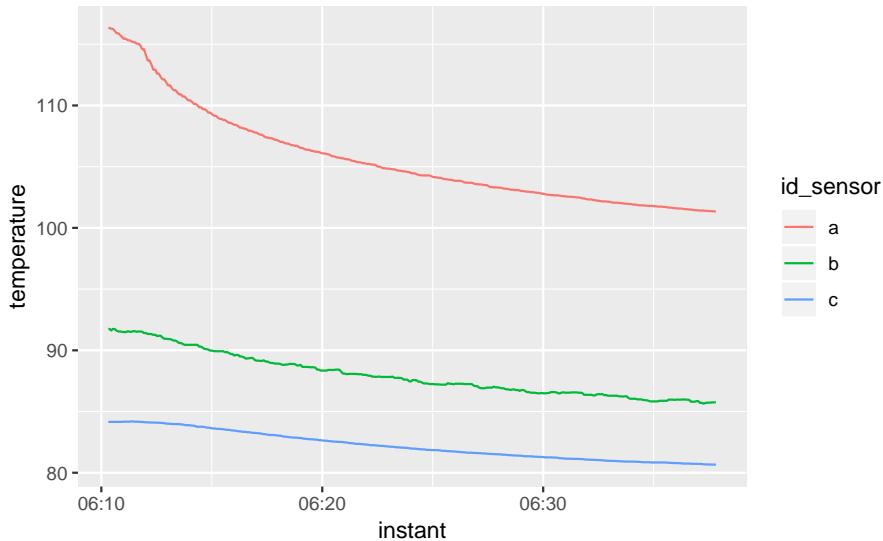
My personal observation is that “tidy” can depend on the context, on what you want to do with the data.

45.1.6 Let's get this into a tidy form

```
temperature_tall <-
  temperature_wide %>%
  gather(key = "id_sensor", value = "temperature", starts_with("temp")) %>%
  mutate(id_sensor = str_replace(id_sensor, "temperature_", ""))
  print()
#> # A tibble: 981 x 3
#>   instant      id_sensor temperature
#>   <dttm>      <chr>        <dbl>
#> 1 2015-11-13 06:10:19 a            116.
#> 2 2015-11-13 06:10:23 a            116.
#> 3 2015-11-13 06:10:27 a            116.
#> 4 2015-11-13 06:10:31 a            116.
#> 5 2015-11-13 06:10:36 a            116.
#> 6 2015-11-13 06:10:41 a            116.
#> # ... with 975 more rows
```

45.1.7 Now, it's easier to visualize

```
temperature_tall %>%
  ggplot(aes(x = instant, y = temperature, color = id_sensor)) +
  geom_line()
```



45.1.8 Calculate delta time (Δt) and delta temperature (ΔT)

`delta_time` Δt

change in time since event started, s

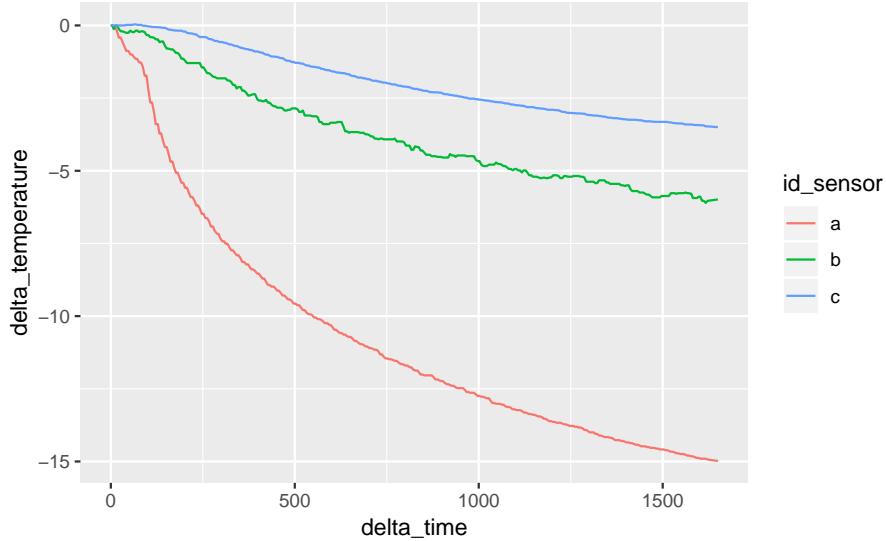
`delta_temperature`: ΔT

change in temperature since event started, °C

```
delta <-
  temperature_tall %>%
  arrange(id_sensor, instant) %>%
  group_by(id_sensor) %>%
  mutate(
    delta_time = as.numeric(instant) - as.numeric(instant[[1]]),
    delta_temperature = temperature - temperature[[1]]
  ) %>%
  select(id_sensor, delta_time, delta_temperature)
```

45.1.9 Let's have a look

```
# plot delta time vs delta temperature, by sensor
delta %>%
  ggplot(aes(x = delta_time, y = delta_temperature, color = id_sensor)) +
  geom_line()
```



45.2 Define the models

We want to see how three different curve-fits might perform on these three data-sets:

45.2.0.1 Newtonian cooling

$$\Delta T = \Delta T_0 * (1 - e^{-\frac{\delta t}{\tau_0}})$$

45.2.1 Semi-infinite solid

$$\Delta T = \Delta T_0 * erfc(\sqrt{\frac{\tau_0}{\delta t}})$$

45.2.2 Semi-infinite solid with convection

$$\Delta T = \Delta T_0 * [\operatorname{erfc}\left(\sqrt{\frac{\tau_0}{\delta t}}\right) - e^{Bi_0 + (\frac{Bi_0}{2})^2 \frac{\delta t}{\tau_0}} * \operatorname{erfc}\left(\sqrt{\frac{\tau_0}{\delta t}} + \frac{Bi_0}{2} * \sqrt{\frac{\delta t}{\tau_0}}\right)]$$

45.2.3 `erf` and `erfc` functions

```
# reference: http://stackoverflow.com/questions/29067916/r-error-function-erfz
# (see Abramowitz and Stegun 29.2.29)
erf <- function(x) 2 * pnorm(x * sqrt(2)) - 1
erfc <- function(x) 2 * pnorm(x * sqrt(2), lower = FALSE)
```

45.2.4 Newton cooling equation

```
newton_cooling <- function(x) {
  nls(
    delta_temperature ~ delta_temperature_0 * (1 - exp(-delta_time/tau_0)),
    start = list(delta_temperature_0 = -10, tau_0 = 50),
    data = x
  )
}
```

45.2.5 Temperature models: simple and convection

```
semi_infinite_simple <- function(x) {
  nls(
    delta_temperature ~ delta_temperature_0 * erfc(sqrt(tau_0 / delta_time)),
    start = list(delta_temperature_0 = -10, tau_0 = 50),
    data = x
  )
}

semi_infinite_convection <- function(x){
  nls(
    delta_temperature ~
      delta_temperature_0 * (
        erfc(sqrt(tau_0 / delta_time)) -
        exp(Bi_0 + (Bi_0/2)^2 * delta_time / tau_0) *
        erfc(sqrt(tau_0 / delta_time)) +
        (Bi_0/2) * sqrt(delta_time / tau_0))
    ),
    start = list(delta_temperature_0 = -5, tau_0 = 50, Bi_0 = 1.e6),
    data = x
  )
}
```

45.3 Test modeling on one dataset

45.3.1 Before going into purrr

Before doing anything, we want to show that we can do something with one dataset and one model-function:

```
# only one sensor; it is a test
tmp_data <- delta %>% filter(id_sensor == "a")

tmp_model <- newton_cooling(tmp_data)

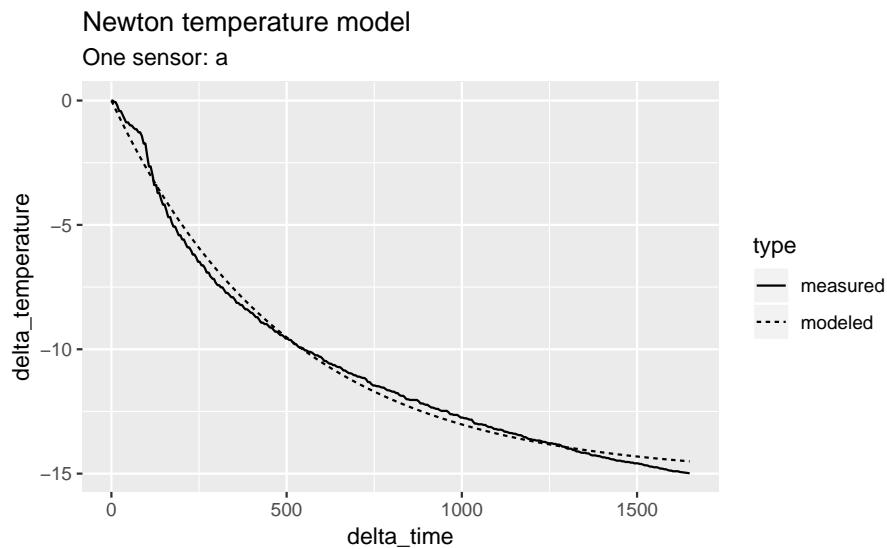
summary(tmp_model)
#>
#> Formula: delta_temperature ~ delta_temperature_0 * (1 - exp(-delta_time/tau_0))
#>
#> Parameters:
#>             Estimate Std. Error t value Pr(>|t|)
#> delta_temperature_0 -15.0608     0.0526    -286   <2e-16 ***
#> tau_0                500.0138    4.8367     103   <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.327 on 325 degrees of freedom
#>
#> Number of iterations to convergence: 7
#> Achieved convergence tolerance: 4.14e-06
```

45.3.2 Look at predictions

```
# apply prediction and make it tidy
tmp_pred <-
  tmp_data %>%
  mutate(modeled = predict(tmp_model, data = .)) %>%
  select(id_sensor, delta_time, measured = delta_temperature, modeled) %>%
  gather("type", "delta_temperature", measured:modeled) %>%
  print()
#> # A tibble: 654 x 4
#> # Groups:   id_sensor [1]
#>   id_sensor delta_time type      delta_temperature
#>   <chr>        <dbl> <chr>                <dbl>
#> 1 a            0 measured           0
#> 2 a            4 measured           0
#> 3 a            8 measured          -0.06
#> 4 a           12 measured          -0.06
#> 5 a           17 measured          -0.211
#> 6 a           22 measured          -0.423
#> # ... with 648 more rows
```

45.3.3 Plot Newton model

```
tmp_pred %>%
  ggplot(aes(x = delta_time, y = delta_temperature, linetype = type)) +
  geom_line() +
  labs(title = "Newton temperature model", subtitle = "One sensor: a")
```



45.3.4 “Regular” data-frame (deltas)

```
print(delta)
#> # A tibble: 981 x 3
#> # Groups:   id_sensor [3]
#>   id_sensor delta_time delta_temperature
#>   <chr>      <dbl>            <dbl>
#> 1 a            0              0
#> 2 a            4              0
#> 3 a            8             -0.06
#> 4 a           12             -0.06
#> 5 a           17             -0.211
#> 6 a           22             -0.423
#> # ... with 975 more rows
```

Each column of the data frame is a vector - in this case, a character vector and two doubles

45.4 Making a nested data frame

45.4.1 How to make a weird data-frame

Here's where the fun starts - a column of a data-frame can be a list.

- use `tidyr::nest()` to makes a column `data`, which is a list of data-frames
- this seems like a stronger expression of the `dplyr::group_by()` idea

```
# nest delta_time and delta_temperature variables
delta_nested <-
  delta %>%
  nest(-id_sensor) %>%
  print()
#> # A tibble: 3 x 2
#>   id_sensor      data
#>   <chr>        <list>
#> 1 a            <tibble [327 x 2]>
#> 2 b            <tibble [327 x 2]>
#> 3 c            <tibble [327 x 2]>
```

45.4.2 Map dataframes to a modeling function (Newton)

- `map()` is like `lapply()`
- `map()` returns a list-column (it keeps the weirdness)

```
model_nested <-
  delta_nested %>%
  mutate(model = map(data, newton_cooling)) %>%
  print()
#> # A tibble: 3 x 3
#>   id_sensor      data           model
#>   <chr>        <list>        <list>
#> 1 a            <tibble [327 x 2]> <nls>
#> 2 b            <tibble [327 x 2]> <nls>
#> 3 c            <tibble [327 x 2]> <nls>
```

We get an additional list-column `model`.

45.4.3 We can use `map2()` to make the predictions

- `map2()` is like `mapply()`
- designed to map two columns (`model`, `data`) to a function `predict()`

```
predict_nested <-
  model_nested %>%
  mutate(pred = map2(model, data, predict)) %>%
  print()
#> # A tibble: 3 x 4
#>   id_sensor      data           model  pred
#>   <chr>        <list>        <list> <list>
#> 1 a            <tibble [327 x 2]> <nls>  <dbl [327]>
#> 2 b            <tibble [327 x 2]> <nls>  <dbl [327]>
#> 3 c            <tibble [327 x 2]> <nls>  <dbl [327]>
```

Another list-column `pred` for the prediction results.

45.4.4 We need to get out of the weirdness

- use `unnest()` to get back to a regular data-frame

```

predict_unnested <-
  predict_nested %>%
  unnest(data, pred) %>%
  print()
#> # A tibble: 981 x 4
#>   id_sensor  pred delta_time delta_temperature
#>   <chr>      <dbl>     <dbl>            <dbl>
#> 1 a          0        0             0
#> 2 a         -0.120     4             0
#> 3 a         -0.239     8             -0.06
#> 4 a         -0.357    12             -0.06
#> 5 a         -0.503    17             -0.211
#> 6 a         -0.648    22             -0.423
#> # ... with 975 more rows

```

45.4.5 We can wrangle the predictions

- get into a form that makes it easier to plot

```

predict_tall <-
  predict_unnested %>%
  rename(modeled = pred, measured = delta_temperature) %>%
  gather("type", "delta_temperature", modeled, measured) %>%
  print()
#> # A tibble: 1,962 x 4
#>   id_sensor delta_time type      delta_temperature
#>   <chr>      <dbl> <chr>            <dbl>
#> 1 a           0     modeled       0
#> 2 a           4     modeled     -0.120
#> 3 a           8     modeled     -0.239
#> 4 a          12    modeled     -0.357
#> 5 a          17    modeled     -0.503
#> 6 a          22    modeled     -0.648
#> # ... with 1,956 more rows

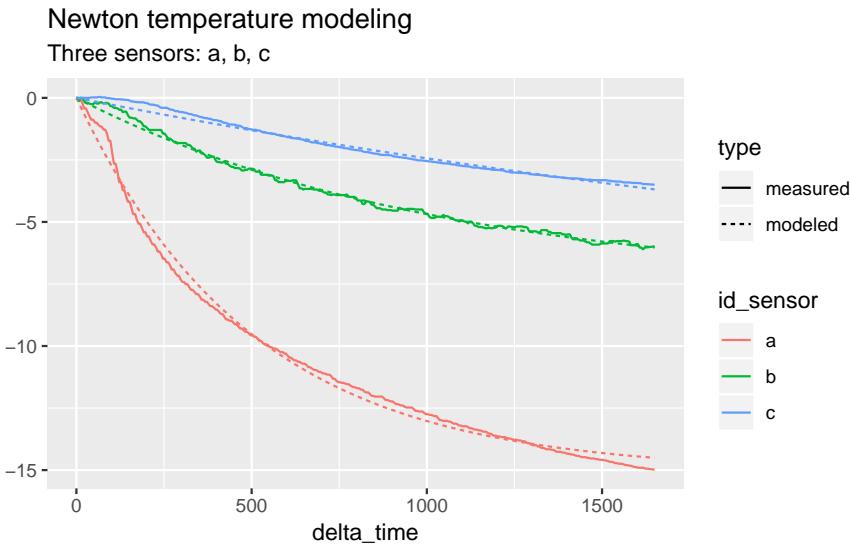
```

45.4.6 We can visualize the predictions

```

predict_tall %>%
  ggplot(aes(x = delta_time, y = delta_temperature)) +
  geom_line(aes(color = id_sensor, linetype = type)) +
  labs(title = "Newton temperature modeling",
       subtitle = "Three sensors: a, b, c")

```



45.5 Apply multiple models on a nested structure

45.5.1 Step 1: Selection of models

Make a list of functions to model:

```
list_model <-  
  list(  
    newton_cooling = newton_cooling,  
    semi_infinite_simple = semi_infinite_simple,  
    semi_infinite_convection = semi_infinite_convection  
)
```

45.5.2 Step 2: write a function to define the “inner” loop

```
# add additional variable with the model name  
  
fn_model <- function(.model, df) {  
  # one parameter for the model in the list, the second for the data  
  # safer to avoid non-standard evaluation  
  # df %>% mutate(model = map(data, .model))  
  
  df$model <- map(df$data, possibly(.model, NULL))  
  df  
}
```

- for a given model-function and a given (weird) data-frame, return a modified version of that data-frame with a column `model`, which is the model-function applied to each element of the data-frame’s `data` column (which is itself a list of data-frames)
- the purrr functions `safely()` and `possibly()` are **very** interesting. I think they could be useful outside of purrr as a friendlier way to do error-handling.

45.5.3 Step 3: Use `map_df()` to define the “outer” loop

```
# this dataframe will be the second input of fn_model
delta_nested %>%
  print()
#> # A tibble: 3 x 2
#>   id_sensor      data
#>   <chr>        <list>
#> 1 a            <tibble [327 x 2]>
#> 2 b            <tibble [327 x 2]>
#> 3 c            <tibble [327 x 2]>

# fn_model is receiving two inputs: one from list_model and from delta_nested
model_nested_new <-
  list_model %>%
  map_df(fn_model, delta_nested, .id = "id_model") %>%
  print()
#> # A tibble: 9 x 4
#>   id_model      id_sensor data          model
#>   <chr>        <chr>    <list>        <list>
#> 1 newton_cooling a        <tibble [327 x 2]> <nls>
#> 2 newton_cooling b        <tibble [327 x 2]> <nls>
#> 3 newton_cooling c        <tibble [327 x 2]> <nls>
#> 4 semi_infinite_simple a <tibble [327 x 2]> <nls>
#> 5 semi_infinite_simple b <tibble [327 x 2]> <nls>
#> 6 semi_infinite_simple c <tibble [327 x 2]> <nls>
#> # ... with 3 more rows
```

- for each element of a list of model-functions, run the inner-loop function, and row-bind the results into a data-frame
- we want to discard the rows where the model failed
- we also want to investigate why they failed, but that’s a different talk

45.5.4 Step 4: Use `map()` to identify the null models

```
model_nested_new <-
  list_model %>%
  map_df(fn_model, delta_nested, .id = "id_model") %>%
  mutate(is_null = map(model, is.null)) %>%
  print()
#> # A tibble: 9 x 5
#>   id_model      id_sensor data          model  is_null
#>   <chr>        <chr>    <list>        <list> <list>
#> 1 newton_cooling a        <tibble [327 x 2]> <nls> <lgl [1]>
#> 2 newton_cooling b        <tibble [327 x 2]> <nls> <lgl [1]>
#> 3 newton_cooling c        <tibble [327 x 2]> <nls> <lgl [1]>
#> 4 semi_infinite_simple a <tibble [327 x 2]> <nls> <lgl [1]>
#> 5 semi_infinite_simple b <tibble [327 x 2]> <nls> <lgl [1]>
#> 6 semi_infinite_simple c <tibble [327 x 2]> <nls> <lgl [1]>
#> # ... with 3 more rows
```

- using `map(model, is.null)` returns a list column

- to use `filter()`, we have to escape the weirdness

45.5.5 Step 5: `map_lgl()` to identify nulls and get out of the weirdness

```
model_nested_new <-
  list_model %>%
  map_df(fn_model, delta_nested, .id = "id_model") %>%
  mutate(is_null = map_lgl(model, is.null)) %>%
  print()
#> # A tibble: 9 x 5
#>   id_model      id_sensor data      model  is_null
#>   <chr>        <chr>     <list>    <list> <lgl>
#> 1 newton_cooling a        <tibble [327 x 2]> <nls> FALSE
#> 2 newton_cooling b        <tibble [327 x 2]> <nls> FALSE
#> 3 newton_cooling c        <tibble [327 x 2]> <nls> FALSE
#> 4 semi_infinite_simple a <tibble [327 x 2]> <nls> FALSE
#> 5 semi_infinite_simple b <tibble [327 x 2]> <nls> FALSE
#> 6 semi_infinite_simple c <tibble [327 x 2]> <nls> FALSE
#> # ... with 3 more rows
```

- using `map_lgl(model, is.null)` returns a vector column

45.5.6 Step 6: `filter()` nulls and `select()` variables to clean up

```
model_nested_new <-
  list_model %>%
  map_df(fn_model, delta_nested, .id = "id_model") %>%
  mutate(is_null = map_lgl(model, is.null)) %>%
  filter(!is_null) %>%
  select(-is_null) %>%
  print()
#> # A tibble: 6 x 4
#>   id_model      id_sensor data      model
#>   <chr>        <chr>     <list>    <list>
#> 1 newton_cooling a        <tibble [327 x 2]> <nls>
#> 2 newton_cooling b        <tibble [327 x 2]> <nls>
#> 3 newton_cooling c        <tibble [327 x 2]> <nls>
#> 4 semi_infinite_simple a <tibble [327 x 2]> <nls>
#> 5 semi_infinite_simple b <tibble [327 x 2]> <nls>
#> 6 semi_infinite_simple c <tibble [327 x 2]> <nls>
```

45.5.7 Step 7: Calculate predictions on nested dataframe

```
predict_nested <-
  model_nested_new %>%
  mutate(pred = map2(model, data, predict)) %>%
  print()
#> # A tibble: 6 x 5
#>   id_model      id_sensor data      model  pred
#>   <chr>        <chr>     <list>    <list> <list>
```

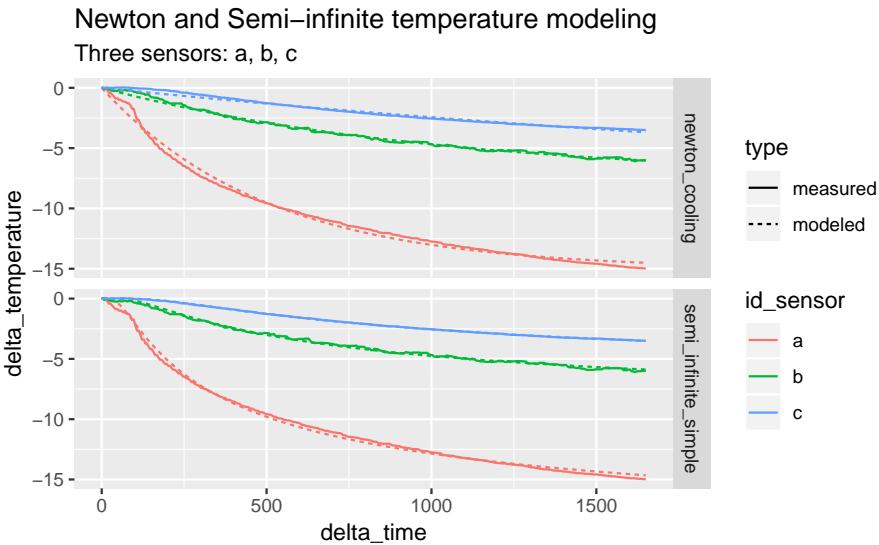
```
#> 1 newton_cooling      a      <tibble [327 x 2]> <nls>  <dbl [327]>
#> 2 newton_cooling      b      <tibble [327 x 2]> <nls>  <dbl [327]>
#> 3 newton_cooling      c      <tibble [327 x 2]> <nls>  <dbl [327]>
#> 4 semi_infinite_simple a    <tibble [327 x 2]> <nls>  <dbl [327]>
#> 5 semi_infinite_simple b    <tibble [327 x 2]> <nls>  <dbl [327]>
#> 6 semi_infinite_simple c    <tibble [327 x 2]> <nls>  <dbl [327]>
```

45.5.8 `unnest()`, make it tall and tidy

```
predict_tall <-
  predict_nested %>%
  unnest(data, pred) %>%
  rename(modeled = pred, measured = delta_temperature) %>%
  gather("type", "delta_temperature", modeled, measured) %>%
  print()
#> # A tibble: 3,924 x 5
#>   id_model      id_sensor delta_time type    delta_temperature
#>   <chr>          <chr>       <dbl> <chr>           <dbl>
#> 1 newton_cooling a            0 modeled        0
#> 2 newton_cooling a            4 modeled     -0.120
#> 3 newton_cooling a            8 modeled     -0.239
#> 4 newton_cooling a           12 modeled     -0.357
#> 5 newton_cooling a           17 modeled     -0.503
#> 6 newton_cooling a           22 modeled     -0.648
#> # ... with 3,918 more rows
```

45.5.9 Visualize the predictions

```
predict_tall %>%
  ggplot(aes(x = delta_time, y = delta_temperature)) +
  geom_line(aes(color = id_sensor, linetype = type)) +
  facet_grid(id_model ~ .) +
  labs(title = "Newton and Semi-infinite temperature modeling",
       subtitle = "Three sensors: a, b, c")
```

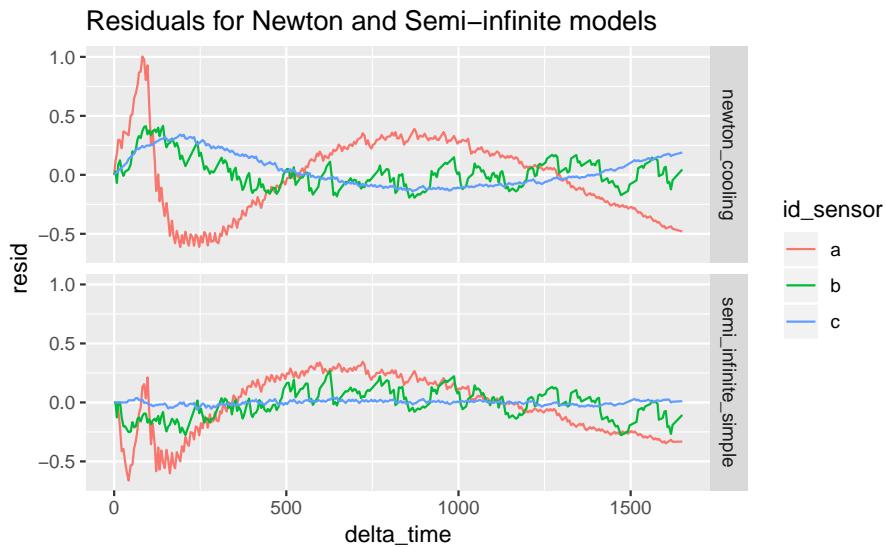


45.5.10 Let's get the residuals

```
resid <-
  model_nested_new %>%
  mutate(resid = map(model, resid)) %>%
  unnest(data, resid) %>%
  print()
#> # A tibble: 1,962 x 5
#>   id_model      id_sensor resid  delta_time delta_temperature
#>   <chr>        <chr>    <dbl>     <dbl>            <dbl>
#> 1 newton_cooling a         0       0             0
#> 2 newton_cooling a        0.120    4             0
#> 3 newton_cooling a        0.179    8            -0.06
#> 4 newton_cooling a        0.297   12            -0.06
#> 5 newton_cooling a        0.292   17            -0.211
#> 6 newton_cooling a        0.225   22            -0.423
#> # ... with 1,956 more rows
```

45.5.11 And visualize them

```
resid %>%
  ggplot(aes(x = delta_time, y = resid)) +
  geom_line(aes(color = id_sensor)) +
  facet_grid(id_model ~ .) +
  labs(title = "Residuals for Newton and Semi-infinite models")
```



45.6 Using broom package to look at model-statistics

We will use a previous defined dataframe with the model and data:

```
model_nested_new %>%
  print()
#> # A tibble: 6 x 4
#>   id_model      id_sensor data          model
#>   <chr>        <chr>    <list>        <list>
#> 1 newton_cooling a        <tibble [327 x 2]> <nls>
#> 2 newton_cooling b        <tibble [327 x 2]> <nls>
#> 3 newton_cooling c        <tibble [327 x 2]> <nls>
#> 4 semi_infinite_simple a <tibble [327 x 2]> <nls>
#> 5 semi_infinite_simple b <tibble [327 x 2]> <nls>
#> 6 semi_infinite_simple c <tibble [327 x 2]> <nls>
```

The `tidy()` function extracts statistics from a model.

```
# apply over model_nested_new but only three variables
model_parameters <- 
  model_nested_new %>%
  select(id_model, id_sensor, model) %>%
  mutate(tidy = map(model, tidy)) %>%
  select(-model) %>%
  unnest() %>%
  print()
#> # A tibble: 12 x 7
#>   id_model id_sensor term      estimate std.error statistic p.value
#>   <chr>     <chr>    <chr>      <dbl>     <dbl>      <dbl>    <dbl>
#> 1 newton_coo~ a   delta_tempe~ -15.1      0.0526    -286.   0.
#> 2 newton_coo~ a   tau_0       500.       4.84     103.  1.07e-250
#> 3 newton_coo~ b   delta_tempe~ -7.59      0.0676    -112.  6.38e-262
#> 4 newton_coo~ b   tau_0       1041.      16.2      64.2  9.05e-187
#> 5 newton_coo~ c   delta_tempe~ -9.87      0.704    -14.0  3.16e- 35
#> 6 newton_coo~ c   tau_0       3525.      299.      11.8  5.61e- 27
#> # ... with 6 more rows
```

45.6.1 Get a sense of the coefficients

```
model_summary <-
  model_parameters %>%
  select(id_model, id_sensor, term, estimate) %>%
  spread(key = "term", value = "estimate") %>%
  print()
#> # A tibble: 6 x 4
#>   id_model      id_sensor delta_temperature_0 tau_0
#>   <chr>        <chr>           <dbl>    <dbl>
#> 1 newton_cooling a            -15.1     500.
#> 2 newton_cooling b            -7.59    1041.
#> 3 newton_cooling c            -9.87    3525.
#> 4 semi_infinite_simple a     -21.5     139.
#> 5 semi_infinite_simple b     -10.6     287.
#> 6 semi_infinite_simple c     -8.04     500.
```

45.6.2 Summary

- this is just a small part of purrr
- there seem to be parallels between `tidy::nest()`/`purrr::map()` and `dplyr::group_by()`/`dplyr::do()`
 - to my mind, the purrr framework is more understandable
 - update tweet from Hadley

References from Hadley:

- purrr 0.1.0 announcement
- purrr 0.2.0 announcement
- chapter from Garrett Grolemund and Hadley's forthcoming book

Chapter 46

Linear Regression. World Happiness

46.1 Introduction

Source: <http://enhanceddatascience.com/2017/04/25/r-basics-linear-regression-with-r/> Data: <https://www.kaggle.com/unsdn/world-happiness>

Linear regression is one of the basics of statistics and machine learning. Hence, it is a must-have to know how to perform a linear regression with R and how to interpret the results.

Linear regression algorithm will fit the best straight line that fits the data? To do so, it will minimise the squared distance between the points of the dataset and the fitted line.

For this tutorial, we will use the World Happiness report dataset from Kaggle. This report analyses the Happiness of each country according to several factors such as wealth, health, family life, ... Our goal will be to find the most important factors of happiness. What a noble goal!

46.2 A quick exploration of the data

Before fitting any model, we need to know our data better. First, let's import the data into R. Please download the dataset from Kaggle and put it in your working directory.

The code below imports the data as data.table and clean the column names (a lot of . were appearing in the original ones)

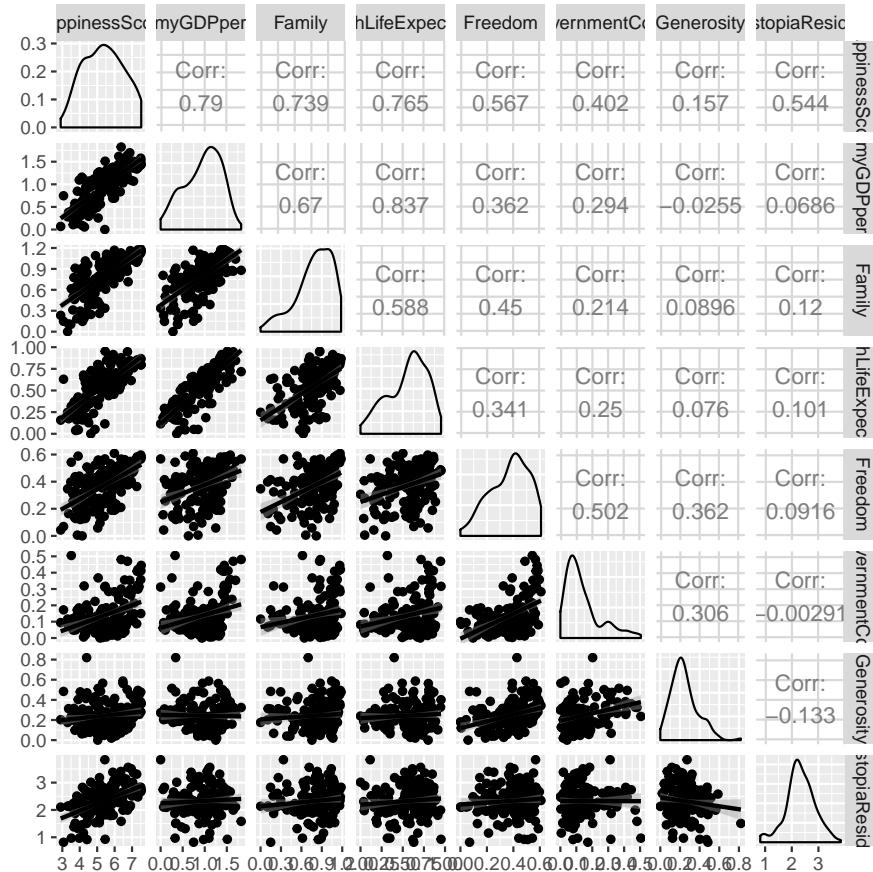
```
require(data.table)
#> Loading required package: data.table
data_happiness_dir <- file.path(data_raw_dir, "happiness")

Happiness_Data = data.table(read.csv(file.path(data_happiness_dir, '2016.csv')))
colnames(Happiness_Data) <- gsub('.','_', colnames(Happiness_Data), fixed=T)
```

Now, let's plot a Scatter Plot Matrix to get a grasp of how our variables are related one to another. To do so, the GGally package is great.

```
require(ggplot2)
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method          from
#>   [.quosures     rlang
```

```
#>   c.quotures    rlang
#>   print.quotures rlang
require(GGally)
#> Loading required package: GGally
#> Registered S3 method overwritten by 'GGally':
#>   method from
#>   +.gg  ggplot2
ggpairs(Happiness_Data[,c(4,7:13), with=F], lower = list(continuous = "smooth"))
```



All the variables are positively correlated with the Happiness score. We can expect that most of the coefficients in the linear regression will be positive. However, the correlation between the variable is often more than 0.5, so we can expect that multicollinearity will appear in the regression.

In the data, we also have access to the Country where the score was computed. Even if it's not useful for the regression, let's plot the data on a map!

```
require('rworldmap')
#> Loading required package: rworldmap
#> Loading required package: sp
#> ### Welcome to rworldmap ####
#> For a short introduction type : vignette('rworldmap')
library(reshape2)
#>
#> Attaching package: 'reshape2'
#> The following objects are masked from 'package:data.table':
#>
#>     dcast, melt
```

```

map.world <- map_data(map="world")

dataPlot<- melt(Happiness_Data, id.vars ='Country',
                 measure.vars = colnames(Happiness_Data)[c(4,7:13)])

#Correcting names that are different
dataPlot[Country == 'United States', Country:='USA']
dataPlot[Country == 'United Kingdoms', Country:='UK']

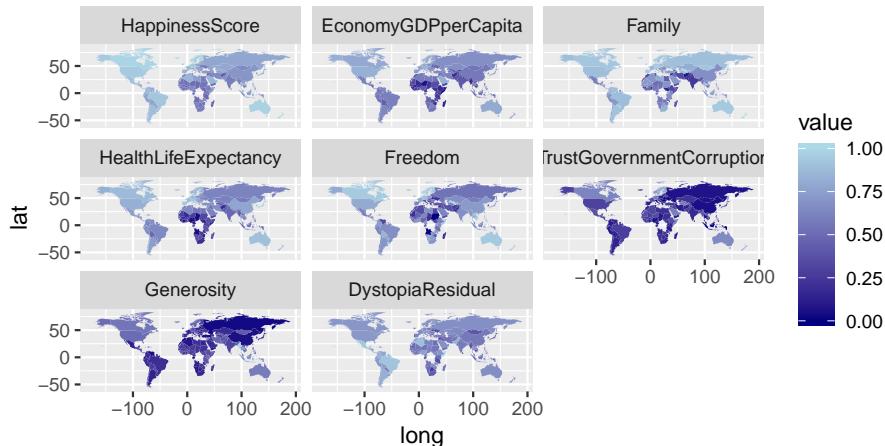
##Rescaling each variable to have nice gradient
dataPlot[,value:=value/max(value), by=variable]
dataMap = data.table(merge(map.world, dataPlot,
                           by.x='region',
                           by.y='Country',
                           all.x=T))
dataMap = dataMap[order(order)]
dataMap = dataMap[order(order)][!is.na(variable)]
gg <- ggplot()
gg <- gg +
  geom_map(data=dataMap, map=dataMap,
            aes(map_id = region, x=long, y=lat, fill=value)) +
  # facet_wrap(~variable, scale='free')
  facet_wrap(~variable)
#> Warning: Ignoring unknown aesthetics: x, y
gg <- gg + scale_fill_gradient(low = "navy", high = "lightblue")
gg <- gg + coord_equal()

```

The code above is a classic code for a map. A few important points:

We reordered the point before plotting to avoid some artefacts. The merge is a right outer join, all the points of the map need to be kept. Otherwise, points will be missing which will mess up the map. Each variable is rescaled so that a facet_wrap can be used. Here, the absolute level of a variable is not of primary interest. This is the relative level of a variable between countries that we want to visualise.

gg



The distinction between North and South is quite visible. In addition to this, countries that have suffered from the crisis are also really visible.

46.3 Linear regression with R

Now that we have taken a look at our data, a first model can be fitted. The explanatory variables are the DGP per capita, the life expectancy, the level of freedom and the trust in the government.

```
##First model
model1 <- lm(HappinessScore ~ EconomyGDPperCapita + Family +
                 HealthLifeExpectancy + Freedom + TrustGovernmentCorruption,
                 data=Happiness_Data)
```

46.4 Regression summary

The summary function provides a very easy way to assess a linear regression in R.

```
require(stargazer)
#> Loading required package: stargazer
#>
#> # Please cite as:
#> Hlavac, Marek (2018). stargazer: Well-Formatted Regression and Summary Statistics Tables.
#> R package version 5.2.2. https://CRAN.R-project.org/package=stargazer

##Quick summary
sum1=summary(model1)
sum1
#>
#> Call:
#> lm(formula = HappinessScore ~ EconomyGDPperCapita + Family +
#>     HealthLifeExpectancy + Freedom + TrustGovernmentCorruption,
#>     data = Happiness_Data)
#>
#> Residuals:
#>    Min      1Q  Median      3Q     Max
#> -1.4833 -0.2817 -0.0277  0.3280  1.4615
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)    
#> (Intercept)  2.212     0.150   14.73 < 2e-16 ***
#> EconomyGDPperCapita 0.697     0.209    3.33  0.0011 ** 
#> Family       1.234     0.229    5.39  2.6e-07 ***
#> HealthLifeExpectancy 1.462     0.343    4.26  3.5e-05 ***
#> Freedom       1.559     0.373    4.18  5.0e-05 ***
#> TrustGovernmentCorruption 0.959     0.455    2.11  0.0365 *  
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.535 on 151 degrees of freedom
#> Multiple R-squared:  0.787, Adjusted R-squared:  0.78 
#> F-statistic: 112 on 5 and 151 DF,  p-value: <2e-16

stargazer(model1,type='text')
#>
#> =====
#> Dependent variable:
```

```
#> -----
#>             HappinessScore
#> -----
#> EconomyGDPperCapita      0.697***  

#>                               (0.209)  

#>  

#> Family                   1.230***  

#>                               (0.229)  

#>  

#> HealthLifeExpectancy    1.460***  

#>                               (0.343)  

#>  

#> Freedom                  1.560***  

#>                               (0.373)  

#>  

#> TrustGovernmentCorruption 0.959**  

#>                               (0.455)  

#>  

#> Constant                 2.210***  

#>                               (0.150)  

#>  

#> -----
#> Observations            157  

#> R2                      0.787  

#> Adjusted R2              0.780  

#> Residual Std. Error     0.535 (df = 151)  

#> F Statistic              112.000*** (df = 5; 151)
#> -----
#> Note: *p<0.1; **p<0.05; ***p<0.01
```

A quick interpretation:

- All the coefficient are significative at a .05 threshold
- The overall model is also significative
- It explains 78.7% of Happiness in the dataset
- As expected all the relationship between the explanatory variables and the output variable are positives.

The model is doing well!

You can also easily get a given indicator of the model performance, such as R^2 , the different coefficients or the p-value of the overall model.

```
##R2  

sum1$r.squared*100  

#> [1] 78.7  

##Coefficients  

sum1$coefficients  

#>                               Estimate Std. Error t value Pr(>|t|)  

#> (Intercept)                2.212     0.150   14.73 5.20e-31  

#> EconomyGDPperCapita       0.697     0.209    3.33 1.10e-03  

#> Family                     1.234     0.229    5.39 2.62e-07  

#> HealthLifeExpectancy      1.462     0.343    4.26 3.53e-05  

#> Freedom                     1.559     0.373    4.18 5.01e-05  

#> TrustGovernmentCorruption  0.959     0.455    2.11 3.65e-02  

##p-value  

df(sum1$fstatistic[1], sum1$fstatistic[2], sum1$fstatistic[3])
```

```
#>      value
#> 3.39e-49

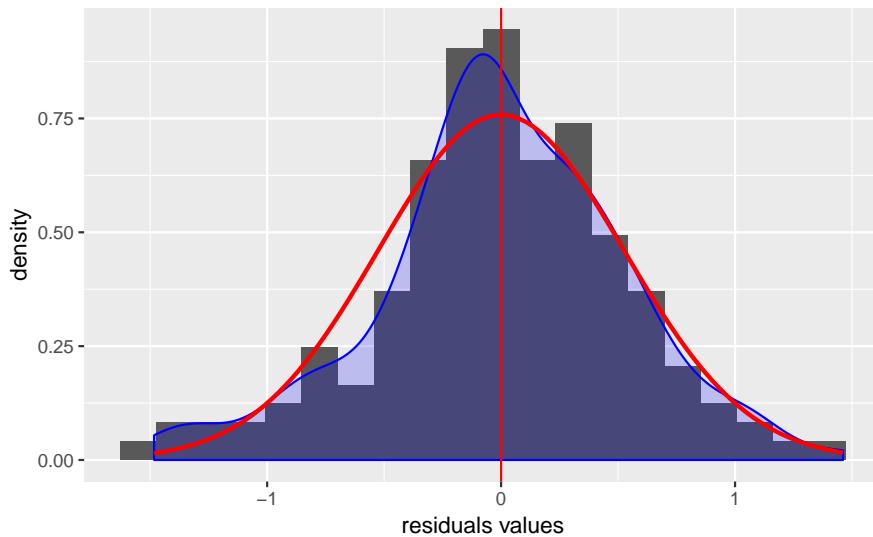
##Confidence interval of the coefficient
confint(model1, level = 0.95)
#>              2.5 % 97.5 %
#> (Intercept)    1.9152   2.51
#> EconomyGDPperCapita 0.2833   1.11
#> Family        0.7821   1.69
#> HealthLifeExpectancy 0.7846   2.14
#> Freedom        0.8212   2.30
#> TrustGovernmentCorruption 0.0609   1.86
confint(model1, level = 0.99)
#>              0.5 % 99.5 %
#> (Intercept)    1.820    2.60
#> EconomyGDPperCapita 0.151    1.24
#> Family        0.637    1.83
#> HealthLifeExpectancy 0.568    2.36
#> Freedom        0.585    2.53
#> TrustGovernmentCorruption -0.227   2.14
confint(model1, level = 0.90)
#>              5 % 95 %
#> (Intercept)    1.963   2.46
#> EconomyGDPperCapita 0.350   1.04
#> Family        0.856   1.61
#> HealthLifeExpectancy 0.895   2.03
#> Freedom        0.941   2.18
#> TrustGovernmentCorruption 0.207   1.71
```

46.5 Regression analysis

46.5.1 Residual analysis

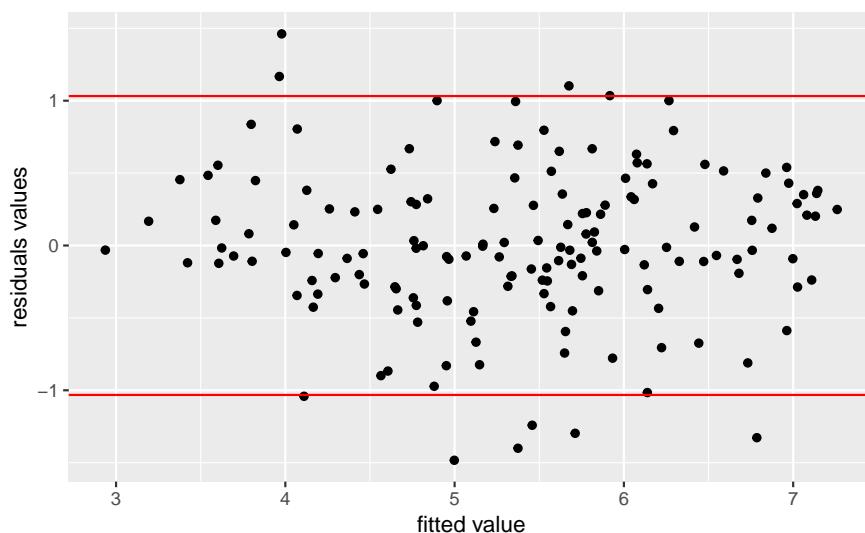
Now that the regression has been done, the analysis and validity of the result can be analysed. Let's begin with residuals and the assumption of normality and homoscedasticity.

```
# Visualisation of residuals
ggplot(model1, aes(model1$residuals)) +
  geom_histogram(bins=20, aes(y = ..density..)) +
  geom_density(color='blue', fill = 'blue', alpha = 0.2) +
  geom_vline(xintercept = mean(model1$residuals), color='red') +
  stat_function(fun=dnorm, color="red", size=1,
               args = list(mean = mean(model1$residuals),
                           sd = sd(model1$residuals))) +
  xlab('residuals values')
```



The residual versus fitted plot is used to see if the residuals behave the same for the different value of the output (i.e., they have the same variance and mean). The plot shows no strong evidence of heteroscedasticity.

```
ggplot(model1, aes(model1$fitted.values, model1$residuals)) +
  geom_point() +
  geom_hline(yintercept = c(1.96 * sd(model1$residuals),
                           - 1.96 * sd(model1$residuals)), color='red') +
  xlab('fitted value') +
  ylab('residuals values')
```



46.6 Analysis of colinearity

The colinearity can be assessed using VIF, the car package provides a function to compute it directly.

```
require('car')
#> Loading required package: car
#> Loading required package: carData
vif(model1)
```

```
#>      EconomyGDPperCapita           Family
#>              4.07                  2.03
#>      HealthLifeExpectancy        Freedom
#>              3.37                  1.61
#> TrustGovernmentCorruption
#>              1.39
```

All the VIF are less than 5, and hence there is no sign of colinearity.

46.7 What drives happiness

Now let's compute standardised betas to see what really drives happiness.

```
##Standardized betas
std_betas = sum1$coefficients[-1,1] *
  data.table(model1$model) [, lapply(.SD, sd), .SDcols=2:6] /
  sd(model1$model$HappinessScore)

std_betas
#>      EconomyGDPperCapita Family HealthLifeExpectancy Freedom
#> 1:             0.252   0.288          0.294    0.199
#> TrustGovernmentCorruption
#> 1:                 0.0933
```

Though the code above may seem complicated, it is just computing the standardised betas for all variables `std_beta=beta*sd(x)/sd(y)`.

The top three coefficients are **Health and Life expectancy, Family and GDP per Capita**. Though money does not make happiness it is among the top three factors of Happiness!

Now you know how to perform a linear regression with R!

Chapter 47

Linear Regression on Advertising

Videos, slides:

- <https://www.r-bloggers.com/in-depth-introduction-to-machine-learning-in-15-hours-of-expert-videos/>

Data:

- <http://www-bcf.usc.edu/~gareth/ISL/Advertising.csv>

code:

- <http://subashish.github.io/pages/ISLwithR/>
- <http://math480-s15-zarringhalam.wikispaces.umb.edu/R+Code>
- <https://github.com/yahwes/ISLR>
- <https://www.tau.ac.il/~saharon/IntroStatLearn.html>
- https://www.waxworksmath.com/Authors/G_M/James/WWW/chapter_3.html
- <https://github.com/asadoughi/stat-learning>

plots:

- <https://onlinecourses.science.psu.edu/stat857/node/28/>

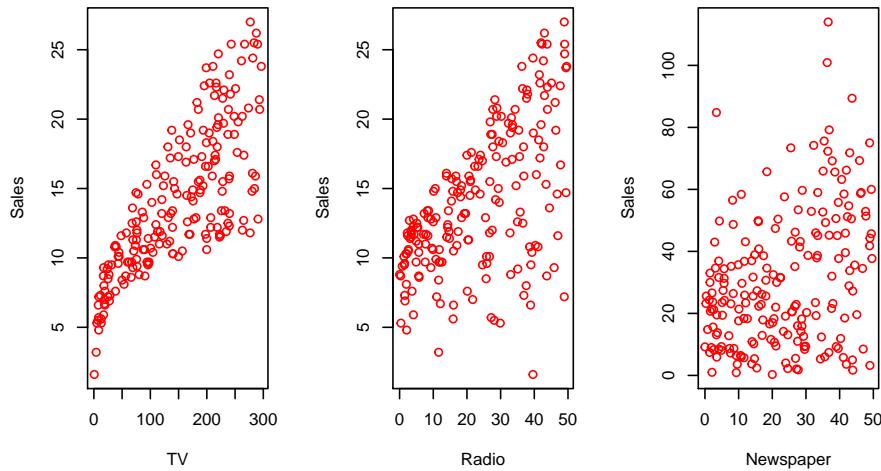
```
library(readr)
```

```
advertising <- read_csv(file.path(data_raw_dir, "Advertising.csv"))
#> Warning: Missing column names filled in: 'X1' [1]
#> Parsed with column specification:
#> cols(
#>   X1 = col_double(),
#>   TV = col_double(),
#>   radio = col_double(),
#>   newspaper = col_double(),
#>   sales = col_double()
#> )
advertising
#> # A tibble: 200 x 5
#>       X1     TV radio newspaper sales
#>   <dbl> <dbl> <dbl>    <dbl> <dbl>
#> 1     1  230.   37.8     69.2  22.1
#> 2     2   44.5   39.3     45.1  10.4
#> 3     3   17.2   45.9     69.3   9.3
#> 4     4  152.   41.3     58.5  18.5
#> 5     5  181.   10.8     58.4  12.9
```

```
#> 6      6   8.7 48.9      75      7.2
#> # ... with 194 more rows
```

The Advertising data set. The plot displays sales, in thousands of units, as a function of TV, radio, and newspaper budgets, in thousands of dollars, for 200 different markets.

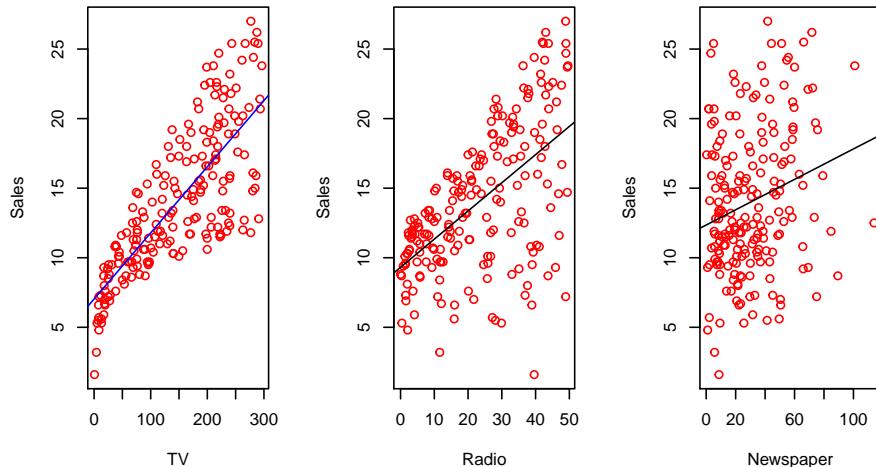
```
par(mfrow=c(1,3))
plot(advertising$TV, advertising$sales, xlab = "TV", ylab = "Sales", col = "red")
plot(advertising$radio, advertising$sales, xlab="Radio", ylab="Sales", col="red")
plot(advertising$radio, advertising$newspaper, xlab="Newspaper",
     ylab="Sales", col="red")
```



In each plot we show the simple least squares fit of sales to that variable, as described in Chapter 3. In other words, each blue line represents a simple model that can be used to predict sales using TV, radio, and newspaper, respectively.

```
par(mfrow=c(1,3))
tv_model <- lm(sales ~ TV, data = advertising)
radio_model <- lm(sales ~ radio, data = advertising)
newspaper_model <- lm(sales ~ newspaper, data = advertising)

plot(advertising$TV, advertising$sales, xlab = "TV", ylab = "Sales", col = "red")
abline(tv_model, col = "blue")
plot(advertising$radio, advertising$sales, xlab="Radio", ylab="Sales", col="red")
abline(radio_model)
plot(advertising$newspaper, advertising$sales, xlab="Newspaper",
     ylab="Sales", col="red")
abline(newspaper_model)
```

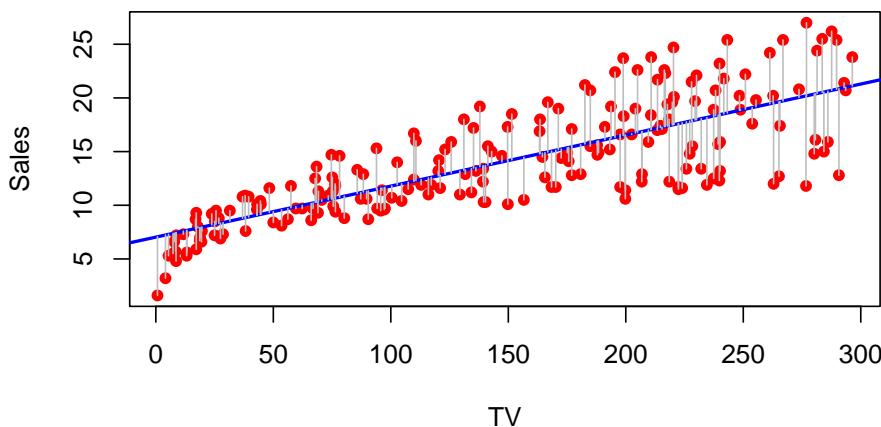


Recall the Advertising data from Chapter 2. Figure 2.1 displays sales (in thousands of units) for a particular product as a function of advertising budgets (in thousands of dollars) for TV, radio, and newspaper media. Suppose that in our role as statistical consultants we are asked to suggest, on the basis of this data, a marketing plan for next year that will result in high product sales. What information would be useful in order to provide such a recommendation? Here are a few important questions that we might seek to address:

1. Is there a relationship between advertising budget and sales?
2. How strong is the relationship between advertising budget and sales?
3. Which media contribute to sales?
4. How accurately can we estimate the effect of each medium on sales?

For the Advertising data, the least squares fit for the regression of sales onto TV is shown. The fit is found by minimizing the sum of squared errors. Each grey line segment represents an error, and the fit makes a compromise by averaging their squares. In this case a linear fit captures the essence of the relationship, although it is somewhat deficient in the left of the plot.

```
tv_model <- lm(sales ~ TV, data = advertising)
plot(advertising$TV, advertising$sales, xlab = "TV", ylab = "Sales",
     col = "red", pch=16)
abline(tv_model, col = "blue", lwd=2)
segments(advertising$TV, advertising$sales, advertising$TV, predict(tv_model),
         col = "gray")
```



```
smry <- summary(tv_model)
smry
```

```
#>
#> Call:
#> lm(formula = sales ~ TV, data = advertising)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -8.386 -1.955 -0.191  2.067  7.212
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) 7.03259   0.45784   15.4 <2e-16 ***
#> TV          0.04754   0.00269   17.7 <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 3.26 on 198 degrees of freedom
#> Multiple R-squared:  0.612, Adjusted R-squared:  0.61
#> F-statistic: 312 on 1 and 198 DF, p-value: <2e-16
```

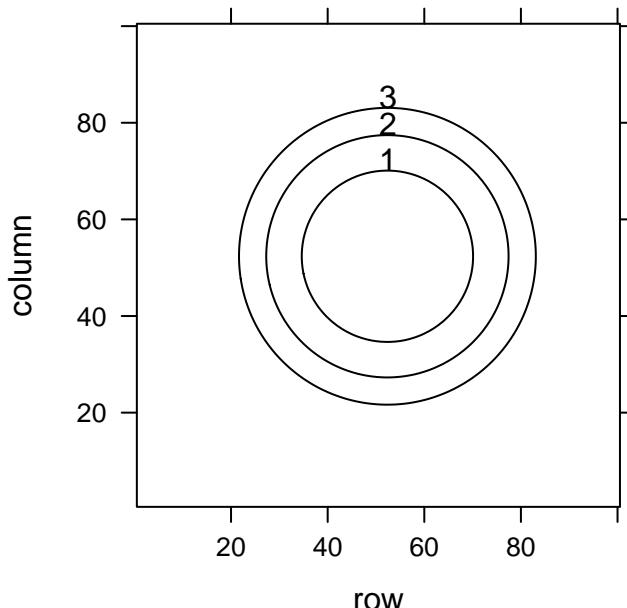
```
library(lattice)

minRss <- sqrt(abs(min(smry$residuals))) * sign(min(smry$residuals))
maxRss <- sqrt(max(smry$residuals))

twovar <- function(x, y) {
  x^2 + y^2 }

mat <- outer( seq(minRss, maxRss, length = 100),
              seq(minRss, maxRss, length = 100),
              Vectorize( function(x,y) twovar(x, y) ) )

contourplot(mat, at = c(1,2,3))
```



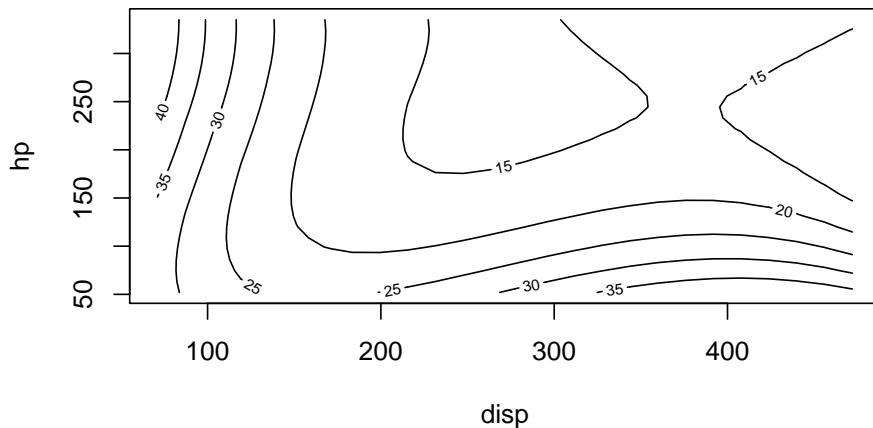
```

tv_model
#>
#> Call:
#> lm(formula = sales ~ TV, data = advertising)
#>
#> Coefficients:
#> (Intercept)          TV
#>    7.0326      0.0475

tv.lm <- lm(sales ~ poly(sales, TV, degree=2), data = advertising)
# contour(tv.lm, sales ~ TV)

library(rsm)
mpg.lm <- lm(mpg ~ poly(hp, disp, degree = 3), data = mtcars)
contour(mpg.lm, hp ~ disp)

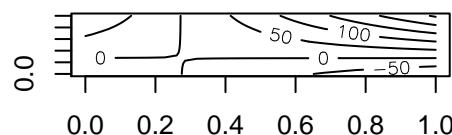
```



```

x <- -6:16
op <- par(mfrow = c(2, 2))
contour(outer(x, x), method = "flattest", vfont = c("sans serif", "plain"))

```



Chapter 48

Lab 3A: Regression. iris dataset

48.1 Introduction

<https://www.matthewrenze.com/workshops/practical-machine-learning-with-r/lab-3a-regression.html>

48.2 Explore the Data

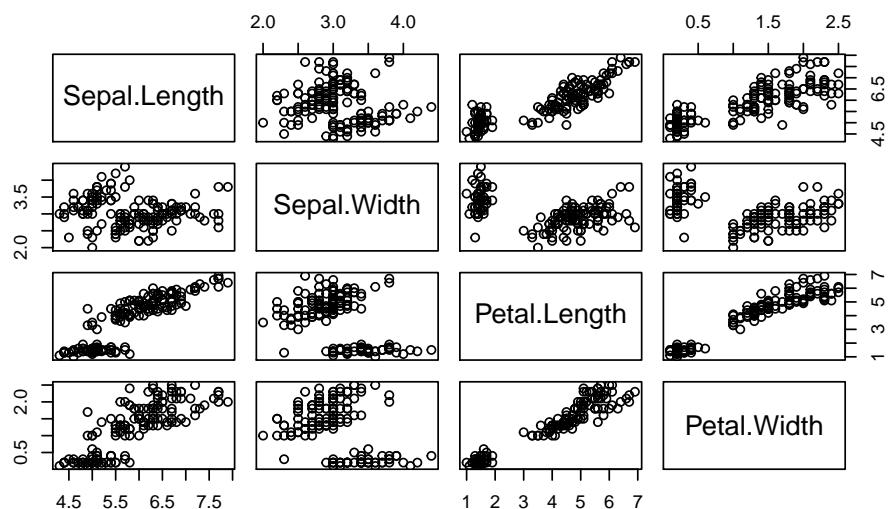
1. Load Iris data
2. Plot scatterplot
3. Plot correlogram

```
data(iris)
```

```
write.csv(iris, file.path(data_raw_dir, "iris.csv"))
```

Create scatterplot matrix

```
plot(iris[1:4])
```



```
library(corrgram)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
```

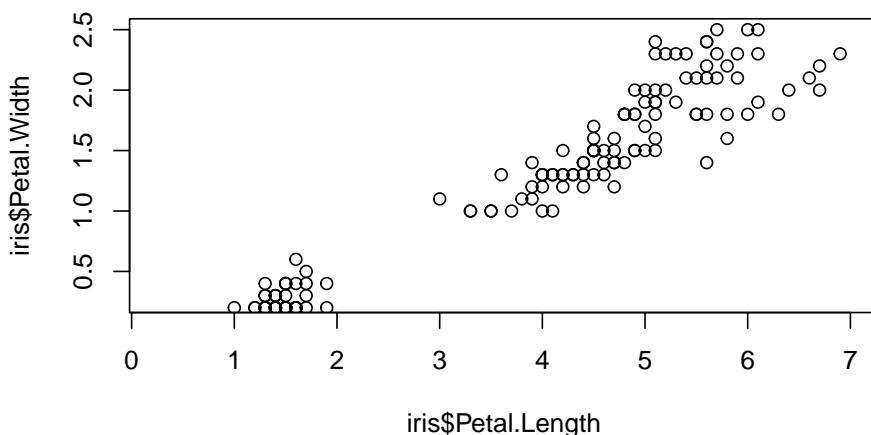
```
#> [.quosures     rlang
#> c.quosures     rlang
#> print.quosures rlang
#> Registered S3 method overwritten by 'seriation':
#>   method      from
#>   reorder.hclust gclus
corrgram(iris[1:4])
```



```
cor(iris[1:4])
#>           Sepal.Length Sepal.Width Petal.Length Petal.Width
#> Sepal.Length     1.000      -0.118     0.872      0.818
#> Sepal.Width      -0.118      1.000     -0.428     -0.366
#> Petal.Length      0.872     -0.428      1.000      0.963
#> Petal.Width       0.818     -0.366      0.963      1.000
```

```
cor(
  x = iris$Petal.Length,
  y = iris$Petal.Width)
#> [1] 0.963
```

```
plot(
  x = iris$Petal.Length,
  y = iris$Petal.Width,
  xlim = c(0.25, 7),
  ylim = c(0.25, 2.5))
```



48.3 Create Training and Test Sets

```
set.seed(42)

indexes <- sample(
  x = 1:150,
  size = 100)

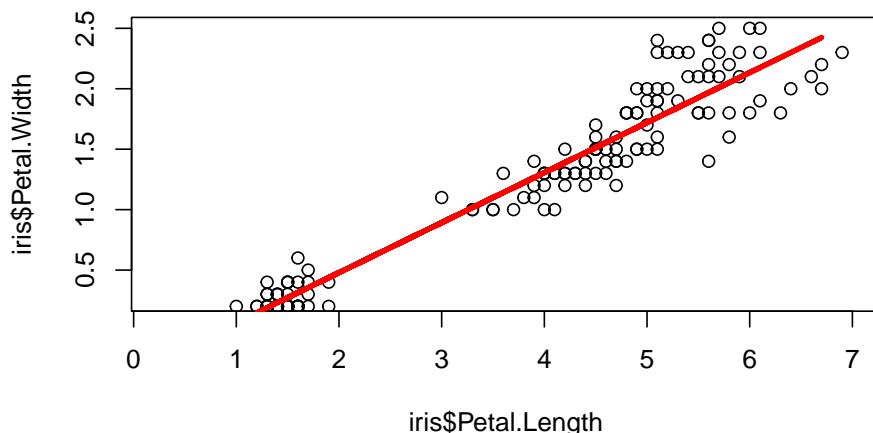
train <- iris[indexes, ]
test <- iris[-indexes, ]
```

48.4 Predict with Simple Linear Regression

```
simpleModel <- lm(
  formula = Petal.Width ~ Petal.Length,
  data = train)

plot(
  x = iris$Petal.Length,
  y = iris$Petal.Width,
  xlim = c(0.25, 7),
  ylim = c(0.25, 2.5))

lines(
  x = train$Petal.Length,
  y = simpleModel$fitted,
  col = "red",
  lwd = 3)
```



```
summary(simpleModel)
#>
#> Call:
#> lm(formula = Petal.Width ~ Petal.Length, data = train)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -0.5684 -0.1279 -0.0307  0.1280  0.6385
#>
```

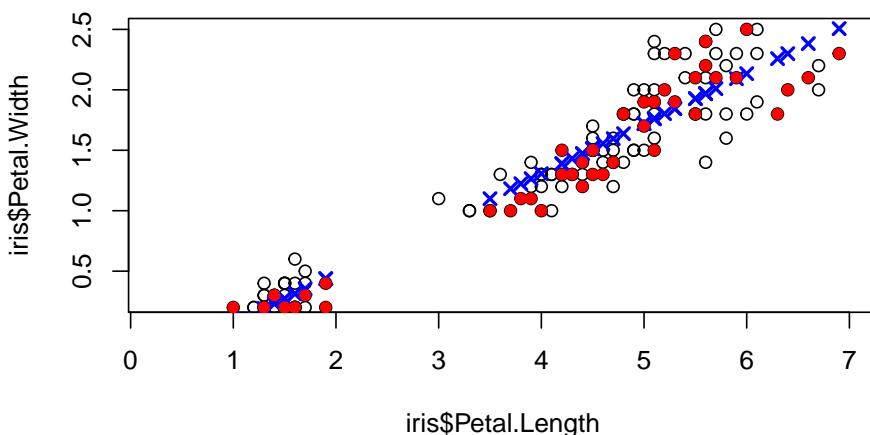
```
#> Coefficients:
#>
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) -0.3486     0.0476   -7.33  6.7e-11 ***
#> Petal.Length  0.4137     0.0119   34.80  < 2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.209 on 98 degrees of freedom
#> Multiple R-squared:  0.925, Adjusted R-squared:  0.924
#> F-statistic: 1.21e+03 on 1 and 98 DF,  p-value: <2e-16

simplePredictions <- predict(
  object = simpleModel,
  newdata = test)

plot(
  x = iris$Petal.Length,
  y = iris$Petal.Width,
  xlim = c(0.25, 7),
  ylim = c(0.25, 2.5))

points(
  x = test$Petal.Length,
  y = simplePredictions,
  col = "blue",
  pch = 4,
  lwd = 2)

points(
  x = test$Petal.Length,
  y = test$Petal.Width,
  col = "red",
  pch = 16)
```



```
simpleRMSE <- sqrt(mean((test$Petal.Width - simplePredictions)^2))
print(simpleRMSE)
#> [1] 0.201
```

48.5 Predict with Multiple Regression

```

multipleModel <- lm(
  formula = Petal.Width ~ .,
  data = train)

summary(multipleModel)
#>
#> Call:
#> lm(formula = Petal.Width ~ ., data = train)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -0.5769 -0.0843 -0.0066  0.0978  0.4731
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) -0.5088    0.2277 -2.23  0.02779 *
#> Sepal.Length -0.0486    0.0593 -0.82  0.41435
#> Sepal.Width   0.2032    0.0594  3.42  0.00092 ***
#> Petal.Length   0.2103    0.0641  3.28  0.00146 **
#> Speciesversicolor  0.6769    0.1583  4.28  4.5e-05 ***
#> Speciesvirginica   1.0762    0.2126  5.06  2.1e-06 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.176 on 94 degrees of freedom
#> Multiple R-squared:  0.949, Adjusted R-squared:  0.947
#> F-statistic: 352 on 5 and 94 DF, p-value: <2e-16

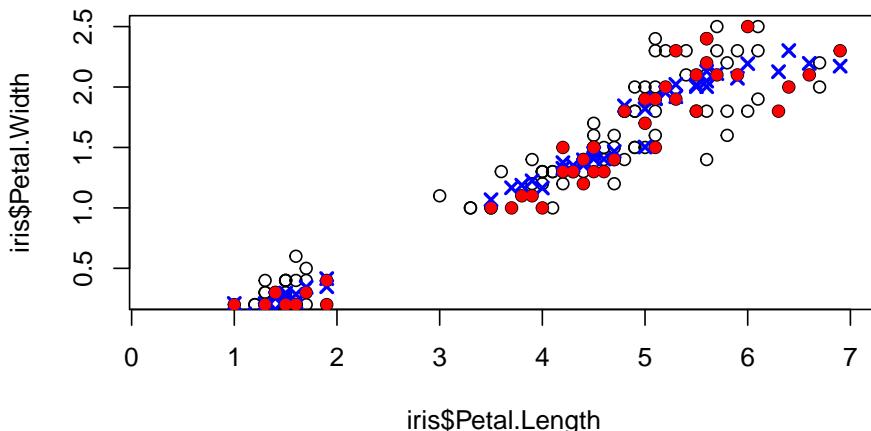
multiplePredictions <- predict(
  object = multipleModel,
  newdata = test)

plot(
  x = iris$Petal.Length,
  y = iris$Petal.Width,
  xlim = c(0.25, 7),
  ylim = c(0.25, 2.5))

points(
  x = test$Petal.Length,
  y = multiplePredictions,
  col = "blue",
  pch = 4,
  lwd = 2)

points(
  x = test$Petal.Length,
  y = test$Petal.Width,
  col = "red",
  pch = 16)

```



```
multipleRMSE <- sqrt(mean((test$Petal.Width - multiplePredictions)^2))
print(multipleRMSE)
#> [1] 0.15
```

48.6 5. Predict with Neural Network Regression

```
normalize <- function(x) {
  (x - min(x)) / (max(x) - min(x)) - 0.5
}

denormalize <- function(x, y) {
  ((x + 0.5) * (max(y) - min(y))) + min(y)
}

scaledIris <- data.frame(
  Sepal.Length = normalize(iris$Sepal.Length),
  Sepal.Width = normalize(iris$Sepal.Width),
  Petal.Length = normalize(iris$Petal.Length),
  Petal.Width = normalize(iris$Petal.Width),
  Species = iris$Species)

scaledTrain <- scaledIris[indexes, ]
scaledTest <- scaledIris[-indexes, ]

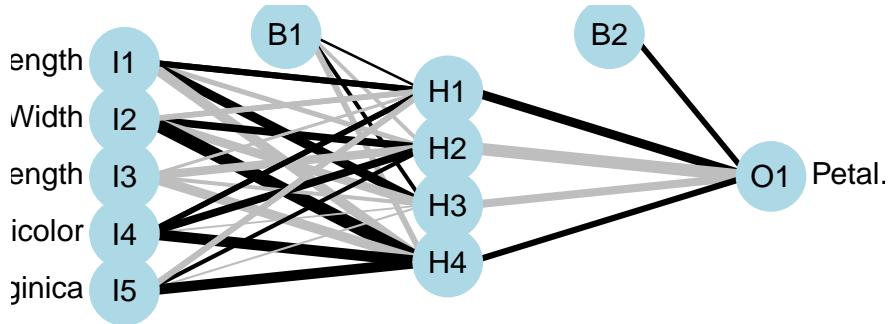
library(nnet)

neuralRegressor <- nnet(
  formula = Petal.Width ~ .,
  data = scaledTrain,
  linout = TRUE,
  skip = TRUE,
  size = 4,
  decay = 0.0001,
  maxit = 500)
#> # weights: 34
#> initial value 64.175158
#> iter 10 value 0.498340
#> iter 20 value 0.439307
#> iter 30 value 0.419373
```

```
#> iter 40 value 0.415119
#> iter 50 value 0.412305
#> iter 60 value 0.410862
#> iter 70 value 0.404854
#> iter 80 value 0.402606
#> iter 90 value 0.397903
#> iter 100 value 0.396295
#> iter 110 value 0.394291
#> iter 120 value 0.392652
#> iter 130 value 0.390227
#> iter 140 value 0.389581
#> iter 150 value 0.388891
#> iter 160 value 0.387501
#> iter 170 value 0.382381
#> iter 180 value 0.377034
#> iter 190 value 0.371871
#> iter 200 value 0.364243
#> iter 210 value 0.357845
#> iter 220 value 0.353726
#> iter 230 value 0.348595
#> iter 240 value 0.345766
#> iter 250 value 0.341638
#> iter 260 value 0.340492
#> iter 270 value 0.339963
#> iter 280 value 0.338600
#> iter 290 value 0.338192
#> iter 300 value 0.336018
#> iter 310 value 0.332364
#> iter 320 value 0.331113
#> iter 330 value 0.330340
#> iter 340 value 0.329913
#> iter 350 value 0.329630
#> iter 360 value 0.329433
#> iter 370 value 0.328969
#> iter 380 value 0.328461
#> iter 390 value 0.327849
#> iter 400 value 0.326887
#> iter 410 value 0.326022
#> iter 420 value 0.325114
#> iter 430 value 0.323672
#> iter 440 value 0.321995
#> iter 450 value 0.320491
#> iter 460 value 0.318875
#> iter 470 value 0.317241
#> iter 480 value 0.316544
#> iter 490 value 0.316008
#> iter 500 value 0.315713
#> final value 0.315713
#> stopped after 500 iterations
```

```
library(NeuralNetTools)
```

```
plotnet(neuralRegressor)
```



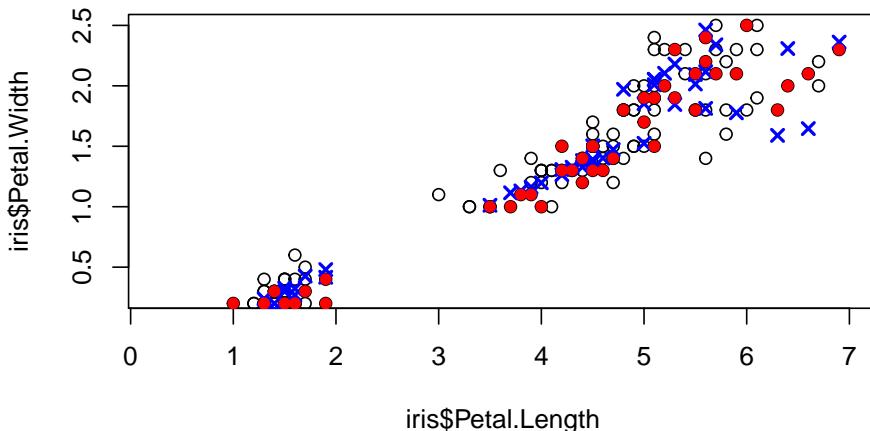
```
scaledPredictions <- predict(  
  object = neuralRegressor,  
  newdata = scaledTest)
```

```
neuralPredictions <- denormalize(  
  x = scaledPredictions,  
  y = iris$Petal.Width)
```

```
plot(  
  x = iris$Petal.Length,  
  y = iris$Petal.Width,  
  xlim = c(0.25, 7),  
  ylim = c(0.25, 2.5))
```

```
points(  
  x = test$Petal.Length,  
  y = neuralPredictions,  
  col = "blue",  
  pch = 4,  
  lwd = 2)
```

```
points(  
  x = test$Petal.Length,  
  y = test$Petal.Width,  
  col = "red",  
  pch = 16)
```



```
neuralRMSE <- sqrt(mean((test$Petal.Width - neuralPredictions)^2))
print(neuralRMSE)
#> [1] 0.183
```

48.7 6. Evaluate all the regression Models

```
print(simpleRMSE)
#> [1] 0.201
print(multipleRMSE)
#> [1] 0.15
print(neuralRMSE)
#> [1] 0.183
```


Chapter 49

Regression 3b. Rates dataset. (*SLR, MLR, NN*)

49.1 Introduction

line 29 does not plot

Source: <https://www.matthewrenze.com/workshops/practical-machine-learning-with-r/lab-3b-regression.html>

```
library(readr)

policies <- read_csv(file.path(data_raw_dir, "Rates.csv"))
#> Parsed with column specification:
#> cols(
#>   Gender = col_character(),
#>   State = col_character(),
#>   State.Rate = col_double(),
#>   Height = col_double(),
#>   Weight = col_double(),
#>   BMI = col_double(),
#>   Age = col_double(),
#>   Rate = col_double()
#> )
policies
#> # A tibble: 1,942 x 8
#>   Gender State State.Rate Height Weight   BMI   Age   Rate
#>   <chr>  <chr>     <dbl>  <dbl>  <dbl>  <dbl>  <dbl>
#> 1 Male    MA        0.100    184    67.8  20.0   77  0.332
#> 2 Male    VA        0.142    163    89.4  33.6   82  0.869
#> 3 Male    NY        0.0908   170    81.2  28.1   31  0.01
#> 4 Male    TN        0.120    175    99.7  32.6   39  0.0215
#> 5 Male    FL        0.110    184    72.1  21.3   68  0.150
#> 6 Male    WA        0.163    166    98.4  35.7   64  0.211
#> # ... with 1,936 more rows

summary(policies)
#>   Gender                  State                 State.Rate          Height
#>   Length:1942              Length:1942            Min.   :0.001      Min.   :150
```

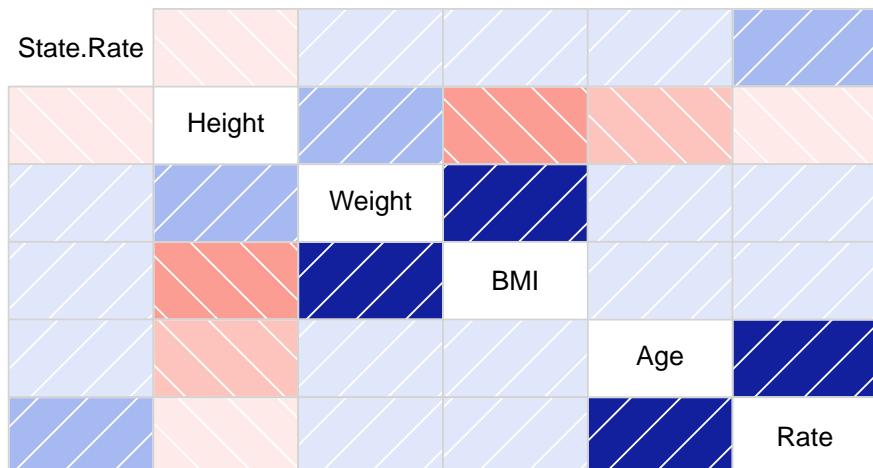
```
#>   Class :character  Class :character  1st Qu.:0.110  1st Qu.:162
#>   Mode  :character  Mode   :character  Median :0.128  Median :170
#>                                                 Mean   :0.138  Mean   :170
#>                                                 3rd Qu.:0.144  3rd Qu.:176
#>                                                 Max.   :0.318  Max.   :190
#> 
#>   Weight          BMI           Age          Rate
#>   Min.   :44.1    Min.   :16.0    Min.   :18.0    Min.   :0.001
#>   1st Qu.:68.6    1st Qu.:23.7    1st Qu.:34.0    1st Qu.:0.015
#>   Median :81.3    Median :28.1    Median :51.0    Median :0.046
#>   Mean   :81.2    Mean   :28.3    Mean   :50.8    Mean   :0.138
#>   3rd Qu.:93.8    3rd Qu.:32.5    3rd Qu.:68.0    3rd Qu.:0.173
#>   Max.   :116.5   Max.   :46.8    Max.   :84.0    Max.   :0.999
```

```
library(RColorBrewer)
palette <- brewer.pal(9, "Reds")
```

```
# plot(
#   x = policies,
#   col = palette[cut(x = policies$Rate, breaks = 9)]
# )
```

```
library(corrgram)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures   rlang
#>   c.quosures   rlang
#>   print.quosures rlang
#> Registered S3 method overwritten by 'seriation':
#>   method      from
#>   reorder.hclust gclus
```

```
corrgram(policies)
```

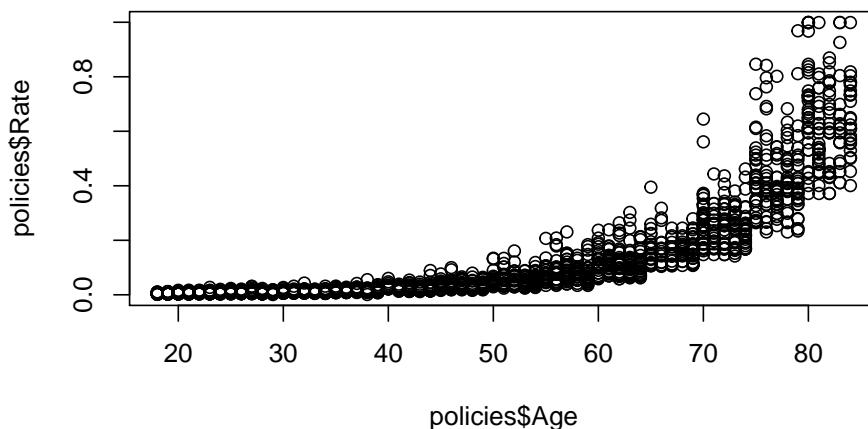


```
cor(policies[3:8])
#> 
#>   State.Rate  Height  Weight       BMI      Age     Rate
#> State.Rate  1.00000 -0.0165  0.00923  0.0192  0.1123  0.2269
#> Height      -0.01652  1.0000  0.23809 -0.3170 -0.1648 -0.1286
#> Weight      0.00923  0.2381  1.00000  0.8396  0.0117  0.0609
#> BMI         0.01924 -0.3170  0.83963  1.0000  0.1023  0.1405
```

```
#> Age          0.11235 -0.1648 0.01168  0.1023  1.0000  0.7801
#> Rate         0.22685 -0.1286 0.06094  0.1405  0.7801  1.0000

cor(
  x = policies$Age,
  y = policies$Rate)
#> [1] 0.78

plot(
  x = policies$Age,
  y = policies$Rate)
```



49.2 Split the Data into Test and Training Sets

```
set.seed(42)

library(caret)
#> Loading required package: lattice
#>
#> Attaching package: 'lattice'
#> The following object is masked from 'package:corrgram':
#>
#>     panel.fill
#> Loading required package: ggplot2

indexes <- createDataPartition(
  y = policies$Rate,
  p = 0.80,
  list = FALSE)

train <- policies[indexes, ]
test <- policies[-indexes, ]

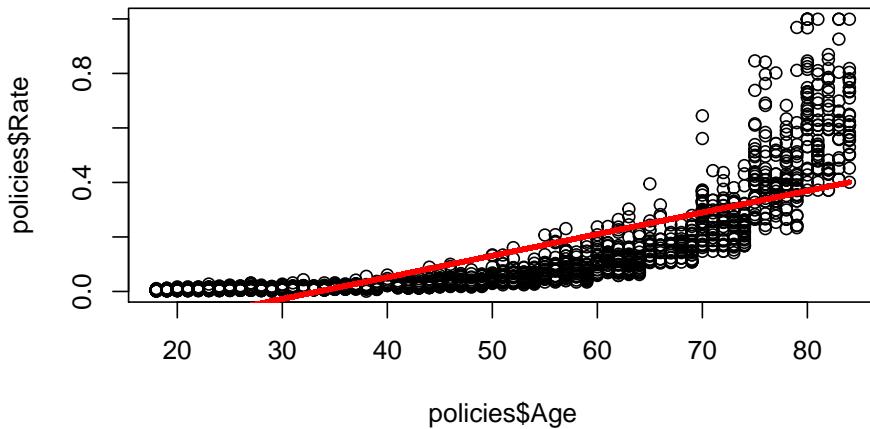
print(nrow(train))
#> [1] 1555
print(nrow(test))
#> [1] 387
```

49.3 Predict with Simple Linear Regression

```
simpleModel <- lm(
  formula = Rate ~ Age,
  data = train)
```

```
plot(
  x = policies$Age,
  y = policies$Rate)
```

```
lines(
  x = train$Age,
  y = simpleModel$fitted,
  col = "red",
  lwd = 3)
```

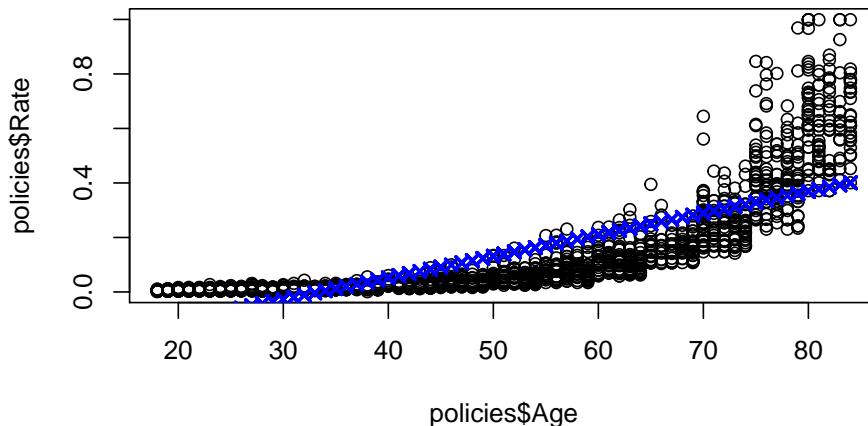


```
summary(simpleModel)
#>
#> Call:
#> lm(formula = Rate ~ Age, data = train)
#>
#> Residuals:
#>    Min      1Q  Median      3Q     Max
#> -0.1799 -0.0881 -0.0208  0.0617  0.6300
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) -0.265244  0.008780 -30.2   <2e-16 ***
#> Age          0.007928  0.000161  49.3   <2e-16 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.123 on 1553 degrees of freedom
#> Multiple R-squared:  0.61,  Adjusted R-squared:  0.609
#> F-statistic: 2.43e+03 on 1 and 1553 DF,  p-value: <2e-16
```

```
simplePredictions <- predict(
  object = simpleModel,
  newdata = test)
```

```
plot(
  x = policies$Age,
  y = policies$Rate)

points(
  x = test$Age,
  y = simplePredictions,
  col = "blue",
  pch = 4,
  lwd = 2)
```



```
simpleRMSE <- sqrt(mean((test$Rate - simplePredictions)^2))
print(simpleRMSE)
#> [1] 0.119
```

49.4 Predict with Multiple Linear Regression

```
multipleModel <- lm(
  formula = Rate ~ Age + Gender + State.Rate + BMI,
  data = train)

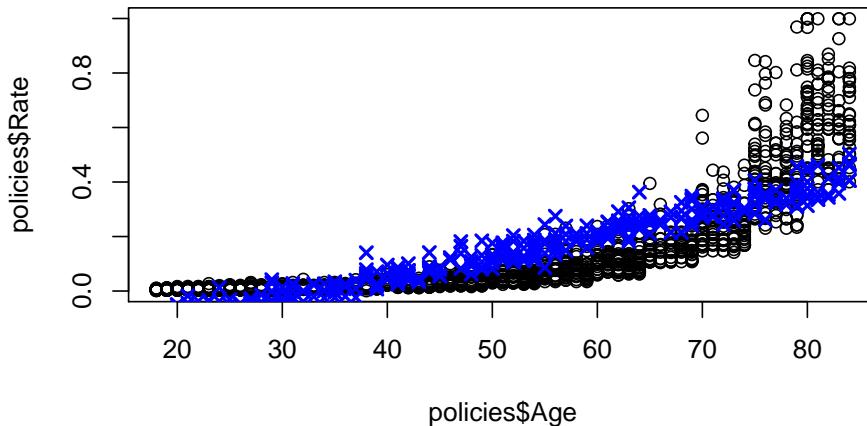
summary(multipleModel)
#>
#> Call:
#> lm(formula = Rate ~ Age + Gender + State.Rate + BMI, data = train)
#>
#> Residuals:
#>      Min       1Q   Median       3Q      Max
#> -0.2255 -0.0865 -0.0292  0.0590  0.6053
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept) -0.428141  0.018742 -22.84 < 2e-16 ***
#> Age          0.007703  0.000156  49.28 < 2e-16 ***
#> GenderMale   0.030350  0.006001   5.06 4.8e-07 ***
#> State.Rate   0.613139  0.068330   8.97 < 2e-16 ***
#> BMI          0.002634  0.000518   5.09 4.1e-07 ***
```

```
#> ---
#> Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 0.118 on 1550 degrees of freedom
#> Multiple R-squared: 0.64, Adjusted R-squared: 0.639
#> F-statistic: 688 on 4 and 1550 DF, p-value: <2e-16

multiplePredictions <- predict(
  object = multipleModel,
  newdata = test)

plot(
  x = policies$Age,
  y = policies$Rate)

points(
  x = test$Age,
  y = multiplePredictions,
  col = "blue",
  pch = 4,
  lwd = 2)
```



```
multipleRMSE <- sqrt(mean((test$Rate - multiplePredictions)^2))
print(multipleRMSE)
#> [1] 0.114
```

49.5 Predict with Neural Network Regression

```
normalize <- function(x) {
  (x - min(x)) / (max(x) - min(x)) - 0.5
}

denormalize <- function(x, y) {
  ((x + 0.5) * (max(y) - min(y))) + min(y)
}

scaledPolicies <- data.frame(
  Gender = policies$Gender,
  State.Rate = normalize(policies$State.Rate),
```

```
BMI = normalize(policies$BMI),  
Age = normalize(policies$Age),  
Rate = normalize(policies$Rate))
```

```
scaledTrain <- scaledPolicies[indexes, ]  
scaledTest <- scaledPolicies[-indexes, ]
```

```
library(nnet)
```

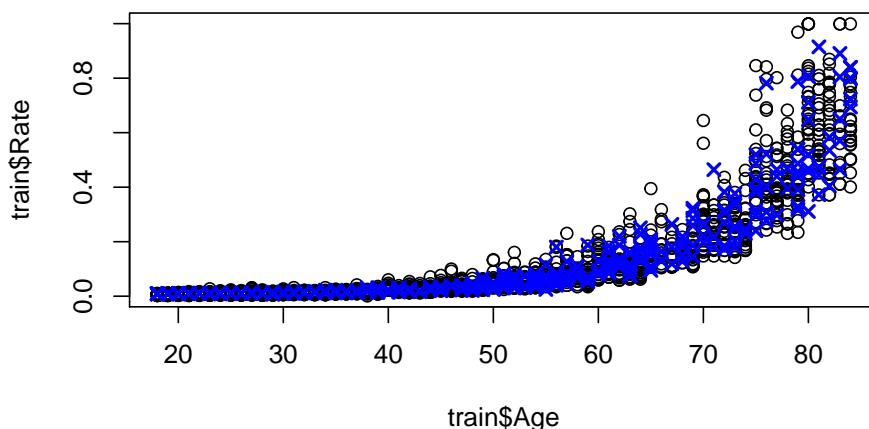
```
neuralRegressor <- nnet(  
  formula = Rate ~ .,  
  data = scaledTrain,  
  linout = TRUE,  
  size = 5,  
  decay = 0.0001,  
  maxit = 1000)  
#> # weights:  31  
#> initial  value 548.090539  
#> iter   10 value 10.610284  
#> iter   20 value 3.927378  
#> iter   30 value 3.735266  
#> iter   40 value 3.513899  
#> iter   50 value 3.073390  
#> iter   60 value 2.547202  
#> iter   70 value 2.296126  
#> iter   80 value 2.166120  
#> iter   90 value 2.106996  
#> iter  100 value 2.092654  
#> iter  110 value 2.058596  
#> iter  120 value 2.039404  
#> iter  130 value 2.023721  
#> iter  140 value 2.018781  
#> iter  150 value 2.006931  
#> iter  160 value 1.999122  
#> iter  170 value 1.993920  
#> iter  180 value 1.990678  
#> iter  190 value 1.989269  
#> iter  200 value 1.988846  
#> iter  210 value 1.988042  
#> iter  220 value 1.987739  
#> iter  230 value 1.987678  
#> iter  240 value 1.987598  
#> iter  250 value 1.987574  
#> iter  260 value 1.987549  
#> iter  270 value 1.987536  
#> iter  280 value 1.987529  
#> final  value 1.987526  
#> converged
```

```
scaledPredictions <- predict(  
  object = neuralRegressor,  
  newdata = scaledTest)
```

```
neuralPredictions <- denormalize(
  x = scaledPredictions,
  y = policies$Rate)
```

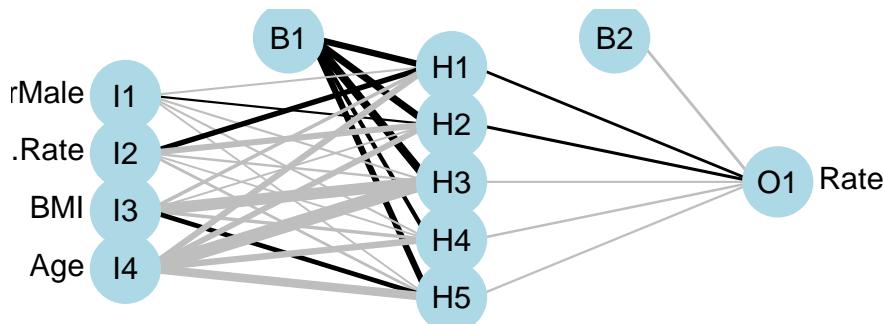
```
plot(
  x = train$Age,
  y = train$Rate)
```

```
points(
  x = test$Age,
  y = neuralPredictions,
  col = "blue",
  pch = 4,
  lwd = 2)
```



```
library(NeuralNetTools)

plotnet(neuralRegressor)
```



```
neuralRMSE <- sqrt(mean((test$Rate - neuralPredictions)^2))
print(neuralRMSE)
#> [1] 0.0368
```

49.6 Evaluate the Regression Models

```
print(simpleRMSE)
#> [1] 0.119
```

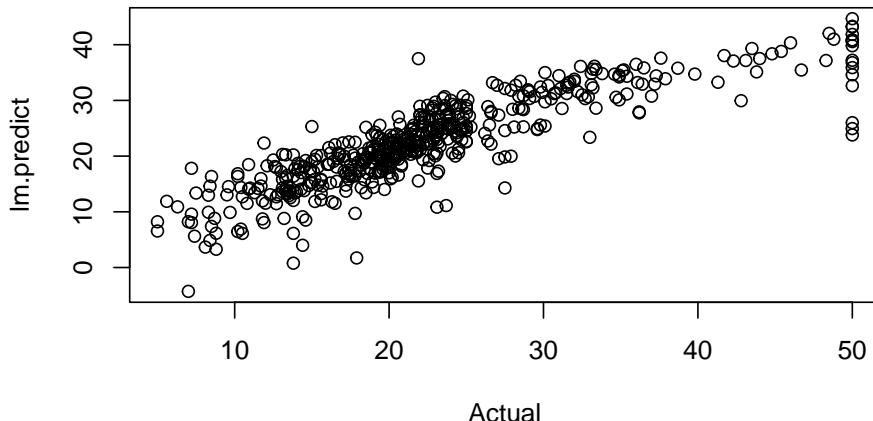
```
print(multipleRMSE)
#> [1] 0.114
print(neuralRMSE)
#> [1] 0.0368
```


Chapter 50

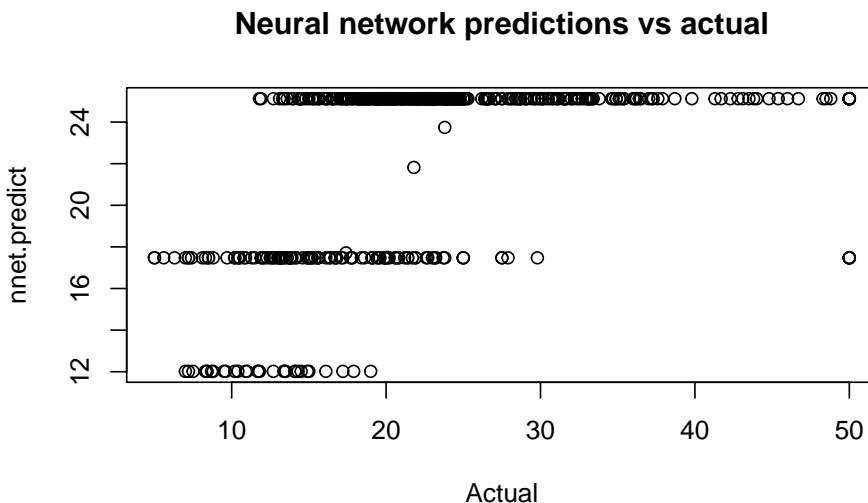
Regression Boston nnet

```
###  
### prepare data  
###  
library(mlbench)  
data(BostonHousing)  
  
# inspect the range which is 1-50  
summary(BostonHousing$medv)  
#>   Min. 1st Qu. Median     Mean 3rd Qu.    Max.  
#>   5.0    17.0   21.2    22.5   25.0    50.0  
  
##  
## model linear regression  
##  
  
lm.fit <- lm(medv ~ ., data=BostonHousing)  
  
lm.predict <- predict(lm.fit)  
  
# mean squared error: 21.89483  
mean((lm.predict - BostonHousing$medv)^2)  
#> [1] 21.9  
  
plot(BostonHousing$medv, lm.predict,  
      main="Linear regression predictions vs actual",  
      xlab="Actual")
```

Linear regression predictions vs actual



```
##  
## model neural network  
##  
require(nnet)  
#> Loading required package: nnet  
  
# scale inputs: divide by 50 to get 0-1 range  
nnet.fit <- nnet(medv/50 ~ ., data=BostonHousing, size=2)  
#> # weights: 31  
#> initial value 17.039194  
#> iter 10 value 13.754559  
#> iter 20 value 13.537235  
#> iter 30 value 13.537183  
#> iter 40 value 13.530522  
#> final value 13.529736  
#> converged  
  
# multiply 50 to restore original scale  
nnet.predict <- predict(nnet.fit)*50  
  
# mean squared error: 16.40581  
mean((nnet.predict - BostonHousing$medv)^2)  
#> [1] 66.8  
  
plot(BostonHousing$medv, nnet.predict,  
     main="Neural network predictions vs actual",  
     xlab="Actual")
```



50.1 Neural Network

Now, let's use the function `train()` from the package `caret` to optimize the neural network hyperparameters decay and size. Also, `caret` performs resampling to give a better estimate of the error. In this case we scale linear regression by the same value, so the error statistics are directly comparable.

```
library(mlbench)
data(BostonHousing)

require(caret)
#> Loading required package: caret
#> Loading required package: lattice
#> Loading required package: ggplot2
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures    rlang
#>   c.quosures    rlang
#>   print.quosures rlang

mygrid <- expand.grid(.decay=c(0.5, 0.1), .size=c(4,5,6))
nnetfit <- train(medv/50 ~ ., data=BostonHousing, method="nnet", maxit=1000, tuneGrid=mygrid, trace=F)
print(nnetfit)
#> Neural Network
#>
#> 506 samples
#> 13 predictor
#>
#> No pre-processing
#> Resampling: Bootstrapped (25 reps)
#> Summary of sample sizes: 506, 506, 506, 506, 506, 506, ...
#> Resampling results across tuning parameters:
#>
#>   decay  size   RMSE   Rsquared   MAE
#>   0.1    4     0.0835  0.787     0.0571
#>   0.1    5     0.0822  0.794     0.0565
#>   0.1    6     0.0799  0.806     0.0544
```

```
#>   0.5    4    0.0908  0.757    0.0626
#>   0.5    5    0.0900  0.761    0.0624
#>   0.5    6    0.0895  0.763    0.0622
#>
#> RMSE was used to select the optimal model using the smallest value.
#> The final values used for the model were size = 6 and decay = 0.1.
```

506 samples
13 predictors

No pre-processing
Resampling: Bootstrap (25 reps)

Summary of sample sizes: 506, 506, 506, 506, 506, 506, ...

Resampling results across tuning parameters:

size	decay	RMSE	Rsquared	RMSE SD	Rsquared SD
4	0.1	0.0852	0.785	0.00863	0.0406
4	0.5	0.0923	0.753	0.00891	0.0436
5	0.1	0.0836	0.792	0.00829	0.0396
5	0.5	0.0899	0.765	0.00858	0.0399
6	0.1	0.0835	0.793	0.00804	0.0318
6	0.5	0.0895	0.768	0.00789	0.0344

50.2 Linear Regression

```
lmfit <- train(medv/50 ~ ., data=BostonHousing, method="lm")
print(lmfit)
#> Linear Regression
#>
#> 506 samples
#> 13 predictor
#>
#> No pre-processing
#> Resampling: Bootstrapped (25 reps)
#> Summary of sample sizes: 506, 506, 506, 506, 506, 506, ...
#> Resampling results:
#>
#>   RMSE     Rsquared   MAE
#>   0.0988   0.726     0.0692
#>
#> Tuning parameter 'intercept' was held constant at a value of TRUE
```

506 samples
13 predictors

No pre-processing
Resampling: Bootstrap (25 reps)

Summary of sample sizes: 506, 506, 506, 506, 506, 506, ...

Resampling results

RMSE	Rsquared	RMSE SD	Rsquared SD
0.0994	0.703	0.00741	0.0389

A tuned neural network has a RMSE of 0.0835 compared to linear regression's RMSE of 0.0994.

Chapter 51

Comparing Multiple vs. Neural Network Regression

51.1 Introduction

Source: <http://beyondvalence.blogspot.com/2014/04/r-comparing-multiple-and-neural-network.html>

Here we will compare and evaluate the results from multiple regression and a neural network on the diamonds data set from the `ggplot2` package in R. Consisting of 53,940 observations with 10 variables, diamonds contains data on the carat, cut, color, clarity, price, and diamond dimensions. These variables have a particular effect on price, and we would like to see if they can predict the price of various diamonds.

```
library(ggplot2)
#> Registered S3 methods overwritten by 'ggplot2':
#>   method      from
#>   [.quosures    rlang
#>   c.quosures    rlang
#>   print.quosures rlang
library(RSNNS)
#> Loading required package: Rcpp
library(MASS)
library(caret)
#> Loading required package: lattice
#>
#> Attaching package: 'caret'
#> The following objects are masked from 'package:RSNNS':
#>
#>   confusionMatrix, train
# library(diamonds)

head(diamonds)
#> # A tibble: 6 x 10
#>   carat  cut   color clarity depth table price     x     y     z
#>   <dbl> <ord> <ord> <ord> <dbl> <dbl> <int> <dbl> <dbl> <dbl>
#> 1 0.23  Ideal    E    SI2     61.5    55   326  3.95  3.98  2.43
#> 2 0.21  Premium  E    SI1     59.8    61   326  3.89  3.84  2.31
#> 3 0.23  Good     E    VS1     56.9    65   327  4.05  4.07  2.31
#> 4 0.290 Premium  I    VS2     62.4    58   334  4.2    4.23  2.63
```

```
#> 5 0.31 Good      J      SI2      63.3     58    335  4.34  4.35  2.75
#> 6 0.24 Very Good J      VVS2     62.8     57    336  3.94  3.96  2.48
```

```
dplyr::glimpse(diamonds)
#> Observations: 53,940
#> Variables: 10
#> $ carat    <dbl> 0.23, 0.21, 0.23, 0.29, 0.31, 0.24, 0.24, 0.26, 0.22, ...
#> $ cut       <ord> Ideal, Premium, Good, Premium, Good, Very Good, Very G...
#> $ color     <ord> E, E, E, I, J, J, H, E, H, J, J, F, J, E, E, I, J, ...
#> $ clarity   <ord> SI2, SI1, VS1, VS2, SI2, VVS2, VVS1, SI1, VS2, VS1, SI...
#> $ depth     <dbl> 61.5, 59.8, 56.9, 62.4, 63.3, 62.8, 62.3, 61.9, 65.1, ...
#> $ table     <dbl> 55, 61, 65, 58, 58, 57, 57, 55, 61, 61, 55, 56, 61, 54...
#> $ price     <int> 326, 326, 327, 334, 335, 336, 336, 337, 337, 338, 339, ...
#> $ x         <dbl> 3.95, 3.89, 4.05, 4.20, 4.34, 3.94, 3.95, 4.07, 3.87, ...
#> $ y         <dbl> 3.98, 3.84, 4.07, 4.23, 4.35, 3.96, 3.98, 4.11, 3.78, ...
#> $ z         <dbl> 2.43, 2.31, 2.31, 2.63, 2.75, 2.48, 2.47, 2.53, 2.49, ...
```

The cut, color, and clarity variables are factors, and must be treated as dummy variables in multiple and neural network regressions. Let us start with multiple regression.

51.2 Multiple Regression

First we ready a Multiple Regression by sampling the rows to randomize the observations, and then create a sample index of 0's and 1's to separate the training and test sets. Note that the depth and table columns (5, 6) are removed because they are linear combinations of the dimensions, x, y, and z. See that the observations in the training and test sets approximate 70% and 30% of the total observations, from which we sampled and set the probabilities.

```
set.seed(1234567)
diamonds <- diamonds[sample(1:nrow(diamonds), nrow(diamonds)),]
d.index = sample(0:1, nrow(diamonds), prob=c(0.3, 0.7), rep = TRUE)
d.train <- diamonds[d.index==1, c(-5,-6)]
d.test <- diamonds[d.index==0, c(-5,-6)]
dim(d.train)
#> [1] 37502     8
dim(d.test)
#> [1] 16438     8
```

Now we move into the next stage with multiple regression via the `train()` function from the `caret` library, instead of the regular `lm()` function. We specify the predictors, the response variable (`price`), the “lm” method, and the cross validation resampling method.

```
x <- d.train[,-5]
y <- as.numeric(d.train[,5]$price)

ds.lm <- caret::train(x, y, method = "lm",
                      trainControl = trainControl(method = "cv"))
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>   extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :
#>   extra argument 'trainControl' will be disregarded
#> Warning: Setting row names on a tibble is deprecated.
```



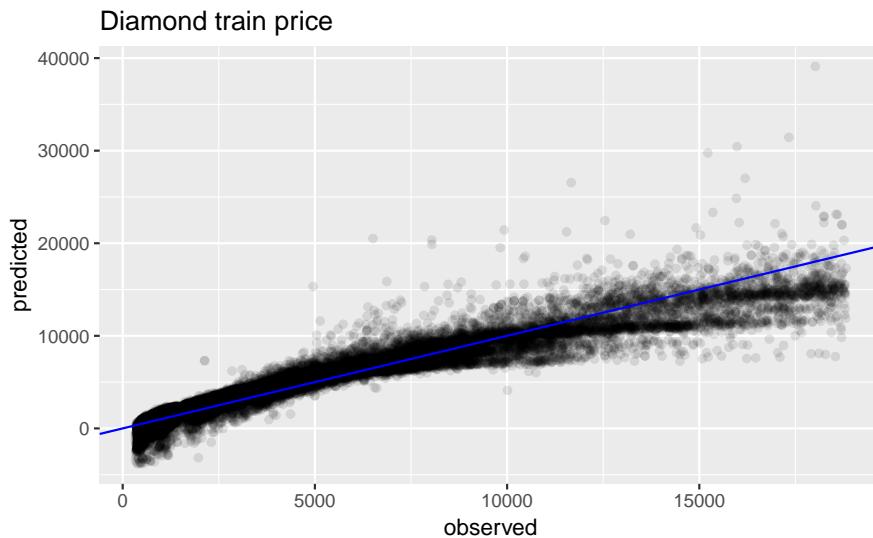
```
#> Warning: Setting row names on a tibble is deprecated.  
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :  
#>   extra argument 'trainControl' will be disregarded  
#> Warning: Setting row names on a tibble is deprecated.  
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :  
#>   extra argument 'trainControl' will be disregarded  
#> Warning: Setting row names on a tibble is deprecated.  
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :  
#>   extra argument 'trainControl' will be disregarded  
#> Warning: Setting row names on a tibble is deprecated.  
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :  
#>   extra argument 'trainControl' will be disregarded  
#> Warning: Setting row names on a tibble is deprecated.  
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :  
#>   extra argument 'trainControl' will be disregarded  
#> Warning: Setting row names on a tibble is deprecated.  
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :  
#>   extra argument 'trainControl' will be disregarded  
#> Warning: Setting row names on a tibble is deprecated.  
#> Warning: In lm.fit(x, y, offset = offset, singular.ok = singular.ok, ...) :  
#>   extra argument 'trainControl' will be disregarded  
ds.lm  
#> Linear Regression  
#>  
#> 37502 samples  
#>     7 predictor  
#>  
#> No pre-processing  
#> Resampling: Bootstrapped (25 reps)  
#> Summary of sample sizes: 37502, 37502, 37502, 37502, 37502, 37502, ...  
#> Resampling results:  
#>  
#>   RMSE  Rsquared  MAE  
#>   1140    0.919    745  
#>  
#> Tuning parameter 'intercept' was held constant at a value of TRUE
```

When we call the train(ed) object, we can see the attributes of the training set, resampling, sample sizes, and the results. Note the root mean square error value of 1150. Will that be low enough to take down heavy weight TEAM: Neural Network? Below we visualize the training diamond prices and the predicted prices with `ggplot()`.

```
library(dplyr)
#>
#> Attaching package: 'dplyr'
#> The following object is masked from 'package:MASS':
#>
#>     select
#> The following objects are masked from 'package:stats':
#>
#>     filter, lag
#> The following objects are masked from 'package:base':
#>
#>     intersect, setdiff, setequal, union

data.frame(obs = y, pred = ds.lm$finalModel$fitted.values) %>%
  ggplot(aes(x = obs, y = pred)) +
```

```
geom_point(alpha=0.1) +
geom_abline(color="blue") +
labs(title="Diamond train price", x="observed", y="predicted")
```

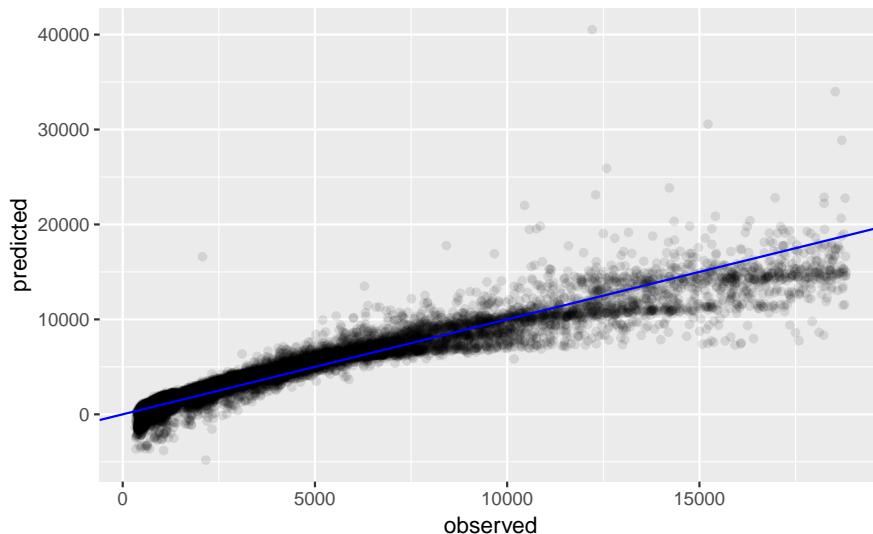


We see from the axis, the predicted prices have some high values compared to the actual prices. Also, there are predicted prices below 0, which cannot be possible in the observed, which will set TEAM: Multiple Regression back a few points.

Next we use `ggplot()` again to visualize the predicted and observed diamond prices from the test data, which did not train the linear regression model.

```
# predict on test set
ds.lm.p <- predict(ds.lm, d.test[,-5], type="raw")

# compare observed vs predicted prices in the test set
data.frame(obs = d.test[,5]$price, pred = ds.lm.p) %>%
  ggplot(aes(x = obs, y = pred)) +
  geom_point(alpha=0.1) +
  geom_abline(color="blue")+
  labs("Diamond Test Price", x="observed", y="predicted")
```



Similar to the training prices plot, we see here in the test prices that the model over predicts larger values and also predicted negative price values. In order for the Multiple Regression to win, the Neural Network has to have more wild prediction values.

Lastly, we calculate the root mean square error, by taking the mean of the squared difference between the predicted and observed diamond prices. The resulting RMSE is 1110.843, similar to the RMSE of the training set.

```
ds.lm.mse <- (1 / nrow(d.test)) * sum((ds.lm.p - d.test[,5])^2)
lm.rmse <- sqrt(ds.lm.mse)
lm.rmse
#> [1] 1168
```

Below is a detailed output of the model summary, with the coefficients and residuals. Observe how carat is the best predictor, with the highest t value at 191.7, with every increase in 1 carat holding all other variables equal, results in a 10,873 dollar increase in value. As we look at the factor variables, we do not see a reliable increase in coefficients with increases in level value.

```
summary(ds.lm)
#>
#> Call:
#> lm(formula = .outcome ~ ., data = dat, trainControl = ..1)
#>
#> Residuals:
#>    Min     1Q Median     3Q    Max
#> -21090   -598   -183    378  10778
#>
#> Coefficients:
#>             Estimate Std. Error t value Pr(>|t|)
#> (Intercept)  3.68      94.63   0.04   0.9690
#> carat       11142.68    57.43 194.02 < 2e-16 ***
#> cut.L        767.70    24.31  31.58 < 2e-16 ***
#> cut.Q       -336.63    21.41 -15.72 < 2e-16 ***
#> cut.C        157.31    18.81   8.36 < 2e-16 ***
#> cut^4       -22.81    14.78  -1.54  0.1228
#> color.L     -1950.28   20.66 -94.42 < 2e-16 ***
#> color.Q     -665.60    18.82 -35.37 < 2e-16 ***
#> color.C     -147.16    17.61  -8.36 < 2e-16 ***
#> color^4      44.64    16.20   2.76  0.0059 **
#> color^5     -91.21    15.32  -5.95  2.7e-09 ***
#> color^6     -54.74    13.92  -3.93  8.5e-05 ***
#> clarity.L    4115.45   36.68 112.19 < 2e-16 ***
#> clarity.Q   -1959.71   34.33 -57.09 < 2e-16 ***
#> clarity.C    990.60    29.29  33.83 < 2e-16 ***
#> clarity^4   -370.82    23.30 -15.92 < 2e-16 ***
#> clarity^5    240.60    18.91  12.72 < 2e-16 ***
#> clarity^6     -7.99    16.37  -0.49  0.6253
#> clarity^7     80.62    14.48   5.57  2.6e-08 ***
#> x          -1400.26   95.70 -14.63 < 2e-16 ***
#> y            545.42    94.57   5.77  8.1e-09 ***
#> z           -190.86   31.20  -6.12  9.6e-10 ***
#> ---
#> Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
#>
#> Residual standard error: 1130 on 37480 degrees of freedom
#> Multiple R-squared:  0.92,  Adjusted R-squared:  0.92
```

```
#> F-statistic: 2.05e+04 on 21 and 37480 DF, p-value: <2e-16
```

Now we move on to the neural network regression.

51.3 Neural Network

Because neural networks operate in terms of 0 to 1, or -1 to 1, we must first normalize the price variable to 0 to 1, making the lowest value 0 and the highest value 1. We accomplished this using the `normalizeData()` function. Save the price output in order to revert the normalization after training the data. Also, we take the factor variables and turn them into numeric labels using `toNumericClassLabels()`. Below we see the normalized prices before they are split into a training and test set with `splitForTrainingAndTest()` function.

```
diamonds[,3] <- toNumericClassLabels(diamonds[,3]$color)
diamonds[,4] <- toNumericClassLabels(diamonds[,4]$clarity)
prices <- normalizeData(diamonds[,7], type="0_1")
head(prices)
#>      [,1]
#> [1,] 0.0841
#> [2,] 0.1491
#> [3,] 0.0237
#> [4,] 0.3247
#> [5,] 0.0280
#> [6,] 0.0252

dsplit <- splitForTrainingAndTest(diamonds[, c(-2,-5,-6,-7,-9,-10)], prices, ratio=0.3)
```

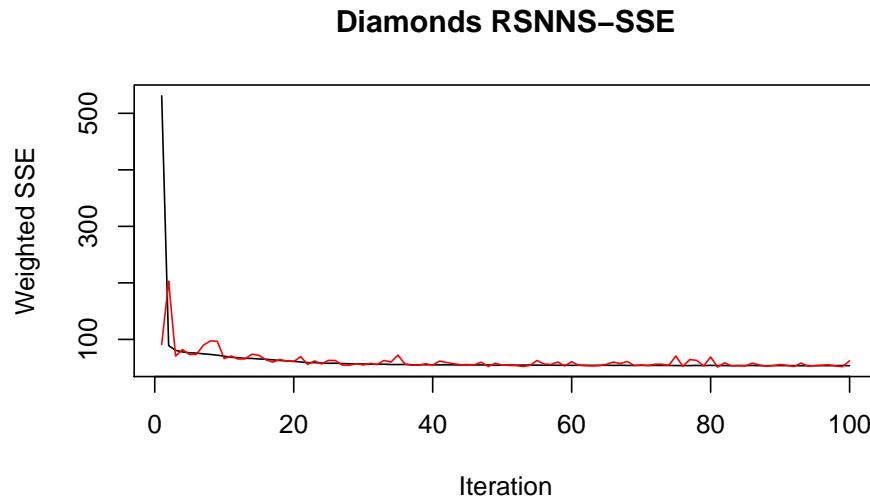
Now the Neural Network are ready for the multi-layer perceptron (MLP) regression. We define the training inputs (predictor variables) and targets (prices), the size of the layer (5), the incremented learning parameter (0.1), the max iterations (100 epochs), and also the test input/targets.

```
# mlp model
d.nn <- mlp(dsplits$inputsTrain,
              dsplits$targetsTrain,
              size = c(5), learnFuncParams = c(0.1), maxit=100,
              inputsTest = dsplits$inputsTest,
              targetsTest = dsplits$targetsTest,
              metric = "RMSE",
              linout = FALSE)
```

If you spectators have dealt with `mlp()` before, you know the summary output can be quite lengthy, so it is omitted (we dislike commercials too). We move to the visual description of the MLP model with the iterative sum of square error for the training and test sets. Additionally, we plot the regression error (predicted vs observed) for the training and test prices.

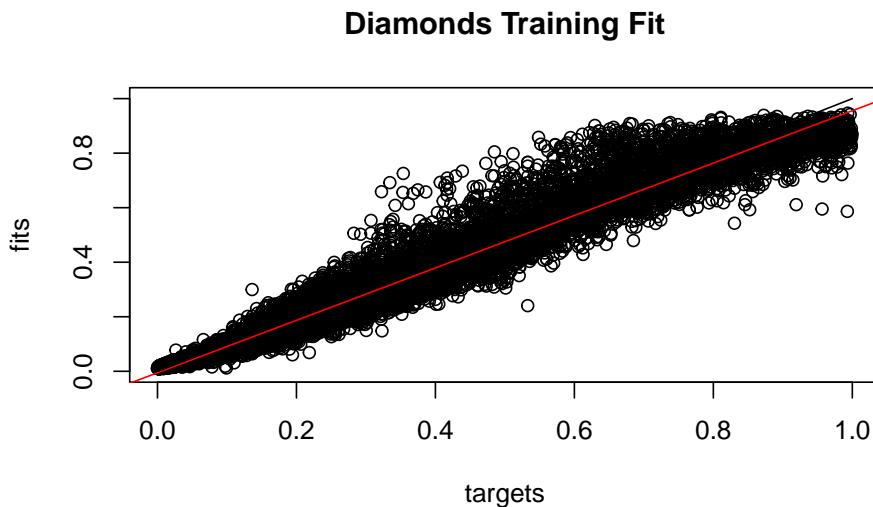
Time for the Neural Network so show off its statistical muscles! First up, we have the iterative sum of square error for each epoch, noting that we specified a maximum of 100 in the MLP model. We see an immediate drop in the SSE with the first few iterations, with the SSE leveling out around 50. The test SSE, in red, fluctuates just above 50 as well. Since the SSE began to plateau, the model fit well but not too well, since we want to avoid over fitting the model. So 100 iterations was a good choice.

```
# SSE error
plotIterativeError(d.nn, main = "Diamonds RSNNS-SSE")
```



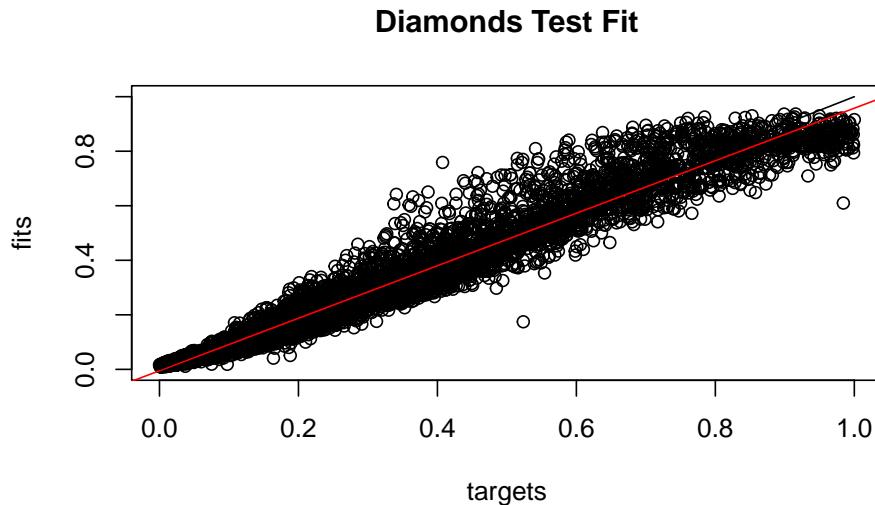
Second, we observe the regression plot with the fitted (predicted) and target (observed) prices from the training set. The prices fit reasonably well, and we see the red model regression line close to the black ($y=x$) optimal line. Note that some middle prices were over predicted by the model, and there were no negative prices, unlike the linear regression model.

```
# regression errors
plotRegressionError(dsplit$targetsTrain, d.nn$fitted.values,
                     main = "Diamonds Training Fit")
```



Third, we look at the predicted and observed prices from the test set. Again the red regression line approximates the optimal black line, and more price values were over predicted by the model. Again, there are no negative predicted prices, a good sign.

```
plotRegressionError(dsplit$targetsTest, d.nn$fittedTestValues,
                     main = "Diamonds Test Fit")
```



Now we calculate the RMSE for the training set, which we get 692.5155. This looks promising for the Neural Network!

```
# train set
train.pred <- denormalizeData(d.nn$fitted.values,
                                getNormParameters(prices))

train.obs <- denormalizeData(dsplits$targetsTrain,
                             getNormParameters(prices))

train.mse <- (1 / nrow(dsplits$inputsTrain)) * sum((train.pred - train.obs)^2)

rsnns.train.rmse <- sqrt(train.mse)
rsnns.train.rmse
#> [1] 739
```

Naturally we want to calculate the RMSE for the test set, but note that in the real world, we would not have the luxury of knowing the real test values. We arrive at 679.5265.

```
# test set
test.pred <- denormalizeData(d.nn$fittedTestValues,
                               getNormParameters(prices))

test.obs <- denormalizeData(dsplits$targetsTest,
                            getNormParameters(prices))

test.mse <- (1 / nrow(dsplits$inputsTest)) * sum((test.pred - test.obs)^2)

rsnns.test.rmse <- sqrt(test.mse)
rsnns.test.rmse
#> [1] 751
```

Which model was better in predicting the diamond price? The linear regression model with 10 fold cross validation, or the multi-layer perceptron model with 5 nodes run to 100 iterations? Who won the rumble?

RUMBLE RESULTS

From calculating the two RMSE's from the training and test sets for the two TEAMS, we wrap them in a list. We named the TEAM: Multiple Regression as linear, and the TEAM: Neural Network regression as neural.

```
# aggregate all rmse
d.rmse <- list(linear.train = ds.lm$results$RMSE,
                linear.test = lm.rmse,
                neural.train = rsnns.train.rmse,
                neural.test = rsnns.test.rmse)
```

Below we can evaluate the models from their RMSE values.

```
d.rmse
#> $linear.train
#> [1] 1140
#>
#> $linear.test
#> [1] 1168
#>
#> $neural.train
#> [1] 739
#>
#> $neural.test
#> [1] 751
```

Looking at the training RMSE first, we see a clear difference as the linear RMSE was 66% larger than the neural RMSE, at 1,152.393 versus 692.5155. Peeking into the test sets, we have a similar 63% larger linear RMSE than the neural RMSE, with 1,110.843 and 679.5265 respectively. TEAM: Neural Network begins to gain the upper hand in the evaluation round.

One important difference between the two models was the range of the predictions. Recall from both training and test plots that the linear regression model predicted negative price values, whereas the MLP model predicted only positive prices. This is a devastating blow to the Multiple Regression. Also, the over-prediction of prices existed in both models, however the linear regression model over predicted those middle values higher the anticipated maximum price values.

Sometimes the simple models are optimal, and other times more complicated models are better. This time, the neural network model prevailed in predicting diamond prices.

Bibliography