

An R Companion for Introduction to Data Mining

Michael Hahsler

2025-12-08



An R Companion for Introduction to Data Mining

by Michael Hahsler



Contents

Preface	13
1 Introduction	15
1.1 Used Software	16
1.2 Base-R	16
1.2.1 Vectors	17
1.2.2 Vectorized Operations	17
1.2.3 Subsetting	17
1.2.4 Names	18
1.2.5 Lists	19
1.2.6 Data Frames	19
1.2.7 Matrices	20
1.2.8 Strings	20
1.2.9 Objects	21
1.2.10 Functions	22
1.2.11 Plotting	22
1.2.12 More on R	23
1.3 R Markdown	23
1.4 Tidyverse	23
1.4.1 Tibbles	24
1.4.2 Transformations	25
1.4.3 ggplot2	26

2 Data	31
Packages Used in this Chapter	31
2.1 Types of Data	32
2.1.1 Attributes and Measurement	32
2.1.2 The Iris Dataset	33
2.2 Data Quality	34
2.3 Data Preprocessing	37
2.3.1 Aggregation	37
2.3.2 Sampling	38
2.3.2.1 Random Sampling	38
2.3.2.2 Stratified Sampling	40
2.3.3 Dimensionality Reduction	41
2.3.3.1 Principal Components Analysis (PCA)	42
2.3.3.2 Multi-Dimensional Scaling (MDS)	47
2.3.3.3 Non-Parametric Multidimensional Scaling	48
2.3.3.4 Embeddings: Nonlinear Dimensionality Reduction Methods	48
2.3.4 Feature Subset Selection	50
2.3.5 Discretization	50
2.3.6 Variable Transformation: Standardization	56
2.4 Measures of Similarity and Dissimilarity	57
2.4.1 Minkowsky Distances	58
2.4.2 Distances for Binary Data	59
2.4.2.1 Hamming Distance	60
2.4.2.2 Jaccard Index	60
2.4.3 Distances for Mixed Data	60
2.4.3.1 Gower's Coefficient	61
2.4.3.2 Using Euclidean Distance with Mixed Data	61
2.4.4 More Proximity Measures	63
2.5 Exercises*	64

3	Classification: Basic Concepts	67
	Packages Used in this Chapter	67
3.1	Basic Concepts	68
3.2	General Framework for Classification	69
3.2.1	The Zoo Dataset	69
3.3	Decision Tree Classifiers	72
3.3.1	Create Tree	72
3.3.2	Make Predictions for New Data	73
3.3.3	Calculation of the Resubstitution Error	74
3.4	Model Overfitting	78
3.5	Model Selection	79
3.6	Model Evaluation	81
3.6.1	Holdout Method	81
3.6.2	Cross-Validation Methods	81
3.7	Hyperparameter Tuning	82
3.8	Pitfalls of Model Selection and Evaluation	89
3.9	Model Comparison	89
3.9.1	Build models	89
3.10	Feature Selection*	94
3.10.1	Univariate Feature Importance Score	94
3.10.2	Feature Subset Selection	98
3.11	Using Dummy Variables for Nominal Features*	99
3.12	Exercises*	102
4	Classification: Alternative Techniques	105
	Packages Used in this Chapter	105
4.1	Types of Classifiers	106
4.1.1	Set up the Training and Test Data	106
4.2	Rule-based classifier: PART	108
4.3	Nearest Neighbor Classifier	109
4.4	Naive Bayes Classifier	110

4.5	Bayesian Network	112
4.6	Logistic Regression	112
4.7	Artificial Neural Network (ANN)	114
4.8	Support Vector Machines	116
4.9	Ensemble Methods	117
4.9.1	Random Forest	117
4.9.2	Gradient Boosted Decision Trees (xgboost)	119
4.10	Model Comparison	121
4.11	Class Imbalance	127
4.11.1	Option 1: Use the Data As Is and Hope For The Best . . .	131
4.11.2	Option 2: Balance Data With Resampling	133
4.11.3	Option 3: Build A Larger Tree and use Predicted Probabilities	137
4.11.3.1	Create A Biased Classifier	139
4.11.3.2	Plot the ROC Curve	141
4.11.4	Option 4: Use a Cost-Sensitive Classifier	142
4.12	Comparing Decision Boundaries of Popular Classification Techniques*	145
4.12.1	Iris Dataset	147
4.12.1.1	Nearest Neighbor Classifier	148
4.12.1.2	Naive Bayes Classifier	151
4.12.1.3	Linear Discriminant Analysis	151
4.12.1.4	Multinomial Logistic Regression	152
4.12.1.5	Decision Trees	153
4.12.1.6	Ensemble: Random Forest	156
4.12.1.7	Support Vector Machine	156
4.12.1.8	Single Layer Feed-forward Neural Networks . . .	159
4.12.2	Circle Dataset	166
4.12.2.1	Nearest Neighbor Classifier	167
4.12.2.2	Naive Bayes Classifier	169
4.12.2.3	Linear Discriminant Analysis	169
4.12.2.4	Multinomial Logistic Regression	170

4.12.2.5	Decision Trees	171
4.12.2.6	Ensemble: Random Forest	173
4.12.2.7	Support Vector Machine	174
4.12.2.8	Single Layer Feed-forward Neural Networks	177
4.13	More Information on Classification with R	181
4.14	Exercises*	181

5 Association Analysis: Basic Concepts 183

	Packages Used in this Chapter	183
5.1	Preliminaries	184
5.1.1	The arules Package	184
5.1.2	Transactions	185
5.1.2.1	Create Transactions	185
5.1.2.2	Alternative Encodings for Continuous Values	187
5.1.2.3	Inspecting Transactions	188
5.1.2.4	Negative Items	193
5.1.2.5	Vertical Layout (Transaction ID Lists)	198
5.2	Frequent Itemset Generation	199
5.3	Rule Generation	204
5.3.1	Additional Interest Measures	209
5.3.2	Mine Using Templates	210
5.3.3	Redundant Rules	212
5.3.4	Saving Rules for External Tools	214
5.4	Compact Representation of Frequent Itemsets	214
5.5	Association Rule Visualization*	216
5.5.1	Static Visualizations	216
5.5.1.1	Scatterplot	218
5.5.1.2	Grouped Matrix Plot	220
5.5.1.3	Graph-based Visualization	221
5.5.2	Interactive Visualizations	222
5.5.2.1	Interactive Inspect With Sorting, Filtering and Paging	224

5.5.2.2	Scatter Plot	225
5.5.2.3	Matrix Visualization	225
5.5.2.4	Visualization as Graph	225
5.5.2.5	Interactive Rule Explorer	226
5.6	Exercises*	226
6	Association Analysis: Advanced Concepts	229
	Packages Used in this Chapter	229
6.1	Handling Categorical Attributes	229
6.2	Handling Continuous Attributes	230
6.3	Handling Concept Hierarchies	232
6.3.1	Aggregation	232
6.3.2	Multi-level Analysis	235
6.4	Sequential Patterns	237
7	Cluster Analysis	241
	Packages Used in this Chapter	241
7.1	Overview	242
7.1.1	Data Preparation	242
7.1.2	Data cleaning	243
7.1.3	Scale data	244
7.2	K-means	245
7.2.1	Inspect Clusters	249
7.2.2	Extract a Single Cluster	250
7.3	Agglomerative Hierarchical Clustering	252
7.3.1	Creating a Dendrogram	252
7.3.2	Extracting a Partitional Clustering	254
7.4	DBSCAN	258
7.4.1	DBSCAN Parameters	258
7.4.2	Cluster using DBSCAN	259
7.5	Cluster Evaluation	261
7.5.1	Unsupervised Cluster Evaluation	261

7.5.1.1	Visual Methods	261
7.5.1.2	Evaluation Metrics	265
7.5.2	Determining the Correct Number of Clusters	271
7.5.2.1	Elbow Method: Within-Cluster Sum of Squares	272
7.5.2.2	Average Silhouette Width	273
7.5.2.3	Similarity Matrix Visualization	274
7.5.2.4	Dunn Index	277
7.5.2.5	Gap Statistic	278
7.5.3	Clustering Tendency	279
7.5.3.1	Scatter plots	280
7.5.3.2	Visual Analysis for Cluster Tendency Assessment (VAT)	280
7.5.3.3	Hopkins statistic	282
7.5.3.4	Data Without Clustering Tendency	283
7.5.3.5	k-means on Data Without Clustering Tendency	286
7.5.4	Supervised Cluster Evaluation	287
7.6	More Clustering Algorithms*	295
7.6.1	Partitioning Around Medoids (PAM)	296
7.6.2	Gaussian Mixture Models	298
7.6.3	Spectral Clustering	300
7.6.4	Deep Clustering Methods	302
7.6.5	Fuzzy C-Means Clustering	302
7.7	Scale Issues in Clustering*	306
7.8	Outliers in Clustering*	307
7.8.1	Visual Identification of Outliers	310
7.8.2	Local Outlier Factor	311
7.9	Exercises*	316

A	Data Exploration and Visualization	319
	Packages Used in this Chapter	319
A.1	Exploring Data	320
A.1.1	Basic statistics	320
A.1.2	Grouped Operations and Calculations	322
A.1.3	Tabulate data	323
A.1.4	Percentiles (Quantiles)	326
A.1.5	Correlation	327
	A.1.5.1 Pearson Correlation	327
	A.1.5.2 Rank Correlation	330
A.1.6	Density	332
	A.1.6.1 Histograms	333
	A.1.6.2 Kernel Density Estimate (KDE)	336
A.2	Visualization	337
A.2.1	Histogram	338
A.2.2	Boxplot	341
A.2.3	Scatter plot	344
A.2.4	Scatter Plot Matrix	347
A.2.5	Matrix Visualization	349
A.2.6	Correlation Matrix	352
A.2.7	Parallel Coordinates Plot	358
A.2.8	Star Plot	360
A.2.9	More Visualizations	361
A.3	Exercises*	361
B	Regression	363
	Packages Used in this Chapter	363
B.1	Introduction	363
B.2	A First Linear Regression Model	364
B.3	Comparing Nested Models	369
B.4	Stepwise Variable Selection	372

<i>CONTENTS</i>	11
B.5 Modeling with Interaction Terms	374
B.6 Prediction	375
B.7 Using Nominal Variables	377
B.8 Alternative Regression Models	380
B.8.1 Regression Trees	380
B.8.2 Regularized Regression	382
B.8.3 ANNs	388
B.8.4 Other Types of Regression	390
B.9 Exercises	390
C Logistic Regression	393
C.1 Introduction	393
C.2 Data Preparation	395
C.3 A first Logistic Regression Model	396
C.4 Stepwise Variable Selection	398
C.5 Calculate the Response	400
C.6 Check Classification Performance	400
C.7 Regularized Logistic Regression	402
C.8 Multinomial Logistic Regression	403
C.9 Exercises	405
References	407

Preface

This companion book contains documented R examples to accompany several chapters of the popular data mining textbook *Introduction to Data Mining*¹ by Pang-Ning Tan, Michael Steinbach, Anuj Karpatne and Vipin Kumar. It is not intended as a replacement for the textbook since it does not cover the theory, but as a guide accompanying the textbook. The companion book can be used with either edition: 1st edition (Tan, Steinbach, and Kumar 2005) or 2nd edition (Tan et al. 2017). The sections are numbered to match the 2nd edition. Sections marked with an asterisk are additional content that is not covered in the textbook.

The code examples collected in this book were developed for the course *CS 5/7331 Data Mining* taught at the advanced undergraduate and graduate level at the Computer Science Department² at Southern Methodist University (SMU) since Spring 2013 and will be regularly updated and improved. The learning method used in this book is learning-by-doing. The code examples throughout this book are written in a self-contained manner so you can copy and paste a portion of the code, try it out on the provided dataset and then apply it directly to your own data. Instructors can use this companion as a component to create an introduction to data mining course for advanced undergraduates and graduate students who are proficient in programming and have basic statistics knowledge. The latest version of this book (html and PDF) with a complete set of lecture slides (PDF and PowerPoint) is provided on the book's GitHub page.³

The latest update includes the use of the popular packages in the meta-package **tidyverse** (Wickham 2023b) including **ggplot2** (Wickham et al. 2025) for data wrangling and visualization, along with **caret** (Kuhn 2024) for model building and evaluation. Please use the edit function within this book or visit the book's GitHub project page⁴ to submit corrections or suggest improvements.

¹<https://www-users.cs.umn.edu/~kumar001/dmbook/>

²<https://www.smu.edu/lyle/departments/cs>

³https://github.com/mhahsler/Introduction_to_Data_Mining_R_Examples

⁴https://github.com/mhahsler/Introduction_to_Data_Mining_R_Examples

To cite this book, use:

Michael Hahsler (2024). *An R Companion for Introduction to Data Mining*. figshare. DOI: 10.6084/m9.figshare.26750404 URL: https://mhahsler.github.io/Introduction_to_Data_Mining_R_Examples/book/

I hope this book helps you to learn to use R more efficiently for your data mining projects.

Michael Hahsler



© 2024 Michael Hahsler. This book is licensed under the Creative Commons Attribution 4.0 International License⁵.

The cover art is based on “rocks”⁶ by stebulus⁷ licensed with CC BY 2.0⁸.

This book was built on Mon Dec 8 11:19:28 2025 (latest GitHub tag: 1.0.3)

⁵<http://creativecommons.org/licenses/by/4.0/>

⁶<https://www.flickr.com/photos/69017177@N00/5063131410>

⁷<https://www.flickr.com/photos/69017177@N00>

⁸<https://creativecommons.org/licenses/by/2.0/?ref=ccsearch&atype=rich>

Chapter 1

Introduction

Data mining¹ has the goal of finding patterns in large data sets. The popular data mining textbook *Introduction to Data Mining*² (Tan et al. 2017) covers many important aspects of data mining. This companion contains annotated R code examples to complement the textbook. To make following along easier, we follow the chapters in data mining textbook which is organized by the main data mining tasks:

2. Data covers types of data. We also include data preparation and exploratory data analysis in Appendix A Data Exploration and Visualization.
3. Classification: Basic Concepts introduces the purpose of classification, basic classifiers using decision trees, and model training and evaluation.
4. Classification: Alternative Techniques introduces and compares methods including rule-based classifiers, nearest neighbor classifiers, naive Bayes classifier, logistic regression and artificial neural networks.
5. Association Analysis: Basic Concepts covers algorithms for frequent item-set and association rule generation and analysis including visualization.
6. Association Analysis: Advanced Concepts covers categorical and continuous attributes, concept hierarchies, and frequent sequence pattern mining.
7. Cluster Analysis discusses clustering approaches including k-means, hierarchical clustering, DBSCAN and how to evaluate clustering results.

For completeness, we have added sections on Regression and on Logistic Regression to the Appendix.

¹https://en.wikipedia.org/wiki/Data_mining

²<https://www-users.cs.umn.edu/~kumar001/dmbook/>

Sections with names followed by an asterisk contain code examples for methods that are not included in the data mining textbook.

This book assumes that you are familiar with the basics of R, how to run R code, and install packages. The rest of this chapter will provide an overview and point you to where you can learn more about R and the used packages.

1.1 Used Software

To use this book, you need to have the current version of R³ and RStudio Desktop⁴ installed.

Each book chapter will use a set of packages that must be installed. The installation code can be found at the beginning of each chapter. Here is the code to install the packages used in this chapter:

```
pkgs <- c('tidyverse')

pkgs_install <- pkgs[!(pkgs %in% installed.packages()[,"Package"])]
if(length(pkgs_install)) install.packages(pkgs_install)
```

The code examples in this book use the R package collection `tidyverse` (Wickham 2023b) to manipulate data. Tidyverse also includes the package `ggplot2` (Wickham et al. 2025) for visualization. Tidyverse packages make working with data in R very convenient. Data analysis and data mining reports are typically done by creating R Markdown documents. Everything in R is built on top of the core R programming language and the packages that are automatically installed with R. This is referred to as Base-R.

1.2 Base-R

Base-R is covered in detail in An Introduction to R⁵. The most important differences between R and many other programming languages like Python are:

- R is a functional programming language (with some extensions).
- R only uses vectors and operations are vectorized. You rarely will see loops.
- R starts indexing into vectors with 1 not 0.
- R uses `<-` for assignment. Do not use `=` for assignment.

³<https://cran.r-project.org/>

⁴<https://posit.co/products/open-source/rstudio/>

⁵<https://cran.r-project.org/doc/manuals/r-release/R-intro.html>

1.2.1 Vectors

The basic data structure in R is a vector of real numbers called `numeric`. Scalars do not exist, they are just vectors of length 1.

We can combine values into a vector using the combine function `c()`. Special values for infinity (`Inf`) and missing values (`NA`) can be used.

```
x <- c(10.4, 5.6, Inf, NA, 21.7)
x
## [1] 10.4  5.6  Inf   NA 21.7
```

We often use sequences. A simple sequence of integers can be produced using the colon operator in the form `from:to`.

```
3:10
## [1]  3  4  5  6  7  8  9 10
```

More complicated sequences can be created using `seq()`.

```
y <- seq(from = 0, to = 1, length.out = 5)
y
## [1] 0.00 0.25 0.50 0.75 1.00
```

1.2.2 Vectorized Operations

Operations are vectorized and applied to each element, so loops are typically not necessary in R.

```
x + 1
## [1] 11.4  6.6  Inf   NA 22.7
```

Comparisons are also vectorized and are performed element-wise. They return a `logical` vector (R's name for the datatype Boolean).

```
x > y
## [1] TRUE TRUE TRUE  NA TRUE
```

1.2.3 Subsetting

We can select vector elements using the `[]` operator like in other programming languages but the index always starts at 1.

We can select elements 1 through 3 using an index sequence.

```
x[1:3]
## [1] 10.4 5.6 Inf
```

Negative indices remove elements. We can select all but the first element.

```
x[-1]
## [1] 5.6 Inf NA 21.7
```

We can use any function that creates a `logical` vector for subsetting. Here we select all non-missing values using the function `is.na()`.

```
is.na(x)
## [1] FALSE FALSE FALSE TRUE FALSE
x[!is.na(x)] # select all non-missing values
## [1] 10.4 5.6 Inf 21.7
```

We can also assign values to a selection. For example, this code gets rid of infinite values.

```
x[!is.finite(x)] <- NA
x
## [1] 10.4 5.6 NA NA 21.7
```

1.2.4 Names

Another useful thing is that vectors and many other data structures in R support names. For example, we can store the count of cars with different colors in the following way.

```
counts <- c(12, 5, 18, 13)
names(counts) <- c("black", "red", "white", "silver")

counts
## black red white silver
## 12 5 18 13
```

Names can then be used for subsetting and we do not have to remember what entry is associated with what color.

```
counts[c("red", "silver")]
## red silver
## 5 13
```

1.2.5 Lists

Lists can store a sequence of elements of different datatypes.

```
lst <- list(name = "Fred", spouse = "Mary", no.children = 3,
  child.ages = c(4, 7, 9))
lst
## $name
## [1] "Fred"
##
## $spouse
## [1] "Mary"
##
## $no.children
## [1] 3
##
## $child.ages
## [1] 4 7 9
```

Elements can be accessed by index or name.

```
lst[[2]]
## [1] "Mary"
lst$child.ages
## [1] 4 7 9
```

Many functions in R return a list.

1.2.6 Data Frames

A data frame looks like a spread sheet and represents a matrix-like structure with rows and columns where different columns can contain different data types.

```
df <- data.frame(name = c("Michael", "Mark", "Maggie"),
  children = c(2, 0, 2), age = c(40, 30, 50))
df
##      name children age
## 1 Michael        2  40
## 2   Mark         0  30
## 3  Maggie        2  50
```

Data frames are stored as a list of columns. We can access elements using matrix subsetting (missing indices mean the whole row or column) or list subsetting. Here are some examples

```
df[1,1]
## [1] "Michael"
df[, 2]
## [1] 2 0 2
df$name
## [1] "Michael" "Mark"    "Maggie"
```

Data frames are the most important way to represent data for data mining.

1.2.7 Matrices

A matrix is similar to a data frame but it only contains values of the same data type. Some data mining algorithms require a numeric matrix as the input. The numeric part of Data frames can be coerced into a matrix. In the following example we cast the last two columns of `df` into a matrix. Column names are taken from the data frame and we add manually row names.

```
x <- as.matrix(df[, -1])
rownames(x) <- df[, 1]
x
##           children age
## Michael         2  40
## Mark            0  30
## Maggie          2  50
```

Data type coercion functions in R start with `as..` Examples as `as.logical()`, `as.data.frame()`, `as.list()`, etc.

1.2.8 Strings

R uses `character` vectors where the datatype `character` represents a string and not like in other programming language just a single character. R accepts double or single quotation marks to delimit strings.

```
string <- c("Hello", "Goodbye")
string
## [1] "Hello"  "Goodbye"
```

Strings can be combined using `paste()`.


```
paste(string, "World!")
## [1] "Hello World!" "Goodbye World!"
```

Note that `paste` is vectorized and the string "World!" is used twice to match the length of `string`. This behavior is called in R *recycling* and works if one vector's length is an exact multiple of the other vector's length. The special case where one vector has one element is particularly often used. We have used it already above in the expression `x + 1` where one is recycled for each element in `x`.

1.2.9 Objects

Other often used data structures include `list`, `data.frame`, `matrix`, and `factor`. In R, everything is an object. Objects are printed by either just using the object's name or put it explicitly into the `print()` function.

```
x
##           children age
## Michael         2  40
## Mark            0  30
## Maggie          2  50
```

For many objects, a summary can be created.

```
summary(x)
##           children           age
## Min.      :0.00   Min.      :30
## 1st Qu.:1.00   1st Qu.:35
## Median :2.00   Median :40
## Mean     :1.33   Mean     :40
## 3rd Qu.:2.00   3rd Qu.:45
## Max.     :2.00   Max.     :50
```

Objects have a class.

```
class(x)
## [1] "matrix" "array"
```

Some functions are generic, meaning that they do something else depending on the class of the first argument. For example, `plot()` has many methods implemented to visualize data of different classes. The specific function is specified with a period after the method name. There also exists a default method. For `plot`, this method is `plot.default()`. Different methods may have their own

manual page, so specifying the class with the dot can help finding the documentation.

It is often useful to know what information it stored in an object. `str()` returns a humanly readable string representation of the object.

```
str(x)
##  num [1:3, 1:2] 2 0 2 40 30 50
##  - attr(*, "dimnames")=List of 2
##  ..$ : chr [1:3] "Michael" "Mark" "Maggie"
##  ..$ : chr [1:2] "children" "age"
```

1.2.10 Functions

Functions work like in many other languages. However, they also operate vectorized and arguments can be specified positional or by named. An increment function can be defined as:

```
inc <- function(x, by = 1) {
  x + by
}
```

The value of the last evaluated expression in the function body is automatically returned by the function. The function can also explicitly use `return(return_value)`.

Calling the increment function on vector with the numbers 1 through 10.

```
v <- 1:10
inc(v, by = 2)
##  [1] 3 4 5 6 7 8 9 10 11 12
```

Before you implement your own function, check if R does not already provide an implementation. R has many built-in functions like `min()`, `max()`, `mean()`, and `sum()`.

1.2.11 Plotting

Basic plotting in Base-R is done by calling `plot()`.

```
plot(x, y)
```

Other plot functions are `pairs()`, `hist()`, `barplot()`. In this book, we will focus on plots created with the `ggplot2` package introduced below.

1.2.12 More on R

There is much to learn about R. It is highly recommended to go through the official An Introduction to R⁶ manual. There is also a good Base R Cheat Sheet⁷ available.

1.3 R Markdown

R Markdown is a simple method to include R code inside a text document written using markdown syntax. Using R markdown is especially convenient to analyze data and compose a data mining report. RStudio makes creating and translating R Markdown documents easy. Just choose **File -> New File -> R Markdown...** RStudio will create a small demo markdown document that already includes examples for code and how to include plots. You can switch to visual mode if you prefer a What You See Is What You Get editor.

To convert the R markdown document to HTML, PDF or Word, just click the Knit button. All the code will be executed and combined with the text into a complete document.

Examples and detailed documentation can be found on RStudio's R Markdown website⁸ and in the R Markdown Cheatsheet⁹.

1.4 Tidyverse

tidyverse (Wickham 2023b) is a collection of many useful packages that work well together by sharing design principles and data structures. **tidyverse** also includes **ggplot2** (Wickham et al. 2025) for visualization.

In this book, we will typically use

- tidyverse tibbles to replace R's built-in data.frames,
- the pipe operator `|>` to chain functions together, and
- data transformation functions like `filter()`, `arrange()`, `select()`, `group_by()`, and `mutate()` provided by the tidyverse package **dplyr** (Wickham et al. 2023).

A good introduction can be found in the Section on Data Transformation¹⁰ (Wickham, Çetinkaya-Rundel, and Golemund 2023).

⁶<https://cran.r-project.org/doc/manuals/r-release/R-intro.html>

⁷<https://github.com/rstudio/cheatsheets/blob/main/base-r.pdf>

⁸<https://rmarkdown.rstudio.com/>

⁹<https://rstudio.github.io/cheatsheets/html/rmarkdown.html>

¹⁰<https://r4ds.hadley.nz/data-transform>

To use tidyverse, we first have to load the tidyverse meta package.

```
library(tidyverse)
## -- Attaching core tidyverse packages ---- tidyverse 2.0.0 --
## v dplyr      1.1.4      v readr      2.1.5
## v forcats    1.0.0      v stringr   1.5.2
## v ggplot2    4.0.0      v tibble    3.3.0
## v lubridate  1.9.4      v tidyr     1.3.1
## v purrr      1.1.0
## -- Conflicts ----- tidyverse_conflicts() --
## x dplyr::filter() masks stats::filter()
## x dplyr::lag()    masks stats::lag()
## i Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to be resolved.
```

Conflicts may indicate that base R functionality is changed by tidyverse packages.

1.4.1 Tibbles

Tibbles are tidyverse's replacement for R's `data.frame`. Here is a short example that analyses the vitamin C content of different fruits that will get you familiar with the basic syntax. We create a tibble with the price in dollars per pound and the vitamin C content in milligrams (mg) per pound for five different fruits.

```
fruit <- tibble(
  name = c("apple", "banana", "mango", "orange", "lime"),
  price = c(2.5, 2.0, 4.0, 3.5, 2.5),
  vitamin_c = c(20, 45, 130, 250, 132),
  type = c("pome", "tropical", "tropical", "citrus", "citrus"))
fruit
## # A tibble: 5 x 4
##   name    price vitamin_c type
##   <chr>   <dbl>     <dbl> <chr>
## 1 apple     2.5         20 pome
## 2 banana     2         45 tropical
## 3 mango      4        130 tropical
## 4 orange    3.5        250 citrus
## 5 lime      2.5        132 citrus
```

Next, we will transform the data to find affordable fruits with a high vitamin C content and then visualize the data.

1.4.2 Transformations

We can modify the table by adding a column with the vitamin C (in mg) that a dollar buys using `mutate()`. Then we filter only rows with fruit that provides more than 20 mg per dollar, and finally we arrange the data rows by the vitamin C per dollar from largest to smallest.

```
affordable_vitamin_c_sources <- fruit |>
  mutate(vitamin_c_per_dollar = vitamin_c / price) |>
  filter(vitamin_c_per_dollar > 20) |>
  arrange(desc(vitamin_c_per_dollar))

affordable_vitamin_c_sources
## # A tibble: 4 x 5
##   name    price vitamin_c type    vitamin_c_per_dollar
##   <chr>   <dbl>    <dbl> <chr>          <dbl>
## 1 orange    3.5      250 citrus          71.4
## 2 lime      2.5      132 citrus          52.8
## 3 mango     4       130 tropical         32.5
## 4 banana    2       45 tropical         22.5
```

The pipes operator `|>` lets you pass the value to the left (often the result of a function) as the first argument to the function on the right. This makes composing a sequence of function calls to transform data much easier to write and read. You will often see `%>%` as the pipe operator, especially in examples using tidyverse. Both operators work similarly where `|>` is a native R operator while `%>%` is provided in the extension package `magrittr`.

The code above starts with the fruit data and pipes it through three transformation functions. The final result is assigned with `<-` to a variable.

We can create summary statistics of price using `summarize()`.

```
affordable_vitamin_c_sources |>
  summarize(min = min(price),
            mean = mean(price),
            max = max(price))
## # A tibble: 1 x 3
##   min mean max
##   <dbl> <dbl> <dbl>
## 1     2     3     4
```

We can also calculate statistics for groups by first grouping the data with `group_by()`. Here we produce statistics for fruit types.

```
affordable_vitamin_c_sources |>
  group_by(type) |>
  summarize(min = min(price),
            mean = mean(price),
            max = max(price))
## # A tibble: 2 x 4
##   type      min mean  max
##   <chr>    <dbl> <dbl> <dbl>
## 1 citrus    2.5     3   3.5
## 2 tropical  2       3    4
```

Often, we want to apply the same function to multiple columns. This can be achieved using `across(columns, function)`.

```
affordable_vitamin_c_sources |>
  summarize(across(c(price, vitamin_c), mean))
## # A tibble: 1 x 2
##   price vitamin_c
##   <dbl>      <dbl>
## 1     3        139.
```

`dplyr` syntax and evaluation is slightly different from standard R which may lead to confusion. One example is that column names can be references without quotation marks. A very useful reference resource when working with `dplyr` is the RStudio Data Transformation Cheatsheet¹¹ which covers on two pages almost everything you will need and also contains simple example code that you can take and modify for your use case.

1.4.3 ggplot2

For visualization, we will use mainly `ggplot2`. The *gg* in `ggplot2` stands for **Grammar of Graphics** introduced by Wilkinson (2005). The main idea is that every graph is built from the same basic components:

- the data,
- a coordinate system, and
- visual marks representing the data (geoms).

The main plotting function is `ggplot()`. The components of the plot are combined using the `+` operator.

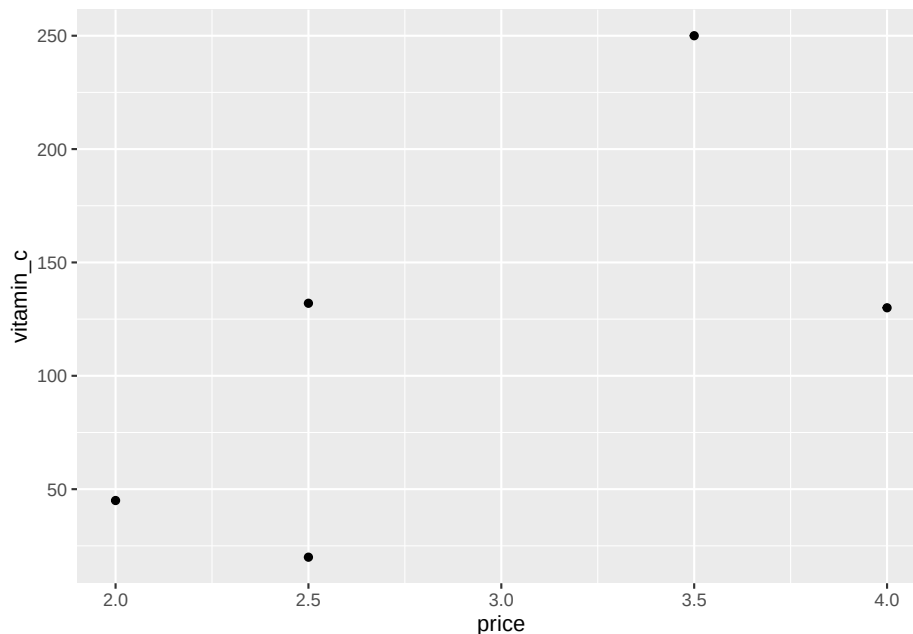
¹¹<https://rstudio.github.io/cheatsheets/html/data-transformation.html>

```
ggplot(data, mapping = aes(x = ..., y = ..., color = ...)) +  
  geom_point()
```

Since we typically use a Cartesian coordinate system, `ggplot()` uses that by default. Each `geom_` function uses a `stat_` function to calculate what is visualized. For example, `geom_bar()` uses `stat_count()` to create a bar chart by counting how often each value appears in the data (see `? geom_bar`). `geom_point()` just uses the stat `"identity"` to display the points using the coordinates as they are. A great introduction can be found in the Section on Data Visualization¹² (Wickham, Çetinkaya-Rundel, and Grolemund 2023), and very useful is RStudio's Data Visualization Cheatsheet¹³.

We can visualize our fruit data as a scatter plot.

```
ggplot(fruit, aes(x = price, y = vitamin_c)) +  
  geom_point()
```

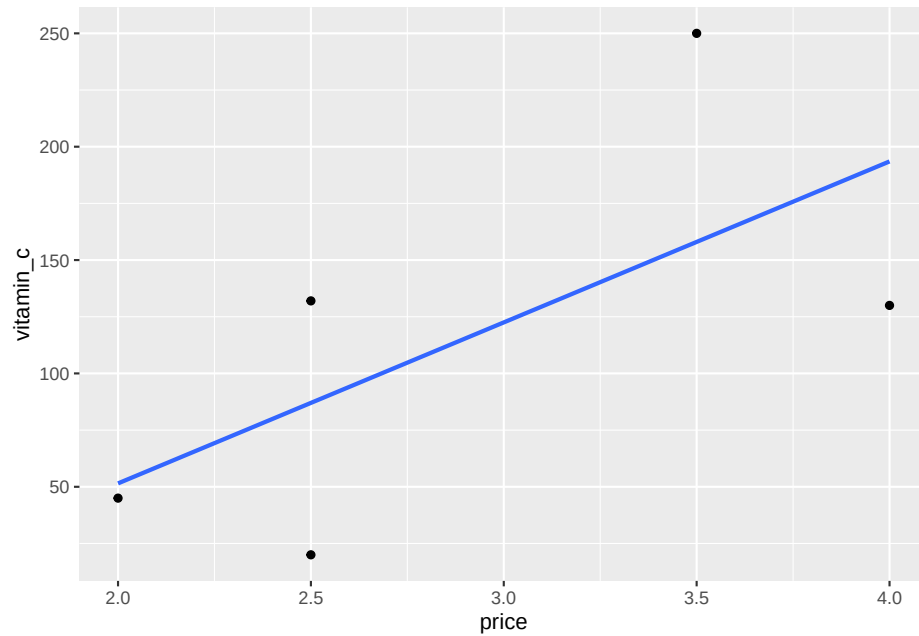


It is easy to add more geoms. For example, we can add a regression line using `geom_smooth` with the method `"lm"` (linear model). We suppress the confidence interval since we only have 3 data points.

¹²<https://r4ds.hadley.nz/data-visualize>

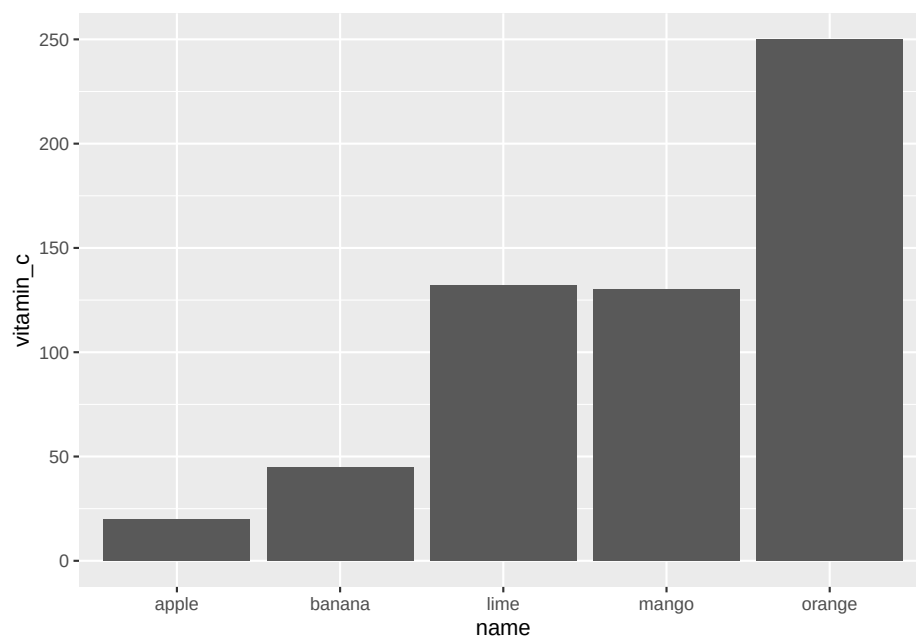
¹³<https://rstudio.github.io/cheatsheets/html/data-visualization.html>

```
ggplot(fruit, aes(x = price, y = vitamin_c)) +  
  geom_point() +  
  geom_smooth(method = "lm", se = FALSE)  
## `geom_smooth()` using formula = 'y ~ x'
```



Alternatively, we can visualize each fruit's vitamin C content per dollar using a bar chart.

```
ggplot(fruit, aes(x = name, y = vitamin_c)) +  
  geom_bar(stat = "identity")
```

Note that `geom_bar` by default uses the `stat_count` function to aggregate data by counting, but we just want to visualize the value in the tibble, so we specify the identity function instead.

Chapter 2

Data

Data for data mining is typically organized in tabular form, with rows containing the objects of interest and columns representing attributes describing the objects. We will discuss topics like data quality, sampling, feature selection, and how to measure similarities between objects and features.

After this chapter, you can read Appendix A Data Exploration and Visualization to learn more about data exploration and visualization in R.

Packages Used in this Chapter

```
pkgs <- c("arules", "caret", "factoextra", "GGally",
          "palmerpenguins", "plotly",
          "proxy", "Rtsne", "tidyverse")

pkgs_install <- pkgs[!(pkgs %in% installed.packages()[, "Package"])]
if(length(pkgs_install)) install.packages(pkgs_install)
```

The packages used for this chapter are:

- *arules* (Hahsler et al. 2025)
- *caret* (Kuhn 2024)
- *factoextra* (Kassambara and Mundt 2020)
- *GGally* (Schloerke et al. 2025)
- *palmerpenguins* (Horst, Hill, and Gorman 2022)
- *plotly* (Sievert et al. 2025)
- *proxy* (Meyer and Buchta 2022)
- *Rtsne* (Krijthe 2023)
- *tidyverse* (Wickham 2023b)

2.1 Types of Data

2.1.1 Attributes and Measurement

The values of features can be measured on several scales¹ ranging from simple labels all the way to numbers. The scales come in four levels.

Scale Name	Description	Operations	Statistics	R
Nominal	just a label (e.g., red, green)	$==, !=$	counts	factor
Ordinal	label with order (e.g., small, med., large)	$<, >$	median	ordered factor
Interval	difference between two values is meaningful (regular number)	$+, -$	mean, sd	numeric
Ratio	has a natural zero (e.g., count, distance)	$/, *$	percent	numeric

The scales build on each other meaning that an ordinal variable also has the characteristics of a nominal variable with the added order information. We often do not differentiate between interval and ratio scale because we rarely not need to calculate percentages or other statistics that require a meaningful zero value.

Nominal data is created using `factor()`. If the factor levels are not specified, then they are created in alphabetical order.

```
factor(c("red", "green", "green", "blue"))
## [1] red   green green blue
## Levels: blue green red
```

Ordinal data is created using `ordered()`. The levels specify the order.

```
ordered(c("S", "L", "M", "S"),
        levels = c("S", "M", "L"))
## [1] S L M S
## Levels: S < M < L
```

Ratio/interval data is created as a simple vector.

¹https://en.wikipedia.org/wiki/Level_of_measurement

```
c(1, 2, 3, 4, 3, 3)
## [1] 1 2 3 4 3 3
```

2.1.2 The Iris Dataset

We will use a toy dataset that comes with R. Fisher's iris dataset² gives the measurements in centimeters of the variables sepal length, sepal width petal length, and petal width representing the features for 150 flowers (the objects). The dataset contains 50 flowers from each of 3 species of iris. The species are Iris Setosa, Iris Versicolor, and Iris Virginica. For more details see `? iris`.

We load the iris data set. Datasets that come with R or R packages can be loaded with `data()`. The standard format for data in R is a `data.frame`. We convert the `data.frame` into a tidyverse tibble.

```
library(tidyverse)
data(iris)
iris <- as_tibble(iris)
iris
## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##   <dbl>         <dbl>         <dbl>         <dbl> <fct>
## 1           5.1           3.5           1.4           0.2 setosa
## 2           4.9           3             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             3.6           1.4           0.2 setosa
## 6           5.4           3.9           1.7           0.4 setosa
## 7           4.6           3.4           1.4           0.3 setosa
## 8           5             3.4           1.5           0.2 setosa
## 9           4.4           2.9           1.4           0.2 setosa
## 10          4.9           3.1           1.5           0.1 setosa
## # i 140 more rows
```

We see that the data contains 150 rows (flowers) and 5 features. tibbles only show the first few rows and do not show all features, if they do not fit the screen width. We can call `print` and define how many rows to show using parameter `n` and force print to show all features by changing the `width` to infinity.

```
print(iris, n = 3, width = Inf)
## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##   <dbl>         <dbl>         <dbl>         <dbl> <fct>
```

²https://en.wikipedia.org/wiki/Iris_flower_data_set

```
## 1      5.1      3.5      1.4      0.2 setosa
## 2      4.9      3      1.4      0.2 setosa
## 3      4.7      3.2      1.3      0.2 setosa
## # i 147 more rows
```

2.2 Data Quality

Assessing the quality of the available data is crucial before we start using the data. Start with summary statistics for each column to identify outliers and missing values. The easiest way is to use the base R function `summary()`.

```
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
##
##
##
```

Feature names will be used in plots and analysis so having understandable feature names is part of data quality. Iris's names are good but for most datasets, you may need to change the names using either `rownames()` or `rename()`.

You can also summarize individual columns using tidyverse's dplyr functions.

```
iris |>
  summarize(mean = mean(Sepal.Length))
## # A tibble: 1 x 1
##   mean
##   <dbl>
## 1  5.84
```

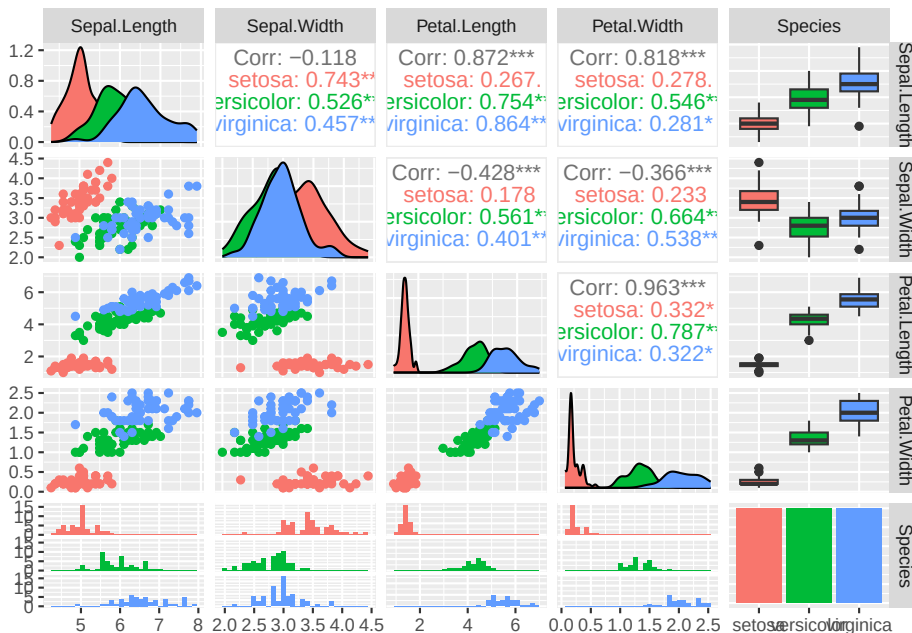
Using `across()`, multiple columns can be summarized. Un the following, we calculate all numeric columns using the `mean` function.

```
iris |>
  summarize(across(where(is.numeric), mean))
## # A tibble: 1 x 4
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
##         <dbl>         <dbl>         <dbl>         <dbl>
## 1         5.84         3.06         3.76         1.20
```

To find outliers or data problems, you need to look for very small values (often a suspicious large number of zeros) using min and for extremely large values using max. Comparing median and mean tells us if the distribution is symmetric.

A visual method to inspect the data is to use a scatterplot matrix (we use here `ggpairs()` from package `GGally`). In this plot, we can visually identify noise data points and outliers (points that are far from the majority of other points).

```
library(GGally)
ggpairs(iris, aes(color = Species), progress = FALSE)
## `stat_bin()` using `bins = 30`. Pick better value
## `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value
## `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value
## `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value
## `binwidth`.
```



This useful visualization combines many visualizations used to understand the data and check for quality issues. Rows and columns are the features in the data. We have also specified the aesthetic that we want to group each species using a different color.

- The visualizations in the diagonal panels show the smoothed histograms with the distribution for each feature. The plot tries to pick a good number of bins for the histogram (see messages above). The distribution can be checked if it is close to normal, unimodal or highly skewed. Also, we can see if the different groups are overlapping or separable for each feature. For example, the three distributions for `Sepal.Width` are almost identical meaning that it is hard to distinguish between the different species using this feature alone. `Petal.Length` and `Petal.Width` are much better.
- The lower-left triangle panels contain scatterplots for all pairs features. These are useful to see if there if features are correlated (the pearson correlation coefficient if printed in the upper-right triangle). For example, `Petal.Length` and `Petal.Width` are highly correlated overall This makes sense since larger plants will have both longer and wider petals. Inside the `Setosa` group this correlation it is a lot weaker. We can also see if groups are well separated using projections on two variables. Almost all panels show that `Setosa` forms a point cloud well separated from the other two classes while `Versicolor` and `Virginica` overlap. We can also see outliers that are far from the other data points in its group. See if you can spot the one red dot that is far away from all others.
- The last row/column represents in this data set the class label `Species`. It is a nominal variable so the plots are different. The bottom row panels show (regular) histograms. The last column shows boxplots to represent the distribution of the different features by group. Dots represent outliers. Finally, the bottom-right panel contains the counts for the different groups as a barplot. In this data set, each group has the same number of observations.

Many data mining methods require complete data, that is the data cannot contain missing values (`NA`). To remove missing values and duplicates (identical data points which might be a mistake in the data), we often do this:

```
clean.data <- iris |>
  drop_na() |>
  unique()

summary(clean.data)
##   Sepal.Length   Sepal.Width   Petal.Length
##   Min.      :4.30   Min.      :2.00   Min.      :1.00
##   1st Qu.:5.10   1st Qu.:2.80   1st Qu.:1.60
```



```
## Median :5.80 Median :3.00 Median :4.30
## Mean :5.84 Mean :3.06 Mean :3.75
## 3rd Qu.:6.40 3rd Qu.:3.30 3rd Qu.:5.10
## Max. :7.90 Max. :4.40 Max. :6.90
## Petal.Width Species
## Min. :0.10 setosa :50
## 1st Qu.:0.30 versicolor:50
## Median :1.30 virginica :49
## Mean :1.19
## 3rd Qu.:1.80
## Max. :2.50
```

The iris dataset has no missing values, but one non-unique case is gone leaving only 149 flowers. Since only 1 out of 150 flowers in the dataset was affected, results on the remaining data will give very similar results to the complete data.

Typically, you should spend a lot more time on data cleaning. It is important to always describe how you clean the data, and how many objects are removed. You need to argue that conclusions based on only the remaining data are still valid.

2.3 Data Preprocessing

2.3.1 Aggregation

Data often contains groups and we want to compare these groups. We group the iris dataset by species and then calculate a summary statistic for each group.

```
iris |>
  group_by(Species) |>
  summarize(across(everything(), mean))
## # A tibble: 3 x 5
##   Species Sepal.Length Sepal.Width Petal.Length Petal.Width
##   <fct>      <dbl>      <dbl>      <dbl>      <dbl>
## 1 setosa      5.01      3.43      1.46      0.246
## 2 versico~    5.94      2.77      4.26      1.33
## 3 virgini~    6.59      2.97      5.55      2.03
iris |>
  group_by(Species) |>
  summarize(across(everything(), median))
## # A tibble: 3 x 5
##   Species Sepal.Length Sepal.Width Petal.Length Petal.Width
##   <fct>      <dbl>      <dbl>      <dbl>      <dbl>
## 1 setosa      5      3.4      1.5      0.2
```

```
## 2 versico~      5.9      2.8      4.35      1.3
## 3 virginia~     6.5      3       5.55      2
```

Using this information, we can compare how features differ between groups.

2.3.2 Sampling

Sampling³ is often used in data mining to reduce the dataset size before modeling or visualization. Another application is to split data randomly into training and testing data for working with supervised models (e.g., classification models).

2.3.2.1 Random Sampling

The built-in sample function can sample from a vector. Here we sample with replacement.

```
sample(c("A", "B", "C"), size = 10, replace = TRUE)
## [1] "C" "C" "A" "A" "C" "C" "B" "B" "C" "B"
```

We often want to sample rows from a dataset. This can be done by sampling without replacement from a vector with row indices (using the functions `seq()` and `nrow()`). The sample vector is then used to subset the rows of the dataset.

```
take <- sample(seq(nrow(iris)), size = 15)
take
## [1] 40 68 112 67 110 3 88 69 56 61 106 101 60 48
## [15] 28
iris[take, ]
## # A tibble: 15 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##   <dbl>         <dbl>         <dbl>         <dbl> <fct>
## 1         5.1         3.4         1.5         0.2 setosa
## 2         5.8         2.7         4.1         1 versico~
## 3         6.4         2.7         5.3         1.9 virgin~
## 4         5.6         3         4.5         1.5 versico~
## 5         7.2         3.6         6.1         2.5 virgin~
## 6         4.7         3.2         1.3         0.2 setosa
## 7         6.3         2.3         4.4         1.3 versico~
## 8         6.2         2.2         4.5         1.5 versico~
## 9         5.7         2.8         4.5         1.3 versico~
## 10        5         2         3.5         1 versico~
```

³[https://en.wikipedia.org/wiki/Sampling_\(statistics\)](https://en.wikipedia.org/wiki/Sampling_(statistics))

```
## 11      7.6      3      6.6      2.1 virgin~
## 12      6.3      3.3      6      2.5 virgin~
## 13      5.2      2.7      3.9      1.4 versic~
## 14      4.6      3.2      1.4      0.2 setosa
## 15      5.2      3.5      1.5      0.2 setosa
```

`dplyr` from `tidyverse` lets us sample rows from tibbles directly using `slice_sample()`. I set the random number generator seed to make the results reproducible.

```
set.seed(1000)
s <- iris |>
  slice_sample(n = 15)

library(GGally)
ggpairs(s, aes(color = Species), progress = FALSE)
## `stat_bin()` using `bins = 30`. Pick better value
## `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value
## `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value
## `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value
## `binwidth`.
```



Instead of `n` you can also specify the proportion of rows to select using `prob`.

2.3.2.2 Stratified Sampling

Stratified sampling⁴ is a method of sampling from a population which can be partitioned into subpopulations, while controlling the proportions of the subpopulation in the resulting sample.

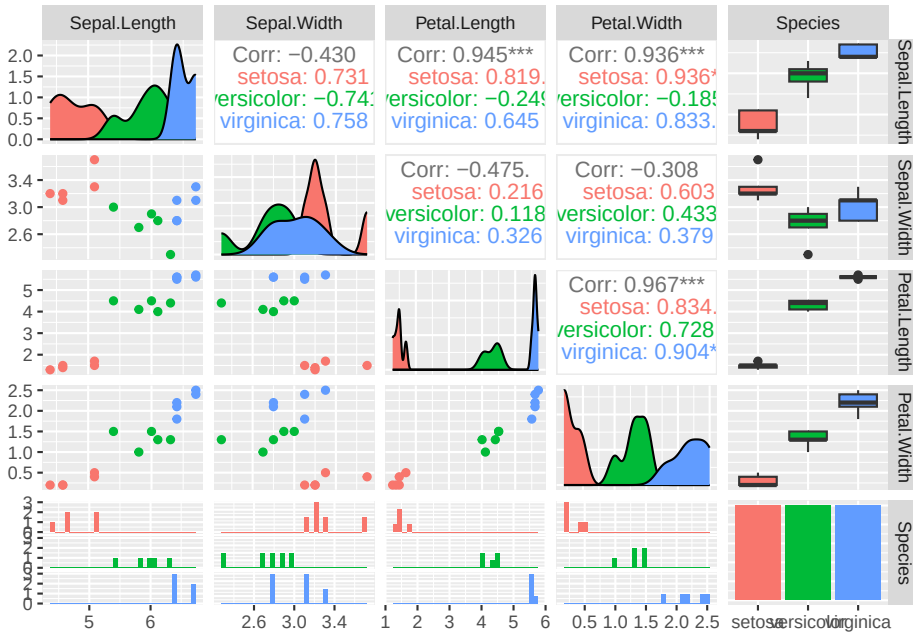
In the following, the subpopulations are the different types of species and we want to make sure to sample the same number (5) flowers from each. This can be achieved by first grouping the data by species and then sampling a number of flowers from each group.

```
set.seed(1000)

s2 <- iris |>
  group_by(Species) |>
  slice_sample(n = 5) |>
  ungroup()

library(GGally)
ggpairs(s2, aes(color = Species), progress = FALSE)
## `stat_bin()` using `bins = 30`. Pick better value
## `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value
## `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value
## `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value
## `binwidth`.
```

⁴https://en.wikipedia.org/wiki/Stratified_sampling



More sophisticated sampling procedures are implemented in the package `sampling`.

2.3.3 Dimensionality Reduction

The number of features is often called the dimensional of the data following the idea that each feature (at least the numeric features) can be seen as an axis of the data. High-dimensional data is harder to analyze by the user (e.g., visualize). It also is problematic for many data mining algorithms since it requires more memory and computational resources.

Dimensionality reduction⁵ tries to represent high-dimensional data in a low-dimensional space so that the low-dimensional representation retains some meaningful properties (e.g., information about similarity or distances) of the original data. Dimensionality reduction is used for visualization and as a preprocessing technique before using other data mining methods like clustering and classification.

Recently, data embeddings using artificial neural networks have become very popular. These approaches can not only reduce the dimensionality of the data, but learn a better representation of various kinds of data (e.g., text). As such these approaches can be seen as automatically engineering features from the high-dimensional original data.

⁵https://en.wikipedia.org/wiki/Dimensionality_reduction

2.3.3.1 Principal Components Analysis (PCA)

PCA⁶ calculates principal components (a set of new orthonormal basis vectors in the data space) from data points such that the first principal component explains the most variability in the data, the second the next most and so on. In data analysis, PCA is used to project high-dimensional data points onto the first few (typically two) principal components for visualization as a scatter plot and as preprocessing for modeling (e.g., before k-means clustering). Points that are closer together in the high-dimensional original space, tend also be closer together when projected into the lower-dimensional space,

We can use an interactive 3-d plot (from package `plotly`) to look at three of the four dimensions of the iris dataset. Note that it is hard to visualize more than 3 dimensions.

```
plotly::plot_ly(iris,
  x = ~Sepal.Length,
  y = ~Petal.Length,
  z = ~Sepal.Width,
  color = ~Species, size = 1) |>
plotly::add_markers()
```

*The resulting interactive plot can be seen in the online version of this book.*⁷

The principal components can be calculated from a matrix using the function `prcomp()`. We select all numeric columns (by unselecting the species column) and convert the tibble into a matrix before the calculation.

```
pc <- iris |>
  select(~Species) |>
  as.matrix() |>
  prcomp()
summary(pc)
## Importance of components:
##              PC1      PC2      PC3      PC4
## Standard deviation  2.056 0.4926 0.2797 0.15439
## Proportion of Variance 0.925 0.0531 0.0171 0.00521
## Cumulative Proportion 0.925 0.9777 0.9948 1.00000
```

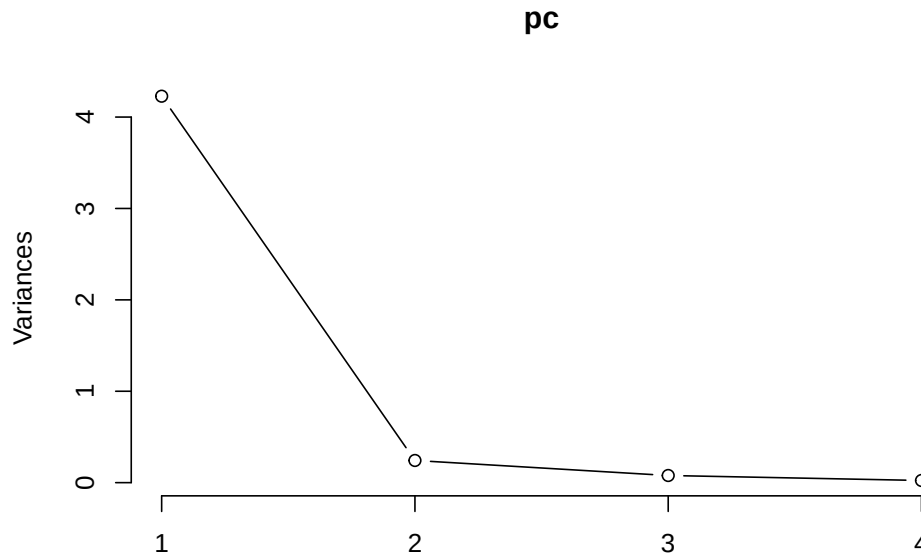
How important is each principal component can also be seen using a scree plot⁸. The plot function for the result of the `prcomp` function visualizes how much variability in the data is explained by each additional principal component.

⁶https://en.wikipedia.org/wiki/Principal_component_analysis

⁷https://mhahsler.github.io/Introduction_to_Data_Mining_R_Examples/book/data.html#dimensionality-reduction

⁸https://en.wikipedia.org/wiki/Scree_plot

```
plot(pc, type = "line")
```



Note that the first principal component (PC1) explains most of the variability in the iris dataset.

To find out what information is stored in the object `pc`, we can inspect the raw object (display *structure*).

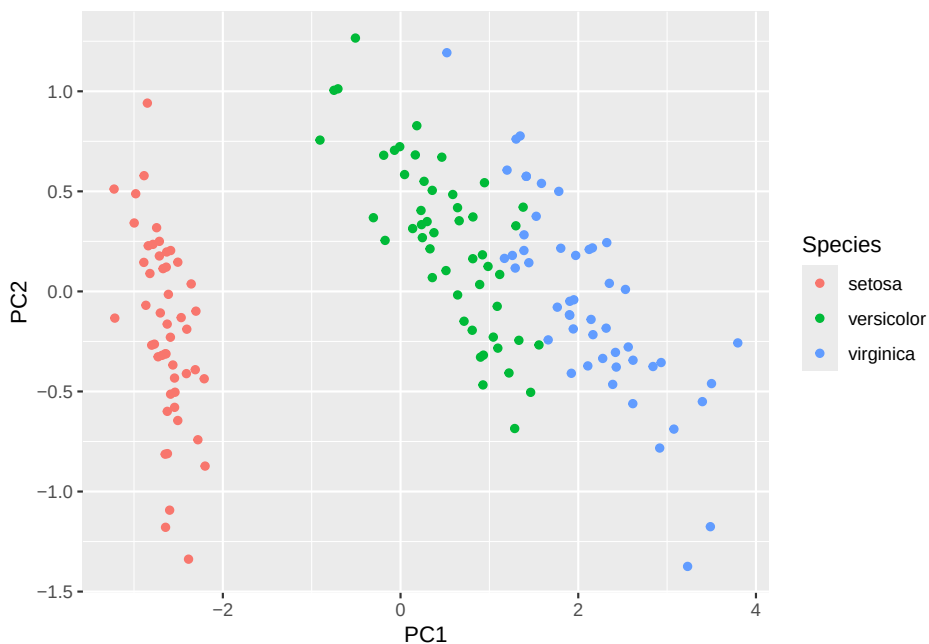
```
str(pc)
## List of 5
## $ sdev      : num [1:4] 2.056 0.493 0.28 0.154
## $ rotation: num [1:4, 1:4] 0.3614 -0.0845 0.8567 0.3583 -0.6566 ...
## ..- attr(*, "dimnames")=List of 2
## .. ..$ : chr [1:4] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width"
## .. ..$ : chr [1:4] "PC1" "PC2" "PC3" "PC4"
## $ center   : Named num [1:4] 5.84 3.06 3.76 1.2
## ..- attr(*, "names")= chr [1:4] "Sepal.Length" "Sepal.Width" "Petal.Length" "Petal.Width"
## $ scale     : logi FALSE
## $ x         : num [1:150, 1:4] -2.68 -2.71 -2.89 -2.75 -2.73 ...
## ..- attr(*, "dimnames")=List of 2
## .. ..$ : NULL
## .. ..$ : chr [1:4] "PC1" "PC2" "PC3" "PC4"
## - attr(*, "class")= chr "prcomp"
```

The object `pc` (like most objects in R) is a list with a class attribute. The list element `x` contains the data points projected on the principal components. We can convert the matrix into a tibble and add the species column from the

original dataset back (since the rows are in the same order), and then display the data projected on the first two principal components.

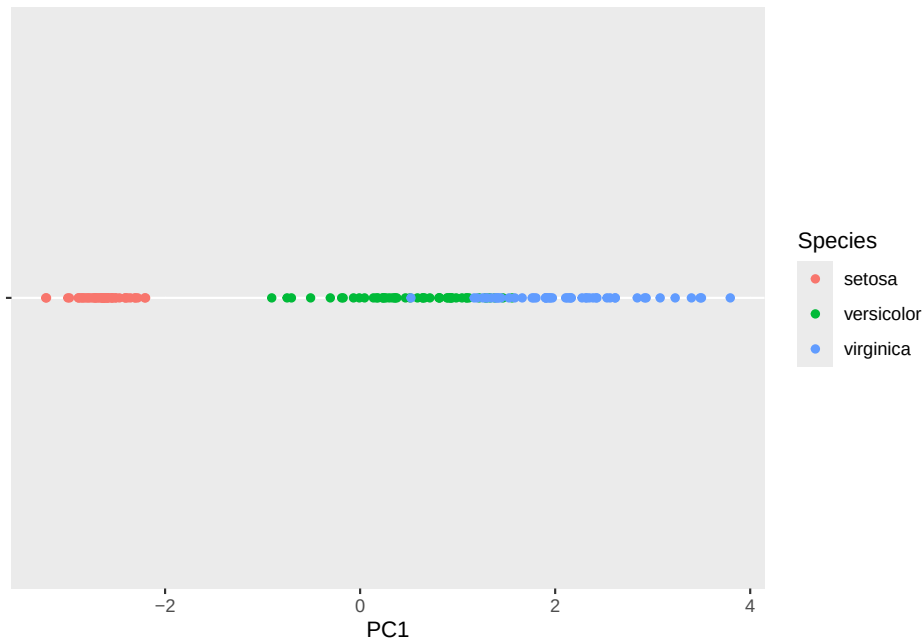
```
iris_projected <- as_tibble(pc$x) |>
  add_column(Species = iris$Species)

ggplot(iris_projected, aes(x = PC1, y = PC2, color = Species)) +
  geom_point()
```



Flowers that are displayed close together in this projection are also close together in the original 4-dimensional space. Since the first principal component represents most of the variability, we can also show the data projected only on PC1.

```
ggplot(iris_projected,
  aes(x = PC1, y = 0, color = Species)) +
  geom_point() +
  scale_y_continuous(expand=c(0,0)) +
  theme(axis.text.y = element_blank(),
    axis.title.y = element_blank())
)
```

We see that we can perfectly separate the species Setosa using just the first principal component. The other two species are harder to separate.

A plot of the projected data with the original axes added as arrows is called a biplot⁹. If the arrows (original axes) align roughly with the axes of the projection, then they are correlated (linearly dependent).

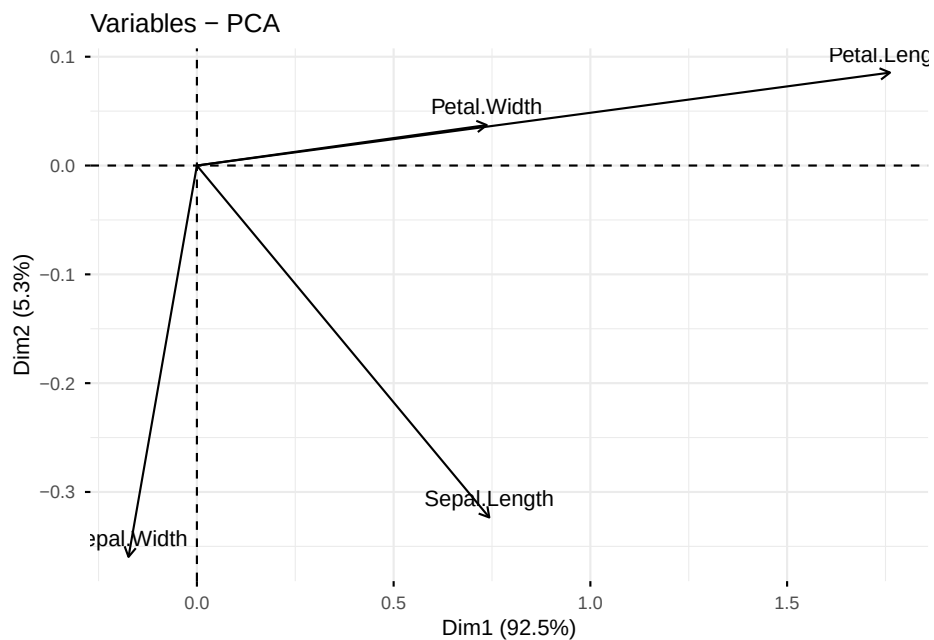
```
library(factoextra)
fviz_pca(pc)
## Warning: Using `size` aesthetic for lines was deprecated in ggplot2
## 3.4.0.
## i Please use `linewidth` instead.
## i The deprecated feature was likely used in the ggpubr
## package.
## Please report the issue at
## <https://github.com/kassambara/ggpubr/issues>.
## This warning is displayed once every 8 hours.
## Call `lifecycle::last_lifecycle_warnings()` to see where
## this warning was generated.
```

⁹<https://en.wikipedia.org/wiki/Biplot>



We can also display only the old and new axes.

```
fviz_pca_var(pc)
```



We see Petal.Width and Petal.Length point in the same direction which indi-

cates that they are highly correlated. They are also roughly aligned with PC1 (called Dim1 in the plot) which means that PC1 represents most of the variability of these two variables. Sepal.Width is almost aligned with the y-axis and therefore it is represented by PC2 (Dim2). Petal.Width/Petal.Length and Sepal.Width are almost at 90 degrees, indicating that they are close to uncorrelated. Sepal.Length is correlated to all other variables and represented by both, PC1 and PC2 in the projection.

2.3.3.2 Multi-Dimensional Scaling (MDS)

MDS¹⁰ is similar to PCA. Instead of data points, it starts with pairwise distances (i.e., a distance matrix) and produces a space where points are placed to represent these distances as well as possible. The axes in this space are called components and are similar to the principal components in PCA.

First, we calculate a distance matrix (Euclidean distances) from the 4-d space of the iris dataset.

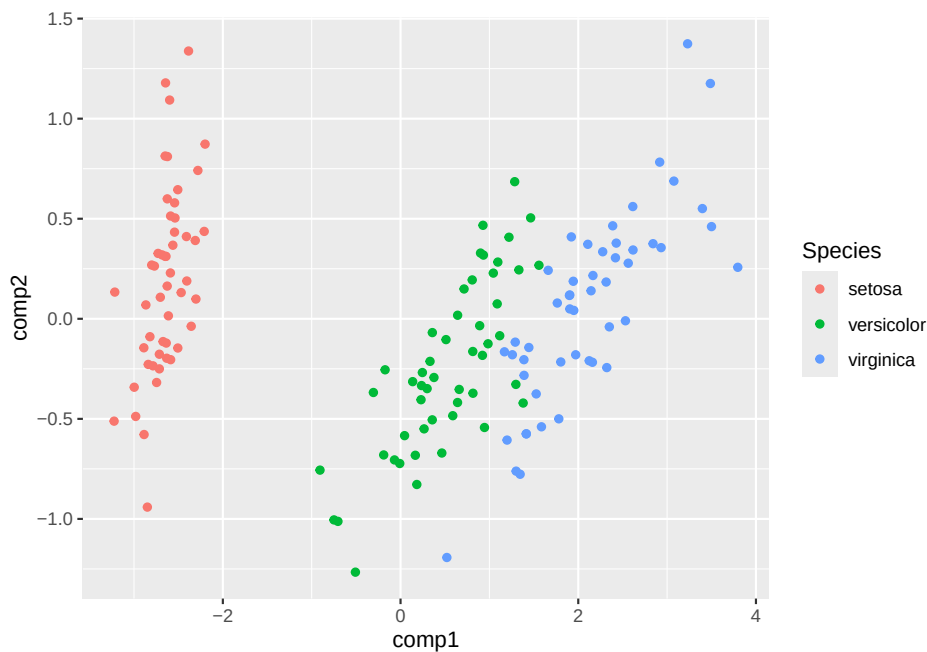
```
d <- iris |>
  select(-Species) |>
  dist()
```

Metric (classic) MDS tries to construct a space where points with lower distances are placed closer together. We project the data represented by a distance matrix on $k = 2$ dimensions.

```
fit <- cmdscale(d, k = 2)
colnames(fit) <- c("comp1", "comp2")
fit <- as_tibble(fit) |>
  add_column(Species = iris$Species)

ggplot(fit, aes(x = comp1, y = comp2, color = Species)) +
  geom_point()
```

¹⁰https://en.wikipedia.org/wiki/Multidimensional_scaling



The resulting projection is similar (except for rotation and reflection) to the result of the projection using PCA.

2.3.3.3 Non-Parametric Multidimensional Scaling

Non-parametric multidimensional scaling performs MDS while relaxing the need of linear relationships. Methods are available in package `MASS` as functions `isoMDS()` (implements isoMAP¹¹) and `sammon()`.

2.3.3.4 Embeddings: Nonlinear Dimensionality Reduction Methods

Nonlinear dimensionality reduction¹² is also called manifold learning or creating a low-dimensional embedding. These methods have become very popular to support visualizing high-dimensional data, mine text by converting words into numeric vectors that can be used as features in models, and as a method to automatically create features for data mining models by finding an efficient representation. Popular methods are:

- Visualization: t-distributed stochastic neighbor embedding¹³ (`Rtsne()` in package `Rtsne`) and uniform manifold approximation and projection

¹¹<https://en.wikipedia.org/wiki/Isomap>

¹²https://en.wikipedia.org/wiki/Nonlinear_dimensionality_reduction

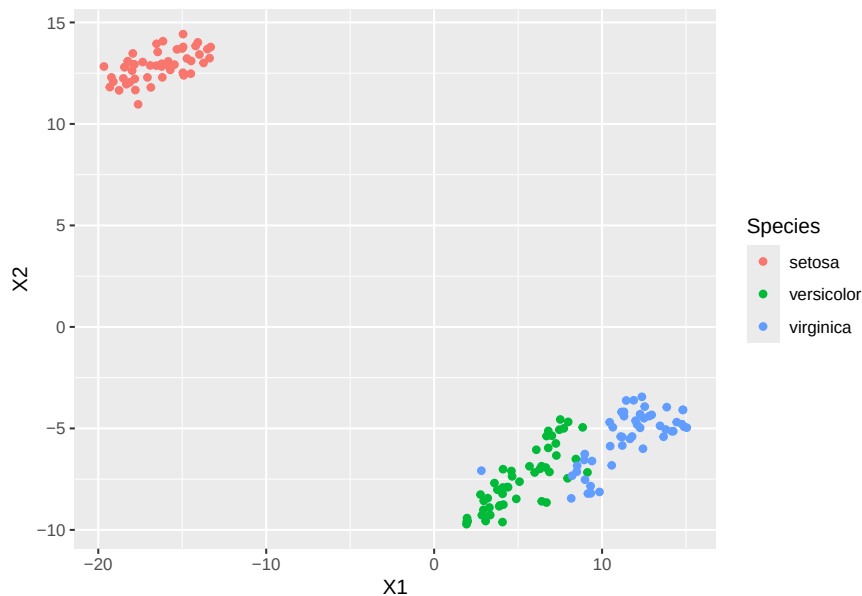
¹³https://en.wikipedia.org/wiki/T-distributed_stochastic_neighbor_embedding

(`umap()` in package `umap`) are used for projecting data to 2 dimensions for visualization. Here is an example of embedding the 4-dimensional iris data using `tsne`. The embedding algorithm requires unique data points.

```
iris_distinct <- iris |> distinct(Sepal.Length, Sepal.Width,
                                Petal.Length, Petal.Width,
                                .keep_all = TRUE)

tsne <- Rtsne::Rtsne(iris_distinct |> select(-Species))

emb <- data.frame(tsne$Y, Species = iris_distinct$Species)
ggplot(emb, aes(X1, X2, color = Species)) + geom_point()
```



We see that the embedding separates the class *Setosa* very well from the other two species indicating that the flowers are very different.

- Text mining: Word2vec¹⁴ (`word2vec()` in `word2vec`) is used in natural language processing¹⁵ to convert words to numeric vectors that represent similarities between words. These vectors can be used as features for data mining models for clustering and classification. Other popular methods for text embedding are GloVe¹⁶ and BERT¹⁷.
- Representation learning: Autoencoders¹⁸ are artificial neural networks

¹⁴<https://en.wikipedia.org/wiki/Word2vec>

¹⁵<https://cran.r-project.org/web/views/NaturalLanguageProcessing.html>

¹⁶<https://nlp.stanford.edu/projects/glove/>

¹⁷[https://en.wikipedia.org/wiki/BERT_\(language_model\)](https://en.wikipedia.org/wiki/BERT_(language_model))

¹⁸<https://en.wikipedia.org/wiki/Autoencoder>

used to learn an efficient representation (encoding) for a set of data by minimizing the reconstruction error. This new representation can be seen as a way to automatically creating features that describe the important characteristics of the data while reducing noise. This representation often helps models to learn better. In R, autoencoders are typically created using the `keras` package¹⁹. Creating an autoencoder requires some work. You will need to decide on the network topology and have sufficient training data.

2.3.4 Feature Subset Selection

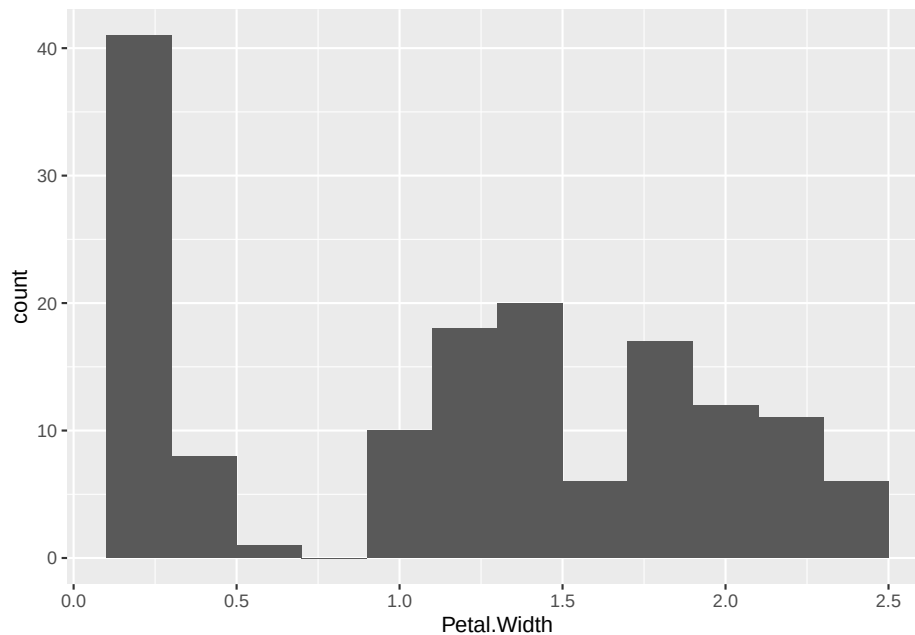
Feature selection is the process of identifying the features that are used to create a model. We will talk about feature selection when we discuss classification models in Chapter 3 in Feature Selection*.

2.3.5 Discretization

Some data mining methods require discrete data. Discretization converts continuous features into discrete features. As an example, we will discretize the continuous feature `Petal.Width`. Before we perform discretization, we should look at the distribution and see if it gives us an idea how we should group the continuous values into a set of discrete values. A histogram visualizes the distribution of a single continuous feature.

```
ggplot(iris, aes(x = Petal.Width)) +  
  geom_histogram(binwidth = .2)
```

¹⁹<https://keras3.posit.co/>



The bins in the histogram represent a discretization using a fixed bin width. The R function `cut()` performs equal interval width discretization creating a vector of type `factor` where each level represents an interval.

```
iris |>
  pull(Sepal.Width) |>
  cut(breaks = 3)
## [1] (2.8,3.6] (2.8,3.6] (2.8,3.6] (2.8,3.6] (2.8,3.6]
## [6] (3.6,4.4] (2.8,3.6] (2.8,3.6] (2.8,3.6] (2.8,3.6]
## [11] (3.6,4.4] (2.8,3.6] (2.8,3.6] (2.8,3.6] (3.6,4.4]
## [16] (3.6,4.4] (3.6,4.4] (2.8,3.6] (3.6,4.4] (3.6,4.4]
## [21] (2.8,3.6] (3.6,4.4] (2.8,3.6] (2.8,3.6] (2.8,3.6]
## [26] (2.8,3.6] (2.8,3.6] (2.8,3.6] (2.8,3.6] (2.8,3.6]
## [31] (2.8,3.6] (2.8,3.6] (3.6,4.4] (3.6,4.4] (2.8,3.6]
## [36] (2.8,3.6] (2.8,3.6] (2.8,3.6] (2.8,3.6] (2.8,3.6]
## [41] (2.8,3.6] (2,2.8] (2.8,3.6] (2.8,3.6] (3.6,4.4]
## [46] (2.8,3.6] (3.6,4.4] (2.8,3.6] (3.6,4.4] (2.8,3.6]
## [51] (2.8,3.6] (2.8,3.6] (2.8,3.6] (2,2.8] (2,2.8]
## [56] (2,2.8] (2.8,3.6] (2,2.8] (2.8,3.6] (2,2.8]
## [61] (2,2.8] (2.8,3.6] (2,2.8] (2.8,3.6] (2.8,3.6]
## [66] (2.8,3.6] (2.8,3.6] (2,2.8] (2,2.8] (2,2.8]
## [71] (2.8,3.6] (2,2.8] (2,2.8] (2,2.8] (2.8,3.6]
## [76] (2.8,3.6] (2,2.8] (2.8,3.6] (2.8,3.6] (2,2.8]
## [81] (2,2.8] (2,2.8] (2,2.8] (2,2.8] (2.8,3.6]
## [86] (2.8,3.6] (2.8,3.6] (2,2.8] (2.8,3.6] (2,2.8]
```

```
## [91] (2,2.8] (2.8,3.6] (2,2.8] (2,2.8] (2,2.8]
## [96] (2.8,3.6] (2.8,3.6] (2.8,3.6] (2,2.8] (2,2.8]
## [101] (2.8,3.6] (2,2.8] (2.8,3.6] (2.8,3.6] (2.8,3.6]
## [106] (2.8,3.6] (2,2.8] (2.8,3.6] (2,2.8] (2.8,3.6]
## [111] (2.8,3.6] (2,2.8] (2.8,3.6] (2,2.8] (2,2.8]
## [116] (2.8,3.6] (2.8,3.6] (3.6,4.4] (2,2.8] (2,2.8]
## [121] (2.8,3.6] (2,2.8] (2,2.8] (2,2.8] (2.8,3.6]
## [126] (2.8,3.6] (2,2.8] (2.8,3.6] (2,2.8] (2.8,3.6]
## [131] (2,2.8] (3.6,4.4] (2,2.8] (2,2.8] (2,2.8]
## [136] (2.8,3.6] (2.8,3.6] (2.8,3.6] (2.8,3.6] (2.8,3.6]
## [141] (2.8,3.6] (2.8,3.6] (2,2.8] (2.8,3.6] (2.8,3.6]
## [146] (2.8,3.6] (2,2.8] (2.8,3.6] (2.8,3.6] (2.8,3.6]
## Levels: (2,2.8] (2.8,3.6] (3.6,4.4]
```

Other discretization methods include equal frequency discretization or using k-means clustering. These methods are implemented by several R packages. We use here the implementation in package `arules` and visualize the results as histograms with blue lines to separate intervals assigned to each discrete value.

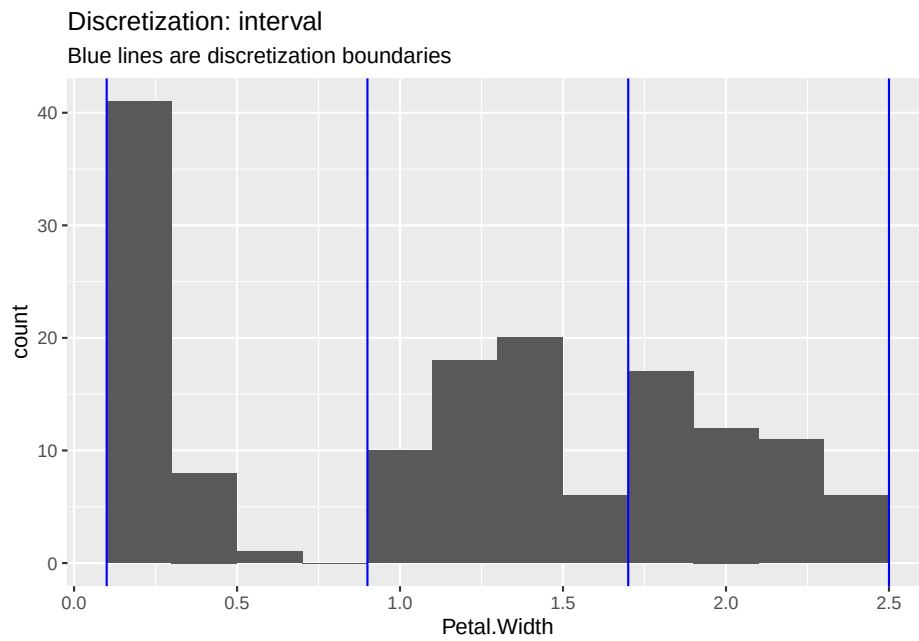
```
library(arules)
## Loading required package: Matrix
##
## Attaching package: 'Matrix'
## The following objects are masked from 'package:tidyr':
##
##   expand, pack, unpack
##
## Attaching package: 'arules'
## The following object is masked from 'package:dplyr':
##
##   recode
## The following objects are masked from 'package:base':
##
##   abbreviate, write
iris |> pull(Petal.Width) |>
  discretize(method = "interval", breaks = 3)
## [1] [0.1,0.9) [0.1,0.9) [0.1,0.9) [0.1,0.9) [0.1,0.9)
## [6] [0.1,0.9) [0.1,0.9) [0.1,0.9) [0.1,0.9) [0.1,0.9)
## [11] [0.1,0.9) [0.1,0.9) [0.1,0.9) [0.1,0.9) [0.1,0.9)
## [16] [0.1,0.9) [0.1,0.9) [0.1,0.9) [0.1,0.9) [0.1,0.9)
## [21] [0.1,0.9) [0.1,0.9) [0.1,0.9) [0.1,0.9) [0.1,0.9)
## [26] [0.1,0.9) [0.1,0.9) [0.1,0.9) [0.1,0.9) [0.1,0.9)
## [31] [0.1,0.9) [0.1,0.9) [0.1,0.9) [0.1,0.9) [0.1,0.9)
## [36] [0.1,0.9) [0.1,0.9) [0.1,0.9) [0.1,0.9) [0.1,0.9)
## [41] [0.1,0.9) [0.1,0.9) [0.1,0.9) [0.1,0.9) [0.1,0.9)
```



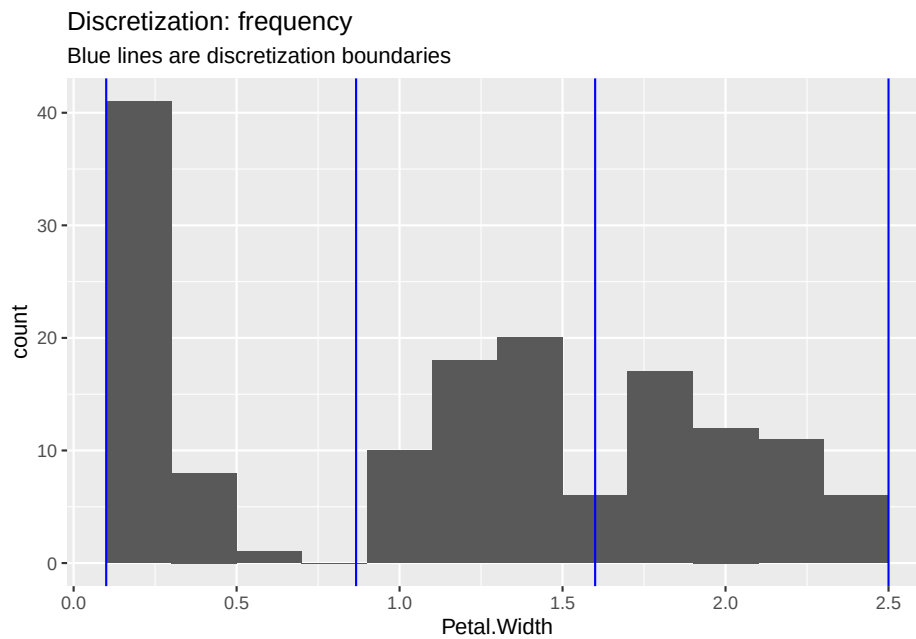
```
## [46] [0.1,0.9) [0.1,0.9) [0.1,0.9) [0.1,0.9) [0.1,0.9)
## [51] [0.9,1.7) [0.9,1.7) [0.9,1.7) [0.9,1.7) [0.9,1.7)
## [56] [0.9,1.7) [0.9,1.7) [0.9,1.7) [0.9,1.7) [0.9,1.7)
## [61] [0.9,1.7) [0.9,1.7) [0.9,1.7) [0.9,1.7) [0.9,1.7)
## [66] [0.9,1.7) [0.9,1.7) [0.9,1.7) [0.9,1.7) [0.9,1.7)
## [71] [1.7,2.5] [0.9,1.7) [0.9,1.7) [0.9,1.7) [0.9,1.7)
## [76] [0.9,1.7) [0.9,1.7) [1.7,2.5] [0.9,1.7) [0.9,1.7)
## [81] [0.9,1.7) [0.9,1.7) [0.9,1.7) [0.9,1.7) [0.9,1.7)
## [86] [0.9,1.7) [0.9,1.7) [0.9,1.7) [0.9,1.7) [0.9,1.7)
## [91] [0.9,1.7) [0.9,1.7) [0.9,1.7) [0.9,1.7) [0.9,1.7)
## [96] [0.9,1.7) [0.9,1.7) [0.9,1.7) [0.9,1.7) [0.9,1.7)
## [101] [1.7,2.5] [1.7,2.5] [1.7,2.5] [1.7,2.5] [1.7,2.5]
## [106] [1.7,2.5] [1.7,2.5] [1.7,2.5] [1.7,2.5] [1.7,2.5]
## [111] [1.7,2.5] [1.7,2.5] [1.7,2.5] [1.7,2.5] [1.7,2.5]
## [116] [1.7,2.5] [1.7,2.5] [1.7,2.5] [1.7,2.5] [0.9,1.7)
## [121] [1.7,2.5] [1.7,2.5] [1.7,2.5] [1.7,2.5] [1.7,2.5]
## [126] [1.7,2.5] [1.7,2.5] [1.7,2.5] [1.7,2.5] [0.9,1.7)
## [131] [1.7,2.5] [1.7,2.5] [1.7,2.5] [0.9,1.7) [0.9,1.7)
## [136] [1.7,2.5] [1.7,2.5] [1.7,2.5] [1.7,2.5] [1.7,2.5]
## [141] [1.7,2.5] [1.7,2.5] [1.7,2.5] [1.7,2.5] [1.7,2.5]
## [146] [1.7,2.5] [1.7,2.5] [1.7,2.5] [1.7,2.5] [1.7,2.5]
## attr(,"discretized:breaks")
## [1] 0.1 0.9 1.7 2.5
## attr(,"discretized:method")
## [1] interval
## Levels: [0.1,0.9) [0.9,1.7) [1.7,2.5]
```

To show the differences between the methods, we use the three discretization methods and draw blue lines in the histogram to show how they cut the data.

```
ggplot(iris, aes(Petal.Width)) + geom_histogram(binwidth = .2) +
  geom_vline(
    xintercept = iris |>
      pull(Petal.Width) |>
      discretize(method = "interval", breaks = 3, onlycuts = TRUE),
    color = "blue"
  ) +
  labs(title = "Discretization: interval",
       subtitle = "Blue lines are discretization boundaries")
```



```
ggplot(iris, aes(Petal.Width)) + geom_histogram(binwidth = .2) +  
  geom_vline(  
    xintercept = iris |>  
    pull(Petal.Width) |>  
    discretize(method = "frequency", breaks = 3, onlycuts = TRUE),  
    color = "blue"  
  ) +  
  labs(title = "Discretization: frequency",  
       subtitle = "Blue lines are discretization boundaries")
```



```
ggplot(iris, aes(Petal.Width)) + geom_histogram(binwidth = .2) +  
  geom_vline(  
    xintercept = iris |>  
    pull(Petal.Width) |>  
    discretize(method = "cluster", breaks = 3, onlycuts = TRUE),  
    color = "blue"  
  ) +  
  labs(title = "Discretization: cluster",  
        subtitle = "Blue lines are discretization boundaries")
```



The user needs to decide on the number of intervals and the used method.

2.3.6 Variable Transformation: Standardization

Standardizing (scaling, normalizing) the range of features values is important to make them comparable. The most popular method is to convert the values of each feature to z-scores²⁰, by subtracting the mean (centering) and dividing by the standard deviation (scaling). The standardized feature will have a mean of zero and are measured in standard deviations from the mean. Positive values indicate how many standard deviation the original feature value was above the average. Negative standardized values indicate below-average values.

R-base provides the function `scale()` to standardize the columns in a `data.frame`. Tidyverse currently does not have a simple `scale` function, so we make one. It mutates all numeric columns using an anonymous function that calculates the z-score.

```
scale_numeric <- function(x)
  x |>
  mutate(across(where(is.numeric),
    function(y) (y - mean(y, na.rm = TRUE)) / sd(y, na.rm = TRUE)))
```

²⁰https://en.wikipedia.org/wiki/Standard_score

```
iris.scaled <- iris |>
  scale_numeric()
iris.scaled
## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##   <dbl>        <dbl>        <dbl>        <dbl> <fct>
## 1    -0.898         1.02         -1.34         -1.31 setosa
## 2    -1.14        -0.132         -1.34         -1.31 setosa
## 3    -1.38         0.327         -1.39         -1.31 setosa
## 4    -1.50         0.0979         -1.28         -1.31 setosa
## 5    -1.02         1.25         -1.34         -1.31 setosa
## 6   -0.535         1.93         -1.17         -1.05 setosa
## 7    -1.50         0.786         -1.34         -1.18 setosa
## 8    -1.02         0.786         -1.28         -1.31 setosa
## 9    -1.74        -0.361         -1.34         -1.31 setosa
## 10   -1.14         0.0979         -1.28         -1.44 setosa
## # i 140 more rows
summary(iris.scaled)
##   Sepal.Length      Sepal.Width      Petal.Length
## Min.   :-1.8638   Min.   :-2.426   Min.   :-1.562
## 1st Qu.: -0.8977   1st Qu.: -0.590   1st Qu.: -1.222
## Median : -0.0523   Median : -0.132   Median :  0.335
## Mean   :  0.0000   Mean   :  0.000   Mean   :  0.000
## 3rd Qu.:  0.6722   3rd Qu.:  0.557   3rd Qu.:  0.760
## Max.   :  2.4837   Max.   :  3.080   Max.   :  1.780
##   Petal.Width      Species
## Min.   :-1.442   setosa    :50
## 1st Qu.: -1.180   versicolor:50
## Median :  0.132   virginica :50
## Mean   :  0.000
## 3rd Qu.:  0.788
## Max.   :  1.706
```

The standardized feature has a mean of zero and most “normal” values will fall in the range $[-3, 3]$ and is measured in standard deviations from the average. Negative values mean smaller than the average and positive values mean larger than the average.

2.4 Measures of Similarity and Dissimilarity

Proximities help with quantifying how similar two objects are. Remember, objects are rows in our data. We call one row the feature vector for the object.

Similarity²¹ is a concept from geometry. It is the opposite of dissimilarity. The best-known way to measure dissimilarity is Euclidean distance²², but dissimilarity and similarity²³ can be measured in many different ways depending on the information we have about the objects.

R stores similarity information always as a dissimilarities/distances matrices. Similarities are converted to dissimilarities. Distances are symmetric, i.e., the distance from A to B is the same as the distance from B to A. R therefore stores only a triangle (typically the lower triangle) of the distance matrix.

2.4.1 Minkowsky Distances

The Minkowsky distance²⁴ is a family of distances including Euclidean and Manhattan distance. It is defined between two feature vectors $\mathbf{p} = (p_1, p_2, \dots, p_n)$ and $\mathbf{q} = (q_1, q_2, \dots, q_n)$ as

$$d(\mathbf{p}, \mathbf{q}) = \left(\sum_{i=1}^n |p_i - q_i|^r \right)^{\frac{1}{r}} = \|\mathbf{p} - \mathbf{q}\|_r.$$

Where the power r is a positive integer. This type of distance is also called a r -norm written as L^r . Special values for r are:

- $r = 1$: Manhattan distance
- $r = 2$: Euclidean distance
- $r = \infty$: Maximum norm (only the largest component distance counts)

To avoid one feature to dominate the distance calculation, scaled data is typically used. We select the first 5 flowers for this example.

```
iris_sample <- iris.scaled |>
  select(-Species) |>
  slice(1:5)
iris_sample
## # A tibble: 5 x 4
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
##   <dbl>         <dbl>         <dbl>         <dbl>
## 1    -0.898         1.02         -1.34         -1.31
## 2    -1.14        -0.132        -1.34         -1.31
## 3    -1.38         0.327        -1.39         -1.31
## 4    -1.50         0.0979       -1.28         -1.31
## 5    -1.02         1.25         -1.34         -1.31
```

²¹[https://en.wikipedia.org/wiki/Similarity_\(geometry\)](https://en.wikipedia.org/wiki/Similarity_(geometry))

²²https://en.wikipedia.org/wiki/Euclidean_distance

²³https://en.wikipedia.org/wiki/Similarity_measure

²⁴https://en.wikipedia.org/wiki/Minkowski_distance

Different types of Minkowsky distance matrices between the first 5 flowers can be calculated using `dist()`.

```
dist(iris_sample, method = "euclidean")
##          1          2          3          4
## 2 1.1723
## 3 0.8428 0.5216
## 4 1.1000 0.4326 0.2829
## 5 0.2593 1.3819 0.9883 1.2460
dist(iris_sample, method = "manhattan")
##          1          2          3          4
## 2 1.3887
## 3 1.2280 0.7570
## 4 1.5782 0.6484 0.4635
## 5 0.3502 1.4973 1.3367 1.6868
dist(iris_sample, method = "maximum")
##          1          2          3          4
## 2 1.1471
## 3 0.6883 0.4589
## 4 0.9177 0.3623 0.2294
## 5 0.2294 1.3766 0.9177 1.1471
```

We see that only the lower triangle of the distance matrices are stored (note that rows start with row 2).

2.4.2 Distances for Binary Data

Binary data can be encoded as 0 and 1 (numeric) or TRUE and FALSE (logical).

```
b <- rbind(
  c(0,0,0,1,1,1,1,0,0,1),
  c(0,0,1,1,1,0,0,1,0,0)
)
b
##          [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]      0      0      0      1      1      1      1      0      0      1
## [2,]      0      0      1      1      1      0      0      1      0      0
b_logical <- apply(b, MARGIN = 2, as.logical)
b_logical
##          [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
## [1,] FALSE FALSE FALSE TRUE TRUE TRUE TRUE FALSE FALSE
## [2,] FALSE FALSE TRUE TRUE TRUE FALSE FALSE TRUE FALSE
##          [,10]
## [1,] TRUE
## [2,] FALSE
```

2.4.2.1 Hamming Distance

The Hamming distance²⁵ is the number of mismatches between two binary vectors. For 0-1 data this is equivalent to the Manhattan distance and also the squared Euclidean distance.

```
dist(b, method = "manhattan")
##      1
## 2 5
dist(b, method = "euclidean")^2
##      1
## 2 5
```

2.4.2.2 Jaccard Index

The Jaccard index²⁶ is a similarity measure that focuses on matching 1s. R converts the similarity into a dissimilarity using $d_J = 1 - s_J$.

```
dist(b, method = "binary")
##      1
## 2 0.7143
```

2.4.3 Distances for Mixed Data

Most distance measures work only on numeric data. Often, we have a mixture of numbers and nominal or ordinal features like this data:

```
people <- tibble(
  height = c(160, 185, 170),
  weight = c(52, 90, 75),
  sex    = c("female", "male", "male")
)
people
## # A tibble: 3 x 3
##   height weight sex
##   <dbl> <dbl> <chr>
## 1   160    52 female
## 2   185    90  male
## 3   170    75  male
```

It is important that nominal features are stored as factors and not character (<chr>).

²⁵https://en.wikipedia.org/wiki/Hamming_distance

²⁶https://en.wikipedia.org/wiki/Jaccard_index


```
people <- people |>
  mutate(across(where(is.character), factor))
people
## # A tibble: 3 x 3
##   height weight sex
##   <dbl>   <dbl> <fct>
## 1    160     52 female
## 2    185     90 male
## 3    170     75 male
```

2.4.3.1 Gower's Coefficient

The Gower's coefficient of similarity works with mixed data by calculating the appropriate similarity for each feature and then aggregating them into a single measure. The package `proxy` implements Gower's coefficient converted into a distance.

```
library(proxy)
##
## Attaching package: 'proxy'
## The following object is masked from 'package:Matrix':
##
##   as.matrix
## The following objects are masked from 'package:stats':
##
##   as.dist, dist
## The following object is masked from 'package:base':
##
##   as.matrix
d_Gower <- dist(people, method = "Gower")
d_Gower
##           1           2
## 2 1.0000
## 3 0.6684 0.3316
```

Gower's coefficient calculation implicitly scales the data because it calculates distances on each feature individually, so there is no need to scale the data first.

2.4.3.2 Using Euclidean Distance with Mixed Data

Sometimes methods (e.g., k-means) can only use Euclidean distance. In this case, nominal features can be converted into 0-1 dummy variables. After scaling, Euclidean distance will result in a usable distance measure.

We use package `caret` to create dummy variables.

```
library(caret)
## Loading required package: lattice
##
## Attaching package: 'caret'
## The following object is masked from 'package:purrr':
##
## lift
data_dummy <- dummyVars(~., people) |>
  predict(people)
data_dummy
## height weight sex.female sex.male
## 1 160 52 1 0
## 2 185 90 0 1
## 3 170 75 0 1
```

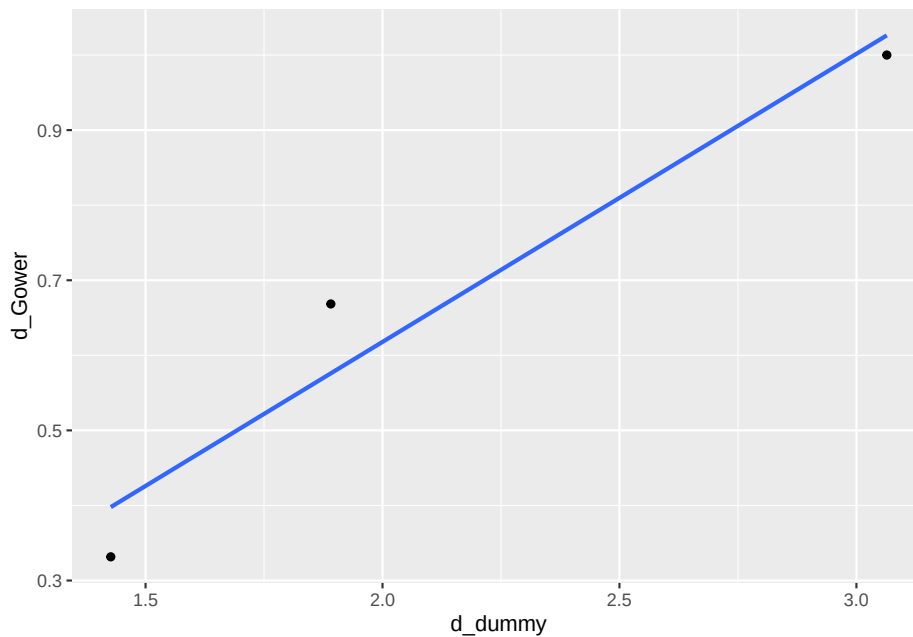
Note that feature `sex` has now two columns. If we want that height, weight and sex have the same influence on the distance measure, then we need to weight the sex columns by $1/2$ after scaling.

```
weight_matrix <- matrix(c(1, 1, 1/2, 1/2),
                        ncol = 4,
                        nrow = nrow(data_dummy),
                        byrow = TRUE)
data_dummy_scaled <- scale(data_dummy) * weight_matrix

d_dummy <- dist(data_dummy_scaled)
d_dummy
## 1 2
## 2 3.064
## 3 1.891 1.427
```

The distance using dummy variables is consistent with Gower's distance. However, note that Gower's distance is scaled between 0 and 1 while the Euclidean distance is not.

```
ggplot(tibble(d_dummy, d_Gower), aes(x = d_dummy, y = d_Gower)) +
  geom_point() +
  geom_smooth(method = "lm", se = FALSE)
## Don't know how to automatically pick scale for object of
## type <dist>. Defaulting to continuous.
## Don't know how to automatically pick scale for object of
## type <dist>. Defaulting to continuous.
## `geom_smooth()` using formula = 'y ~ x'
```



2.4.4 More Proximity Measures

The package `proxy` implements a wide array of proximity measures (similarity measures are converted into distances).

```
library(proxy)
pr_DB$get_entry_names()
## [1] "Jaccard"          "Kulczynski1"      "Kulczynski2"
## [4] "Mountford"       "Fager"           "Russel"
## [7] "simple matching"  "Hamman"          "Faith"
## [10] "Tanimoto"        "Dice"            "Phi"
## [13] "Stiles"          "Michael"         "Mozley"
## [16] "Yule"            "Yule2"           "Ochiai"
## [19] "Simpson"         "Braun-Blanquet"  "cosine"
## [22] "angular"         "eJaccard"        "eDice"
## [25] "correlation"     "Chi-squared"     "Phi-squared"
## [28] "Tschuprow"      "Cramer"          "Pearson"
## [31] "Gower"          "Euclidean"       "Mahalanobis"
## [34] "Bhattacharyya"  "Manhattan"       "supremum"
## [37] "Minkowski"      "Canberra"        "Wave"
## [40] "divergence"     "Kullback"        "Bray"
## [43] "Soergel"        "Levenshtein"     "Podani"
## [46] "Chord"          "Geodesic"        "Whittaker"
## [49] "Hellinger"      "fJaccard"
```

Note that loading the package `proxy` overwrites the default `dist()` function in R. You can specify which `dist` function to use by specifying the package in the call. For example `stats::dist()` calls the default function in R (the package `stats` is part of R) while `proxy::dist()` calls the version in the package `proxy`.

2.5 Exercises*

The R package `palmerpenguins` contains measurements for penguin of different species from the Palmer Archipelago, Antarctica. Install the package. It provides a CSV file which can be read in the following way:

```
library("palmerpenguins")
##
## Attaching package: 'palmerpenguins'
## The following objects are masked from 'package:datasets':
##
##   penguins, penguins_raw
penguins <- read_csv(path_to_file("penguins.csv"))
## Rows: 344 Columns: 8
## -- Column specification -----
## Delimiter: ","
## chr (3): species, island, sex
## dbl (5): bill_length_mm, bill_depth_mm, flipper_length_m...
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
head(penguins)
## # A tibble: 6 x 8
##   species island   bill_length_mm bill_depth_mm
##   <chr>   <chr>         <dbl>         <dbl>
## 1 Adelie Torgersen     39.1           18.7
## 2 Adelie Torgersen     39.5           17.4
## 3 Adelie Torgersen     40.3            18
## 4 Adelie Torgersen      NA            NA
## 5 Adelie Torgersen     36.7           19.3
## 6 Adelie Torgersen     39.3           20.6
## # i 4 more variables: flipper_length_mm <dbl>,
## #   body_mass_g <dbl>, sex <chr>, year <dbl>
```

Create in RStudio a new R Markdown document. Apply the code in the sections of this chapter to the data set to answer the following questions.

1. What is the scale of measurement for each column?

2. Are there missing values in the data? How much data is missing?
3. Compute and discuss basic statistics.
4. Calculate the similarity between five randomly chosen penguins. Do you need to scale the data? Discuss what measures are appropriate for the data.

Make sure your markdown document contains now a well formatted report. Use the `Knit` button to create a HTML document.

Chapter 3

Classification: Basic Concepts

This chapter introduces decision trees for classification and discusses how models are built and evaluated.

The corresponding chapter of the data mining textbook is available online: Chapter 3: Classification: Basic Concepts and Techniques.¹

Packages Used in this Chapter

```
pkgs <- c("basemodels", "caret", "FSelector", "lattice", "mlbench",
          "palmerpenguins", "party", "pROC", "rpart",
          "rpart.plot", "tidyverse")

pkgs_install <- pkgs[!(pkgs %in% installed.packages()[,"Package"])]
if(length(pkgs_install)) install.packages(pkgs_install)
```

The packages used for this chapter are:

- *basemodels* (Y.-J. Chen et al. 2023)
- *caret* (Kuhn 2024)
- *FSelector* (Romanski, Kotthoff, and Schratz 2023)
- *lattice* (Sarkar 2025)
- *mlbench* (Leisch and Dimitriadou 2024)
- *palmerpenguins* (Horst, Hill, and Gorman 2022)

¹https://mhahsler.github.io/Introduction_to_Data_Mining_R_Examples/DM_chapters/ch3_classification.pdf

- *party* (Hothorn et al. 2025)
- *pROC* (Robin et al. 2025)
- *rpart* (Therneau and Atkinson 2025)
- *rpart.plot* (Milborrow 2025)
- *tidyverse* (Wickham 2023b)

In the examples in this book, we use the popular machine learning R package **caret**². It makes preparing training sets, building classification (and regression) models and evaluation easier. A great cheat sheet can be found here³.

A newer R framework for machine learning is **tidymodels**⁴, a set of packages that integrate more naturally with tidyverse. Using tidymodels, or any other framework (e.g., Python’s scikit-learn⁵) should be relatively easy after learning the concepts using caret.

3.1 Basic Concepts

Classification is a machine learning task with the goal to learn a predictive function of the form

$$y = f(\mathbf{x}),$$

where $\mathbf{x}$ is called the attribute set and y the class label. The attribute set consists of feature which describe an object. These features can be measured using any scale (i.e., nominal, interval, ...). The class label is a nominal attribute. If it is a binary attribute, then the problem is called a binary classification problem.

Classification learns the classification model from training data where both the features and the correct class label are available. This is why it is called a supervised learning problem⁶.

A related supervised learning problem is regression⁷, where y is a number instead of a label. Linear regression is a very popular supervised learning model which is taught in almost any introductory statistics course. Code examples for regression are available in the extra Chapter Regression.

This chapter will introduce decision trees, model evaluation and comparison, feature selection, and then explore methods to handle the class imbalance problem.

²<https://topepo.github.io/caret/>

³https://ugoproto.github.io/ugo_r_doc/pdf/caret.pdf

⁴<https://www.tidymodels.org/>

⁵<https://scikit-learn.org/>

⁶https://en.wikipedia.org/wiki/Supervised_learning

⁷https://en.wikipedia.org/wiki/Linear_regression

You can read the free sample chapter from the textbook (Tan, Steinbach, and Kumar 2005): Chapter 3. Classification: Basic Concepts and Techniques⁸

3.2 General Framework for Classification

Supervised learning has two steps:

1. Induction: Training a model on **training data** with known class labels.
2. Deduction: Predicting class labels for new data.

We often test model by predicting the class for data where we know the correct label. We test the model on **test data** with known labels and can then calculate the error by comparing the prediction with the known correct label. It is tempting to measure how well the model has learned the training data, by testing it on the training data. The error on the training data is called **resubstitution error**. It does not help us to find out if the model generalizes well to new data that was not part of the training.

We typically want to evaluate how well the model generalizes new data, so it is important that the test data and the training data do not overlap. We call the error on proper test data the **generalization error**.

This chapter builds up the needed concepts. A complete example of how to perform model selection and estimate the generalization error is in the section Hyperparameter Tuning.

3.2.1 The Zoo Dataset

To demonstrate classification, we will use the Zoo dataset which is included in the R package **mlbench** (you may have to install it). The Zoo dataset containing 17 (mostly logical) variables for 101 animals as a data frame with 17 columns (hair, feathers, eggs, milk, airborne, aquatic, predator, toothed, backbone, breathes, venomous, fins, legs, tail, domestic, catsize, type). The first 16 columns represent the feature vector $\mathbf{x}$ and the last column called type is the class label y . We convert the data frame into a tidyverse tibble (optional).

```
data(Zoo, package="mlbench")
head(Zoo)
##           hair feathers  eggs  milk airborne aquatic
## aardvark  TRUE     FALSE FALSE  TRUE     FALSE  FALSE
## antelope  TRUE     FALSE FALSE  TRUE     FALSE  FALSE
```

⁸https://mhahsler.github.io/Introduction_to_Data_Mining_R_Examples/DM_chapters/ch3_classification.pdf

```
## bass      FALSE      FALSE TRUE FALSE      FALSE      TRUE
## bear      TRUE       FALSE FALSE TRUE      FALSE      FALSE
## boar      TRUE       FALSE FALSE TRUE      FALSE      FALSE
## buffalo   TRUE       FALSE FALSE TRUE      FALSE      FALSE
##           predator toothed backbone breathes venomous fins
## aardvark   TRUE       TRUE      TRUE      TRUE      FALSE FALSE
## antelope   FALSE      TRUE      TRUE      TRUE      FALSE FALSE
## bass       TRUE       TRUE      TRUE      FALSE     FALSE TRUE
## bear       TRUE       TRUE      TRUE      TRUE      FALSE FALSE
## boar       TRUE       TRUE      TRUE      TRUE      FALSE FALSE
## buffalo    FALSE      TRUE      TRUE      TRUE      FALSE FALSE
##           legs tail domestic catsize type
## aardvark   4 FALSE      FALSE      TRUE mammal
## antelope   4 TRUE       FALSE      TRUE mammal
## bass       0 TRUE       FALSE     FALSE fish
## bear       4 FALSE      FALSE      TRUE mammal
## boar       4 TRUE       FALSE      TRUE mammal
## buffalo    4 TRUE       FALSE      TRUE mammal
```

Note: data.frames in R can have row names. The Zoo data set uses the animal name as the row names. tibbles from `tidyverse` do not support row names. To keep the animal name you can add a column with the animal name.

```
library(tidyverse)
Zoo <- as_tibble(Zoo, rownames = "animal")
Zoo
## # A tibble: 101 x 18
##   animal hair feathers eggs milk airborne aquatic
##   <chr>   <lgl> <lgl>   <lgl> <lgl> <lgl>   <lgl>
## 1 aardvark TRUE  FALSE  FALSE TRUE  FALSE  FALSE
## 2 antelope TRUE  FALSE  FALSE TRUE  FALSE  FALSE
## 3 bass    FALSE FALSE  TRUE  FALSE FALSE  TRUE
## 4 bear    TRUE  FALSE  FALSE TRUE  FALSE  FALSE
## 5 boar    TRUE  FALSE  FALSE TRUE  FALSE  FALSE
## 6 buffalo TRUE  FALSE  FALSE TRUE  FALSE  FALSE
## 7 calf    TRUE  FALSE  FALSE TRUE  FALSE  FALSE
## 8 carp    FALSE FALSE  TRUE  FALSE FALSE  TRUE
## 9 catfish FALSE FALSE  TRUE  FALSE FALSE  TRUE
## 10 cavy   TRUE  FALSE  FALSE TRUE  FALSE  FALSE
## # i 91 more rows
## # i 11 more variables: predator <lgl>, toothed <lgl>,
## # backbone <lgl>, breathes <lgl>, venomous <lgl>,
## # fins <lgl>, legs <int>, tail <lgl>, domestic <lgl>,
## # catsize <lgl>, type <fct>
```

You will have to remove the animal column before learning a model since it is a unique identifier!

I translate all the TRUE/FALSE values into factors (nominal). This is often needed for building models. Always check `summary()` to make sure the data is ready for model learning.

```
Zoo <- Zoo |>
  mutate(across(where(is.logical),
    function (x) factor(x, levels = c(TRUE, FALSE)))) |>
  mutate(across(where(is.character), factor))

summary(Zoo)
##      animal      hair      feathers      eggs      milk
##  aardvark: 1   TRUE :43   TRUE :20   TRUE :59   TRUE :41
##  antelope: 1  FALSE:58  FALSE:81  FALSE:42  FALSE:60
##  bass      : 1
##  bear      : 1
##  boar      : 1
##  buffalo   : 1
##  (Other)   :95
##      airborne  aquatic  predator  toothed  backbone
##  TRUE :24   TRUE :36   TRUE :56   TRUE :61   TRUE :83
##  FALSE:77  FALSE:65  FALSE:45  FALSE:40  FALSE:18
##
##
##
##
##      breathes  venomous    fins      legs      tail
##  TRUE :80   TRUE : 8   TRUE :17   Min.   :0.00   TRUE :75
##  FALSE:21  FALSE:93  FALSE:84   1st Qu.:2.00  FALSE:26
##
##                                     Median :4.00
##                                     Mean    :2.84
##                                     3rd Qu.:4.00
##                                     Max.    :8.00
##
##      domestic  catsize      type
##  TRUE :13   TRUE :44  mammal   :41
##  FALSE:88  FALSE:57  bird     :20
##
##                                     reptile   : 5
##                                     fish       :13
##                                     amphibian  : 4
##                                     insect     : 8
##                                     mollusc.et.al:10
```

3.3 Decision Tree Classifiers

We use here the recursive partitioning implementation (`rpart`) which follows largely CART and uses the Gini index to make splitting decisions and then it uses early stopping (also called pre-pruning).

```
library(rpart)
```

3.3.1 Create Tree

We create first a tree with the default settings (see `? rpart.control`). It is very important to not use the identifier column or the algorithm will only use this column and potentially run out of memory.

```
Zoo <- Zoo |> select(-animal)
```

Alternatively, you can use `. - animal` as the formula below.

```
tree_default <- Zoo |>
  rpart(type ~ ., data = _)
tree_default
## n= 101
##
## node), split, n, loss, yval, (yprob)
##      * denotes terminal node
##
##  1) root 101 60 mammal (0.41 0.2 0.05 0.13 0.04 0.079 0.099)
##    2) milk=TRUE 41 0 mammal (1 0 0 0 0 0) *
##    3) milk=FALSE 60 40 bird (0 0.33 0.083 0.22 0.067 0.13 0.17)
##      6) feathers=TRUE 20 0 bird (0 1 0 0 0 0) *
##      7) feathers=FALSE 40 27 fish (0 0 0.12 0.33 0.1 0.2 0.25)
##        14) fins=TRUE 13 0 fish (0 0 0 1 0 0) *
##        15) fins=FALSE 27 17 mollusc.et.al (0 0 0.19 0 0.15 0.3 0.37)
##          30) backbone=TRUE 9 4 reptile (0 0 0.56 0 0.44 0 0) *
##          31) backbone=FALSE 18 8 mollusc.et.al (0 0 0 0 0 0.44 0.56) *
```

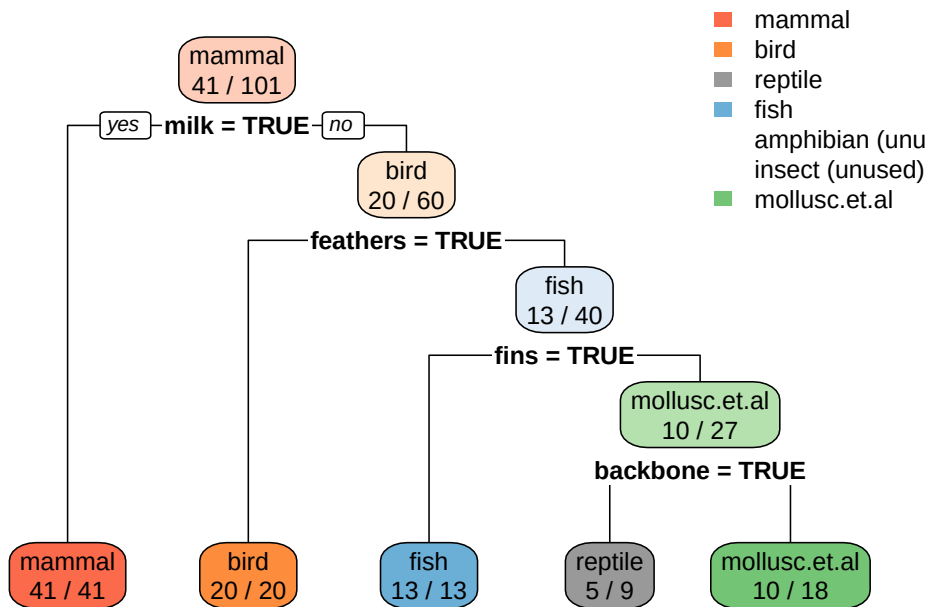
Notes:

- `|>` supplies the data for `rpart`. Since `data` is not the first argument of `rpart`, the syntax `data = _` is used to specify where the data in `Zoo` goes. The call is equivalent to `tree_default <- rpart(type ~ ., data = Zoo)`.
- The formula models the `type` variable by all other features represented by a single period (`.`).

- The class variable needs to be a factor to be recognized as nominal or rpart will create a regression tree instead of a decision tree. Use `as.factor()` on the column with the class label first, if necessary.

We can plot the resulting decision tree.

```
library(rpart.plot)
rpart.plot(tree_default, extra = 2)
```



Note: `extra=2` prints for each leaf node the number of correctly classified objects from data and the total number of objects from the training data falling into that node (correct/total).

3.3.2 Make Predictions for New Data

I will make up my own animal: A lion with feathered wings.

```
my_animal <- tibble(hair = TRUE, feathers = TRUE, eggs = FALSE,
  milk = TRUE, airborne = TRUE, aquatic = FALSE, predator = TRUE,
  toothed = TRUE, backbone = TRUE, breathes = TRUE,
  venomous = FALSE, fins = FALSE, legs = 4, tail = TRUE,
  domestic = FALSE, catsize = FALSE, type = NA)
```

The data types need to match the original data so we change the columns to be factors like in the training set.

```
my_animal <- my_animal |>
  mutate(across(where(is.logical),
    function(x) factor(x, levels = c(TRUE, FALSE))))
my_animal
## # A tibble: 1 x 17
##   hair feathers eggs milk airborne aquatic predator
##   <fct> <fct>   <fct> <fct> <fct>   <fct>   <fct>
## 1 TRUE  TRUE    FALSE TRUE  TRUE    FALSE   TRUE
## # i 10 more variables: toothed <fct>, backbone <fct>,
## #   breathes <fct>, venomous <fct>, fins <fct>, legs <dbl>,
## #   tail <fct>, domestic <fct>, catsize <fct>, type <fct>
```

Next, we make a prediction using the default tree

```
predict(tree_default , my_animal, type = "class")
##      1
## mammal
## 7 Levels: mammal bird reptile fish amphibian ... mollusc.et.al
```

3.3.3 Calculation of the Resubstitution Error

We will calculate error of the model on the training data manually first, so we see how it is calculated. The over all error (i.e., number of incorrectly classified examples) can be broken down into the error for each class. This information is typically presented in the form of a *confusion matrix*.

```
predict(tree_default, Zoo) |> head ()
##   mammal bird reptile fish amphibian insect mollusc.et.al
## 1      1    0      0    0          0      0              0
## 2      1    0      0    0          0      0              0
## 3      0    0      0    1          0      0              0
## 4      1    0      0    0          0      0              0
## 5      1    0      0    0          0      0              0
## 6      1    0      0    0          0      0              0
pred <- predict(tree_default, Zoo, type="class")
head(pred)
##      1      2      3      4      5      6
## mammal mammal  fish mammal mammal mammal
## 7 Levels: mammal bird reptile fish amphibian ... mollusc.et.al
```

We can easily tabulate the true and predicted labels to create a confusion matrix.

```

confusion_table <- with(Zoo, table(type, pred))
confusion_table
##               pred
## type      mammal bird reptile fish amphibian insect
## mammal         41   0      0    0         0      0
## bird           0  20      0    0         0      0
## reptile        0   0      5    0         0      0
## fish           0   0      0   13         0      0
## amphibian      0   0      4    0         0      0
## insect         0   0      0    0         0      0
## mollusc.et.al  0   0      0    0         0      0
##               pred
## type      mollusc.et.al
## mammal                0
## bird                  0
## reptile                0
## fish                   0
## amphibian              0
## insect                 8
## mollusc.et.al         10

```

The counts in the diagonal are correct predictions. Off-diagonal counts represent errors (i.e., confusions).

We can summarize the confusion matrix using the accuracy measure.

```

correct <- confusion_table |> diag() |> sum()
correct
## [1] 89
error <- confusion_table |> sum() - correct
error
## [1] 12

```

Accuracy is just $1 - \text{error rate}$ and give the proportion of correctly classified examples.

```

accuracy <- correct / (correct + error)
accuracy
## [1] 0.8812

```

Here is the accuracy calculation as a simple function.

```

accuracy <- function(prediction, truth) {
  tbl <- table(truth, prediction)
  sum(diag(tbl))/sum(tbl)
}

```

```

}

accuracy(pred, Zoo |> pull(type))
## [1] 0.8812

```

The caret package provides a convenient way to calculate and analyze classification errors with the confusion matrix. It only needs the predicted class labels and the correct class labels as the reference.

```

library(caret)
confusionMatrix(data = pred,
                 reference = Zoo |> pull(type))
## Confusion Matrix and Statistics
##
##              Reference
## Prediction  mammal bird reptile fish amphibian insect
## mammal      41    0      0    0          0      0
## bird         0   20      0    0          0      0
## reptile       0    0      5    0          4      0
## fish         0    0      0   13          0      0
## amphibian     0    0      0    0          0      0
## insect       0    0      0    0          0      0
## mollusc.et.al 0    0      0    0          0      8
##
##              Reference
## Prediction  mollusc.et.al
## mammal              0
## bird                0
## reptile              0
## fish                0
## amphibian            0
## insect              0
## mollusc.et.al       10
##
## Overall Statistics
##
##              Accuracy : 0.881
##              95% CI : (0.802, 0.937)
##      No Information Rate : 0.406
##      P-Value [Acc > NIR] : <2e-16
##
##              Kappa : 0.843
##
##  McNemar's Test P-Value : NA
##
## Statistics by Class:

```



```
##
##                               Class: mammal Class: bird
## Sensitivity                    1.000        1.000
## Specificity                    1.000        1.000
## Pos Pred Value                 1.000        1.000
## Neg Pred Value                 1.000        1.000
## Prevalence                     0.406        0.198
## Detection Rate                 0.406        0.198
## Detection Prevalence          0.406        0.198
## Balanced Accuracy              1.000        1.000
##                               Class: reptile Class: fish
## Sensitivity                    1.0000       1.000
## Specificity                    0.9583       1.000
## Pos Pred Value                 0.5556       1.000
## Neg Pred Value                 1.0000       1.000
## Prevalence                     0.0495       0.129
## Detection Rate                 0.0495       0.129
## Detection Prevalence          0.0891       0.129
## Balanced Accuracy              0.9792       1.000
##                               Class: amphibian Class: insect
## Sensitivity                    0.0000       0.0000
## Specificity                    1.0000       1.0000
## Pos Pred Value                 NaN         NaN
## Neg Pred Value                 0.9604       0.9208
## Prevalence                     0.0396       0.0792
## Detection Rate                 0.0000       0.0000
## Detection Prevalence          0.0000       0.0000
## Balanced Accuracy              0.5000       0.5000
##                               Class: mollusc.et.al
## Sensitivity                    1.000
## Specificity                    0.912
## Pos Pred Value                 0.556
## Neg Pred Value                 1.000
## Prevalence                     0.099
## Detection Rate                 0.099
## Detection Prevalence          0.178
## Balanced Accuracy              0.956
```

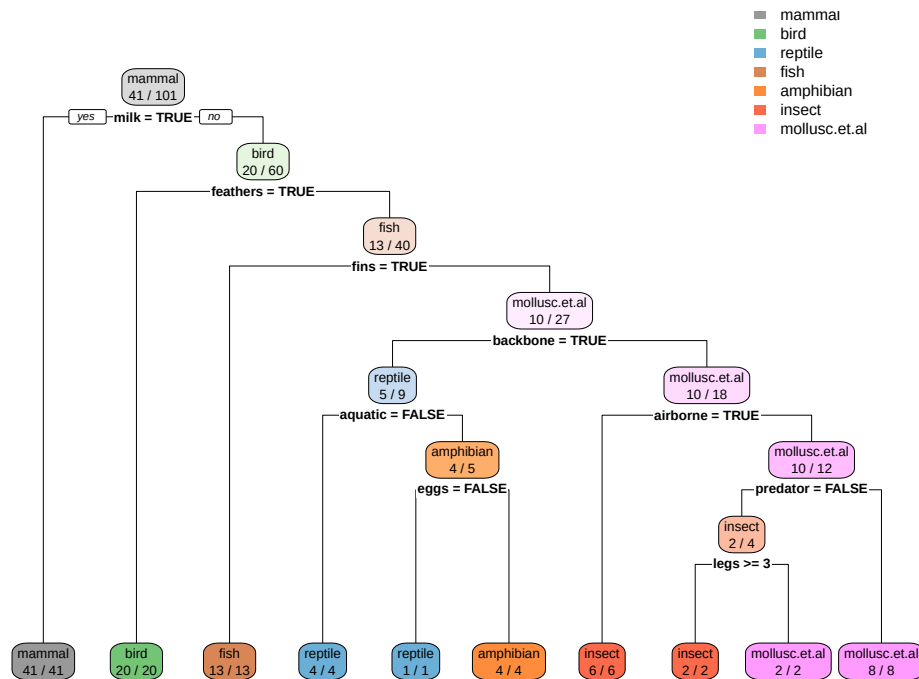
In addition to the confusion matrix, it also includes additional statistics and measures. Details can be found in the Model Evaluation section.

Important note: Calculating accuracy on the training data is not a good idea! A complete example with code for holding out a test set and performing hyperparameter selection using cross-validation can be found in section Hyperparameter Tuning.

3.4 Model Overfitting

We are tempted to create the largest possible tree to get the most accurate model. This can be achieved by changing the algorithm's hyperparameter (parameters that change how the algorithm works). We set the complexity parameter `cp` to 0 (split even if it does not improve the fit) and we set the minimum number of observations in a node needed to split to the smallest value of 2 (see: `?rpart.control`). *Note:* This is not a good idea! As we will see later, full trees overfit the training data!

```
tree_full <- Zoo |>
  rpart(type ~ ., data = _,
        control = rpart.control(minsplit = 2, cp = 0))
rpart.plot(tree_full, extra = 2,
           roundint=FALSE,
           box.palette = list("Gy", "Gn", "Bu", "Bn",
                             "Or", "Rd", "Pu"))
```



```
tree_full
## n= 101
##
## node), split, n, loss, yval, (yprob)
##      * denotes terminal node
```

```
##
## 1) root 101 60 mammal (0.41 0.2 0.05 0.13 0.04 0.079 0.099)
## 2) milk=TRUE 41 0 mammal (1 0 0 0 0 0 0) *
## 3) milk=FALSE 60 40 bird (0 0.33 0.083 0.22 0.067 0.13 0.17)
## 6) feathers=TRUE 20 0 bird (0 1 0 0 0 0 0) *
## 7) feathers=FALSE 40 27 fish (0 0 0.12 0.33 0.1 0.2 0.25)
## 14) fins=TRUE 13 0 fish (0 0 0 1 0 0 0) *
## 15) fins=FALSE 27 17 mollusc.et.al (0 0 0.19 0 0.15 0.3 0.37)
## 30) backbone=TRUE 9 4 reptile (0 0 0.56 0 0.44 0 0)
## 60) aquatic=FALSE 4 0 reptile (0 0 1 0 0 0 0) *
## 61) aquatic=TRUE 5 1 amphibian (0 0 0.2 0 0.8 0 0)
## 122) eggs=FALSE 1 0 reptile (0 0 1 0 0 0 0) *
## 123) eggs=TRUE 4 0 amphibian (0 0 0 0 1 0 0) *
## 31) backbone=FALSE 18 8 mollusc.et.al (0 0 0 0 0 0.44 0.56)
## 62) airborne=TRUE 6 0 insect (0 0 0 0 0 1 0) *
## 63) airborne=FALSE 12 2 mollusc.et.al (0 0 0 0 0 0.17 0.83)
## 126) predator=FALSE 4 2 insect (0 0 0 0 0 0.5 0.5)
## 252) legs>=3 2 0 insect (0 0 0 0 0 1 0) *
## 253) legs< 3 2 0 mollusc.et.al (0 0 0 0 0 0 1) *
## 127) predator=TRUE 8 0 mollusc.et.al (0 0 0 0 0 0 1) *
```

Error on the training set of the full tree

```
pred_full <- predict(tree_full, Zoo, type = "class")

accuracy(pred_full, Zoo |> pull(type))
## [1] 1
```

We see that the error is smaller then for the pruned tree. This, however, does not mean that the model is better. It actually is overfitting the training data (it just memorizes it) and it likely has worse generalization performance on new data. This effect is called overfitting the training data and needs to be avoided.

3.5 Model Selection

We often can create many different models for a classification problem. Above, we have created a decision tree using the default settings and also a full tree. The question is: Which one should we use. This problem is called model selection.

In order to select the model we need to split the training data into a **validation set** and the training set that is actually used to train model. The error rate on the validation set can then be used to choose between several models.

Caret has model selection build into the `train()` function. We will compare two trees one with the default complexity `cp = 0.01` and a full tree `cp = 0`.

The values are set via `tuneGrid`. `trControl` specified how the validation set is obtained. We use Leave Group Out Cross-Validation (LGOCV) which picks randomly a proportion `p` of data to train and uses the rest as the validation set. To get a better estimate of the error, this process is repeated `number` of times and the errors are averaged.

```
fit <- Zoo |>
  train(type ~ .,
        data = _ ,
        method = "rpart",
        control = rpart.control(minsplit = 2), # we have little data
        tuneGrid = data.frame(cp = c(0.01, 0)),
        trControl = trainControl(method = "LGOCV",
                                  p = 0.8,
                                  number = 10),

        tuneLength = 5)

fit
## CART
##
## 101 samples
## 16 predictor
## 7 classes: 'mammal', 'bird', 'reptile', 'fish', 'amphibian', 'insect', 'mollusc.e
##
## No pre-processing
## Resampling: Repeated Train/Test Splits Estimated (10 reps, 80%)
## Summary of sample sizes: 83, 83, 83, 83, 83, 83, ...
## Resampling results across tuning parameters:
##
##   cp    Accuracy  Kappa
##  0.00  0.9722    0.9616
##  0.01  0.9667    0.9538
##
## Accuracy was used to select the optimal model using
## the largest value.
## The final value used for the model was cp = 0.
```

We see that in this case, the full tree model performs slightly better. However, given the small dataset of 101 animals and the tiny validation set (20% of the animals), this may not be a significant difference and we will look at a statistical test for this later.

3.6 Model Evaluation

Models should be evaluated on a test set that has no overlap with the training set. We typically split the data using random sampling. To get reproducible results, we set random number generator seed.

```
set.seed(2000)
```

3.6.1 Holdout Method

Test data is not used in the model building process and set aside purely for testing the model. Here, we partition data the 80% training and 20% testing.

```
inTrain <- createDataPartition(y = Zoo$type, p = .8)[[1]]
Zoo_train <- Zoo |> slice(inTrain)
Zoo_test <- Zoo |> slice(-inTrain)
```

Now we can train on the `Zoo_train` set and get the generalization error on the `Zoo_test` set.

3.6.2 Cross-Validation Methods

There are several cross-validation methods that can use the available data more efficiently than the holdout method. The most popular method is k -fold cross-validation which splits the data randomly into k folds. It then holds one fold back for testing and trains on the other $k - 1$ folds. This is done with each fold and the resulting statistic (e.g., accuracy) is averaged. This method uses the data more efficiently than the holdout method.

Cross validation can be directly used in `train()` using `trControl = trainControl(method = "cv", number = 10)`. If no model selection is necessary then this will give the generalization error.

Cross-validation runs are independent and can be done faster in parallel. To enable multi-core support, `caret` uses the package `foreach` and you need to load a `do` backend. For Linux, you can use `doMC` with 4 cores. Windows needs different backend like `doParallel` (see `caret` cheat sheet above).

```
## Linux backend
# library(doMC)
# registerDoMC(cores = 4)
# getDoParWorkers()

## Windows backend
```

```
# library(doParallel)
# cl <- makeCluster(4, type="SOCK")
# registerDoParallel(cl)
```

3.7 Hyperparameter Tuning

Note: This section contains a complete code example of how data should be used. It first holds out a test set and then performing hyperparameter selection using cross-validation.

Hyperparameters are parameters that change how a training algorithm works. An example is the complexity parameter `cp` for `rpart` decision trees. Tuning the hyperparameter means that we want to perform model selection to pick the best setting.

We typically first use the holdout method to create a test set and then use cross validation using the training data for model selection. Let us use 80% for training and hold out 20% for testing.

```
inTrain <- createDataPartition(y = Zoo$type, p = .8)[[1]]
Zoo_train <- Zoo |> slice(inTrain)
Zoo_test <- Zoo |> slice(-inTrain)
```

The package `caret` combines training and validation for hyperparameter tuning into the `train()` function. It internally splits the data into training and validation sets and thus will provide you with error estimates for different hyperparameter settings. `trainControl` is used to choose how testing is performed.

For `rpart`, `train` tries to tune the `cp` parameter (tree complexity) using accuracy to choose the best model. I set `minsplit` to 2 since we have not much data.

Note: Parameters used for tuning (in this case `cp`) need to be set using a `data.frame` in the argument `tuneGrid`! Setting it in `control` will be ignored.

```
fit <- Zoo_train |>
  train(type ~ .,
        data = _ ,
        method = "rpart",
        control = rpart.control(minsplit = 2), # we have little data
        trControl = trainControl(method = "cv", number = 10),
        tuneLength = 5)

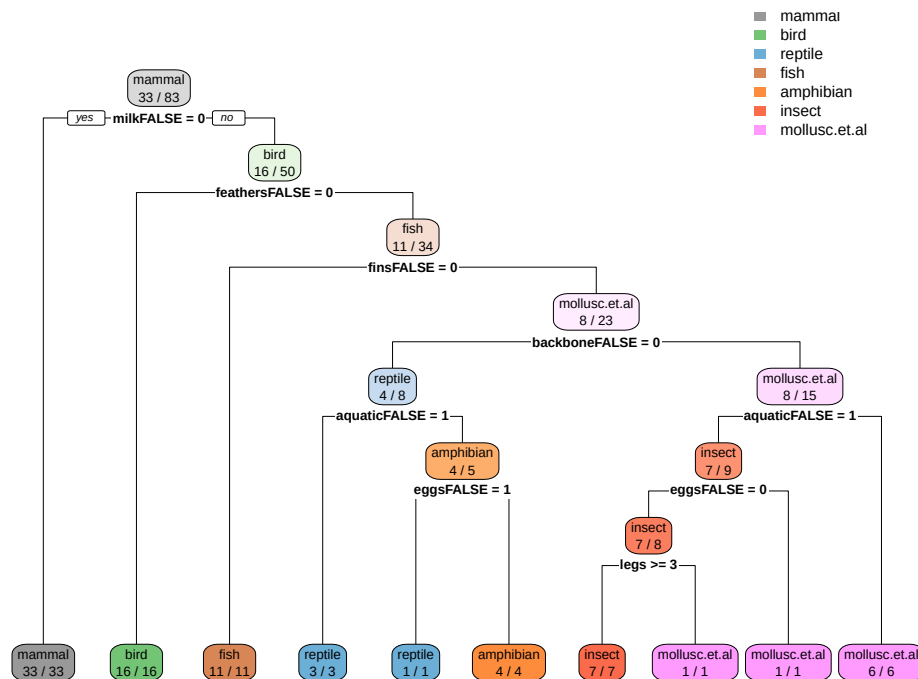
fit
## CART
##
```

```
## 83 samples
## 16 predictors
## 7 classes: 'mammal', 'bird', 'reptile', 'fish', 'amphibian', 'insect', 'mollusc.et.al'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 74, 73, 77, 75, 73, 75, ...
## Resampling results across tuning parameters:
##
##   cp    Accuracy  Kappa
##   0.00  0.9289    0.9058
##   0.08  0.8603    0.8179
##   0.16  0.7296    0.6422
##   0.22  0.6644    0.5448
##   0.32  0.4383    0.1136
##
## Accuracy was used to select the optimal model using
## the largest value.
## The final value used for the model was cp = 0.
```

Note: Train has built 10 trees using the training folds for each value of `cp` and the reported values for accuracy and Kappa are the averages on the validation folds.

A model using the best tuning parameters and using all the data supplied to `train()` is available as `fit$finalModel`.

```
library(rpart.plot)
rpart.plot(fit$finalModel, extra = 2,
  box.palette = list("Gy", "Gn", "Bu", "Bn", "Or", "Rd", "Pu"))
```



caret also computes variable importance. By default it uses competing splits (splits which would be runners up, but do not get chosen by the tree) for rpart models (see ? varImp). Toothed is the runner up for many splits, but it never gets chosen!

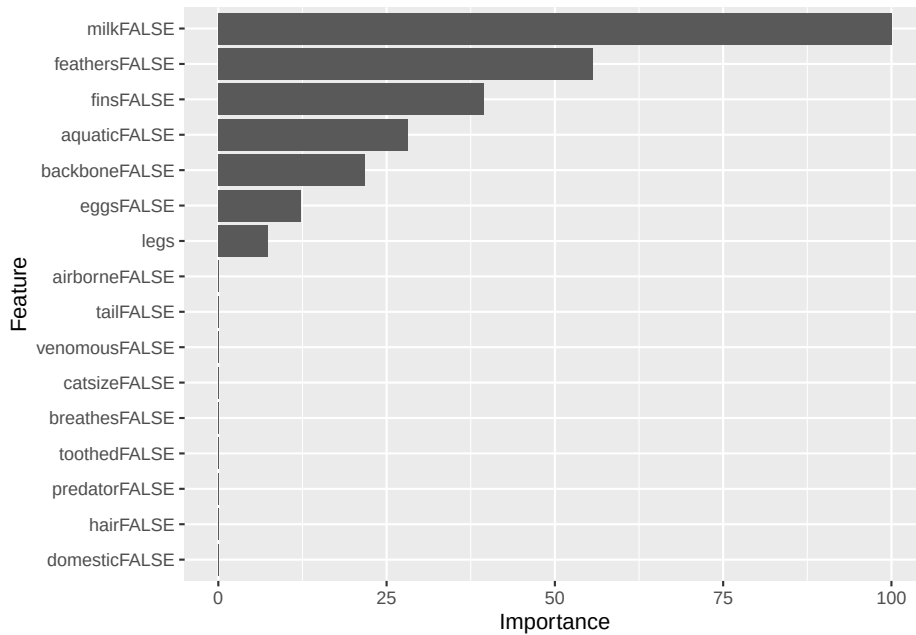
```
varImp(fit)
## rpart variable importance
##
##           Overall
## toothedFALSE 100.0
## feathersFALSE  79.5
## eggsFALSE     67.7
## milkFALSE     63.3
## backboneFALSE 57.3
## finsFALSE     53.5
## hairFALSE     52.1
## breathesFALSE 48.9
## legs          41.4
## tailFALSE     29.0
## aquaticFALSE  27.5
## airborneFALSE 26.5
## predatorFALSE 10.6
## venomousFALSE  1.8
## catsizeFALSE  0.0
```



```
## domesticFALSE      0.0
```

Here is the variable importance without competing splits.

```
imp <- varImp(fit, compete = FALSE)
imp
## rpart variable importance
##
##           Overall
## milkFALSE      100.00
## feathersFALSE   55.69
## finsFALSE       39.45
## aquaticFALSE    28.11
## backboneFALSE   21.76
## eggsFALSE       12.32
## legs            7.28
## tailFALSE        0.00
## domesticFALSE    0.00
## airborneFALSE    0.00
## catsizeFALSE     0.00
## toothedFALSE     0.00
## venomousFALSE    0.00
## hairFALSE        0.00
## breathesFALSE    0.00
## predatorFALSE    0.00
ggplot(imp)
```



Note: Not all models provide a variable importance function. In this case caret might calculate the variable importance by itself and ignore the model (see ? varImp)!

Now, we can estimate the generalization error of the best model on the held out test data.

```
pred <- predict(fit, newdata = Zoo_test)
pred
## [1] mammal      bird      mollusc.et.al bird
## [5] mammal      mammal      insect      bird
## [9] mammal      mammal      mammal      mammal
## [13] bird        fish        fish        reptile
## [17] mammal      mollusc.et.al
## 7 Levels: mammal bird reptile fish amphibian ... mollusc.et.al
```

Caret's `confusionMatrix()` function calculates accuracy, confidence intervals, kappa and many more evaluation metrics. You need to use separate test data to create a confusion matrix based on the generalization error.

```
confusionMatrix(data = pred,
                 ref = Zoo_test |> pull(type))
## Confusion Matrix and Statistics
##
##              Reference
```

```

## Prediction      mammal bird reptile fish amphibian insect
## mammal          8    0      0    0          0    0
## bird            0    4      0    0          0    0
## reptile         0    0      1    0          0    0
## fish            0    0      0    2          0    0
## amphibian       0    0      0    0          0    0
## insect          0    0      0    0          0    1
## mollusc.et.al   0    0      0    0          0    0
##
##               Reference
## Prediction      mollusc.et.al
## mammal          0
## bird            0
## reptile         0
## fish            0
## amphibian       0
## insect          0
## mollusc.et.al   2
##
## Overall Statistics
##
##               Accuracy : 1
##               95% CI : (0.815, 1)
##       No Information Rate : 0.444
##       P-Value [Acc > NIR] : 4.58e-07
##
##               Kappa : 1
##
## Mcnemar's Test P-Value : NA
##
## Statistics by Class:
##
##               Class: mammal Class: bird
## Sensitivity          1.000      1.000
## Specificity          1.000      1.000
## Pos Pred Value       1.000      1.000
## Neg Pred Value       1.000      1.000
## Prevalence           0.444      0.222
## Detection Rate       0.444      0.222
## Detection Prevalence 0.444      0.222
## Balanced Accuracy    1.000      1.000
##
##               Class: reptile Class: fish
## Sensitivity          1.0000     1.000
## Specificity          1.0000     1.000
## Pos Pred Value       1.0000     1.000
## Neg Pred Value       1.0000     1.000

```

```
## Prevalence          0.0556      0.111
## Detection Rate      0.0556      0.111
## Detection Prevalence 0.0556      0.111
## Balanced Accuracy    1.0000      1.000
##                      Class: amphibian Class: insect
## Sensitivity          NA          1.0000
## Specificity          1          1.0000
## Pos Pred Value       NA          1.0000
## Neg Pred Value       NA          1.0000
## Prevalence          0          0.0556
## Detection Rate      0          0.0556
## Detection Prevalence 0          0.0556
## Balanced Accuracy    NA          1.0000
##                      Class: mollusc.et.al
## Sensitivity          1.000
## Specificity          1.000
## Pos Pred Value       1.000
## Neg Pred Value       1.000
## Prevalence          0.111
## Detection Rate      0.111
## Detection Prevalence 0.111
## Balanced Accuracy    1.000
```

Definitions of the additional statistics by class (including alternative names) can be found in caret’s confusion matrix man page⁹.

Some notes

- Many classification algorithms and `train` in caret do not deal well with missing values. If your classification model can deal with missing values (e.g., `rpart`) then use `na.action = na.pass` when you call `train` and `predict`. Otherwise, you need to remove observations with missing values with `na.omit` or use imputation to replace the missing values before you train the model. Make sure that you still have enough observations left.
- Make sure that nominal variables (this includes logical variables) are coded as factors.
- The class variable for train in caret cannot have level names that are keywords in R (e.g., `TRUE` and `FALSE`). Rename them to, for example, “yes” and “no.”
- Make sure that nominal variables (factors) have examples for all possible values. Some methods might have problems with variable values without examples. You can drop empty levels using `droplevels` or `factor`.
- Sampling in train might create a sample that does not contain examples for all values in a nominal (factor) variable. You will get an error message.

⁹<https://rdrr.io/cran/caret/man/confusionMatrix.html>

This most likely happens for variables which have one very rare value. You may have to remove the variable.

3.8 Pitfalls of Model Selection and Evaluation

- Do not measure the error on the training set or use the validation error as a generalization error estimate. Always use the generalization error on a test set!
- The training data and the test sets cannot overlap or we will not evaluate the generalization performance. The training set can be contaminated by things like preprocessing the all the data together.

3.9 Model Comparison

We will compare three models, a majority class baseline classifier, a decision trees with a k-nearest neighbors (kNN) classifier. We will use 10-fold cross-validation for hyper parameter tuning. Caret's `train()` function refits the selected model on all of the training data and performs cross-validation to estimate the generalization error. These cross-validation results can be used to compare models statistically.

3.9.1 Build models

Caret does not provide a baseline classifier, but the package `basemodels` does. We first create a weak baseline model that always predicts the the majority class mammal.

```
baseline <- Zoo_train |> train(type ~ .,
  method = basemodels::dummyClassifier,
  data = _,
  strategy = "constant",
  constant = "mammal",
  trControl = trainControl(method = "cv"
    ))

baseline
## dummyClassifier
##
## 83 samples
## 16 predictors
## 7 classes: 'mammal', 'bird', 'reptile', 'fish', 'amphibian', 'insect', 'mollusc.et.al'
##
```

```
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 73, 74, 73, 75, 74, 77, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.4047     0
```

The second model is a default decision tree.

```
rpartFit <- Zoo_train |>
  train(type ~ .,
        data = _,
        method = "rpart",
        tuneLength = 10,
        trControl = trainControl(method = "cv")
  )
rpartFit
## CART
##
## 83 samples
## 16 predictors
## 7 classes: 'mammal', 'bird', 'reptile', 'fish', 'amphibian', 'insect', 'mollusc.etc'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 75, 75, 75, 73, 76, 72, ...
## Resampling results across tuning parameters:
##
##   cp      Accuracy   Kappa
##   0.00000 0.7841     0.7195
##   0.03556 0.7841     0.7195
##   0.07111 0.7841     0.7195
##   0.10667 0.7841     0.7182
##   0.14222 0.7841     0.7182
##   0.17778 0.7271     0.6369
##   0.21333 0.7071     0.6109
##   0.24889 0.5940     0.4423
##   0.28444 0.5940     0.4423
##   0.32000 0.4968     0.2356
##
## Accuracy was used to select the optimal model using
## the largest value.
## The final value used for the model was cp = 0.1422.
```

The third model is a kNN classifier, this classifier will be discussed in the next Chapter. kNN uses the Euclidean distance between objects. Logicals will be used as 0-1 variables. To make sure the range of all variables is compatible, we ask `train` to scale the data using `preProcess = "scale"`.

```
knnFit <- Zoo_train |>
  train(type ~ .,
        data = _,
        method = "knn",
        preProcess = "scale",
        tuneLength = 10,
        trControl = trainControl(method = "cv")
  )
knnFit
## k-Nearest Neighbors
##
## 83 samples
## 16 predictors
## 7 classes: 'mammal', 'bird', 'reptile', 'fish', 'amphibian', 'insect', 'mollusc.et.al'
##
## Pre-processing: scaled (16)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 75, 74, 74, 74, 74, 75, ...
## Resampling results across tuning parameters:
##
##   k   Accuracy   Kappa
##   5  0.9092     0.8835
##   7  0.8715     0.8301
##   9  0.8579     0.8113
##  11  0.8590     0.8131
##  13  0.8727     0.8302
##  15  0.8727     0.8302
##  17  0.8490     0.7989
##  19  0.8490     0.7967
##  21  0.7943     0.7219
##  23  0.7943     0.7217
##
## Accuracy was used to select the optimal model using
## the largest value.
## The final value used for the model was k = 5.
```

Compare the accuracy and kappa distributions of the final model over all folds.

```
resamps <- resamples(list(
  baseline = baseline,
```

```

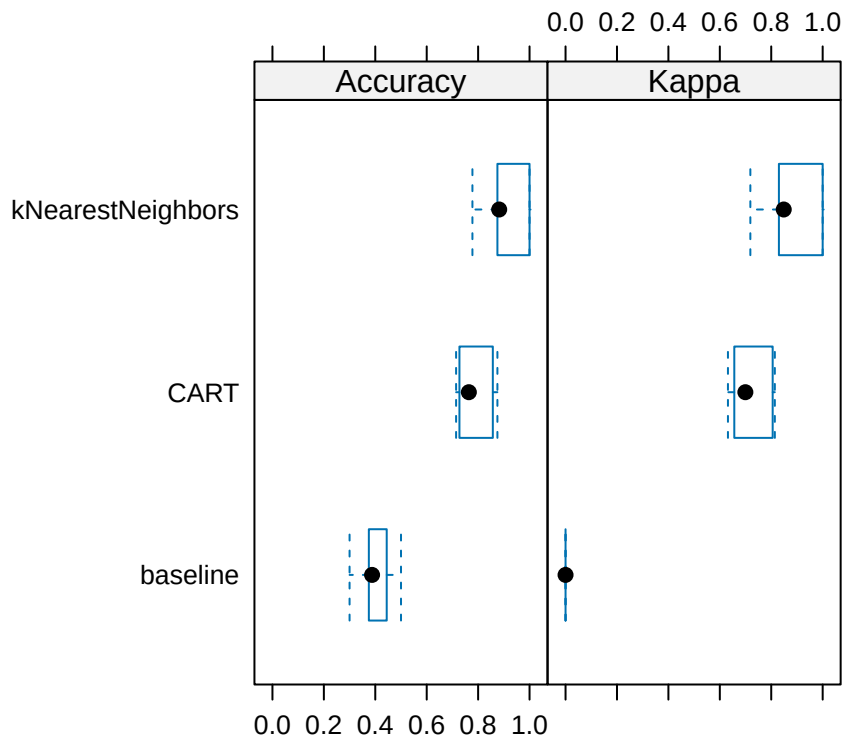
CART = rpartFit,
      kNearestNeighbors = knnFit
))

summary(resamps)
##
## Call:
## summary.resamples(object = resamps)
##
## Models: baseline, CART, kNearestNeighbors
## Number of resamples: 10
##
## Accuracy
##           Min. 1st Qu. Median  Mean 3rd Qu.
## baseline      0.3000  0.375 0.3875 0.4047  0.4444
## CART          0.7143  0.733 0.7639 0.7841  0.8429
## kNearestNeighbors 0.7778  0.875 0.8819 0.9092  1.0000
##           Max. NA's
## baseline      0.500    0
## CART          0.875    0
## kNearestNeighbors 1.000    0
##
## Kappa
##           Min. 1st Qu. Median  Mean 3rd Qu.
## baseline      0.0000  0.0000 0.0000 0.0000  0.0000
## CART          0.6316  0.6622 0.6994 0.7182  0.7917
## kNearestNeighbors 0.7188  0.8307 0.8486 0.8835  1.0000
##           Max. NA's
## baseline      0.000    0
## CART          0.814    0
## kNearestNeighbors 1.000    0

```

`caret` provides some visualizations. For example, a boxplot to compare the accuracy and kappa distribution (over the 10 folds).

```
bwplot(resamps, layout = c(3, 1))
```

We see that the baseline has no predictive power and produces consistently a kappa of 0. KNN performs consistently the best. To find out if one models is statistically better than the other, we can use a statistical test.

```
difs <- diff(resamps)
difs
##
## Call:
## diff.resamples(x = resamps)
##
## Models: baseline, CART, kNearestNeighbors
## Metrics: Accuracy, Kappa
## Number of differences: 3
## p-value adjustment: bonferroni
summary(difs)
##
## Call:
## summary.diff.resamples(object = difs)
##
## p-value adjustment: bonferroni
## Upper diagonal: estimates of the difference
## Lower diagonal: p-value for H0: difference = 0
```

```
##
## Accuracy
##           baseline CART   kNearestNeighbors
## baseline           -0.379 -0.504
## CART              5.19e-06   -0.125
## kNearestNeighbors 4.03e-08 0.031
##
## Kappa
##           baseline CART   kNearestNeighbors
## baseline           -0.718 -0.884
## CART              5.79e-10   -0.165
## kNearestNeighbors 2.87e-09 0.0206
```

p-values gives us the probability of seeing an even more extreme value (difference between accuracy or kappa) given that the null hypothesis (difference = 0) is true. For a better classifier, the p-value should be “significant,” i.e., less than .05 or 0.01. `diff` automatically applies Bonferroni correction for multiple comparisons to adjust the p-value upwards. In this case, CART and kNN perform significantly better than the baseline classifiers. The difference between CART and kNN is only significant at the 0.05 level, so kNN might be slightly better.

3.10 Feature Selection*

Decision trees implicitly select features for splitting, but we can also select features before we apply any learning algorithm. Since different features lead to different models, choosing the best set of features is also a type of model selection.

Many feature selection methods are implemented in the `FSelector` package.

```
library(FSelector)
```

3.10.1 Univariate Feature Importance Score

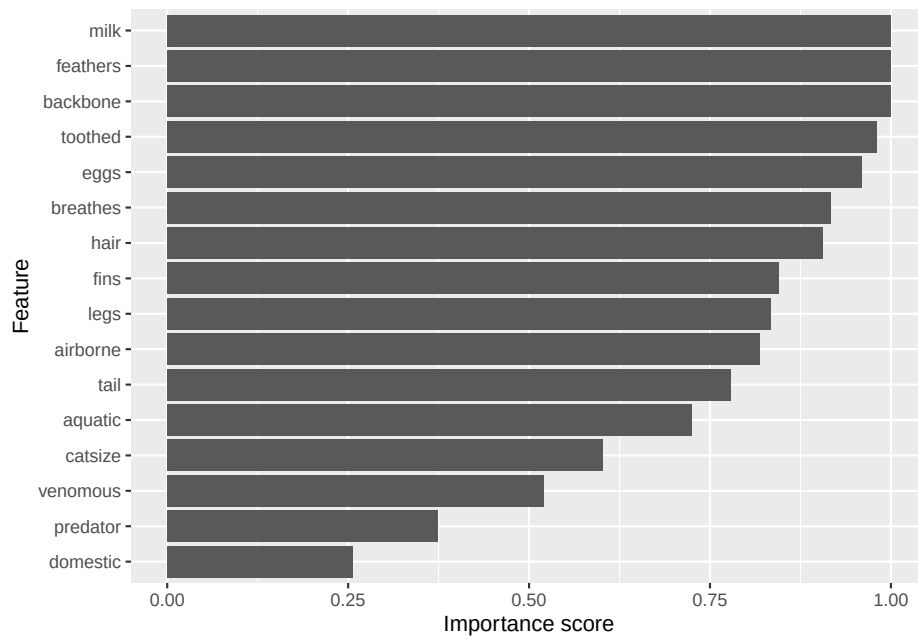
These scores measure how related each feature is to the class variable. For discrete features (as in our case), the chi-square statistic can be used to derive a score.

```
weights <- Zoo_train |>
  chi.squared(type ~ ., data = _) |>
  as_tibble(rownames = "feature") |>
  arrange(desc(attr_importance))
```

```
weights
## # A tibble: 16 x 2
##   feature attr_importance
##   <chr>      <dbl>
## 1 feathers      1
## 2 milk          1
## 3 backbone      1
## 4 toothed    0.981
## 5 eggs        0.959
## 6 breathes    0.917
## 7 hair        0.906
## 8 fins        0.845
## 9 legs        0.834
## 10 airborne   0.818
## 11 tail       0.779
## 12 aquatic    0.725
## 13 catsize    0.602
## 14 venomous   0.520
## 15 predator   0.374
## 16 domestic   0.256
```

We can plot the importance in descending order (using `reorder` to order factor levels used by `ggplot`).

```
ggplot(weights,
  aes(x = attr_importance,
      y = reorder(feature, attr_importance))) +
  geom_bar(stat = "identity") +
  xlab("Importance score") +
  ylab("Feature")
```

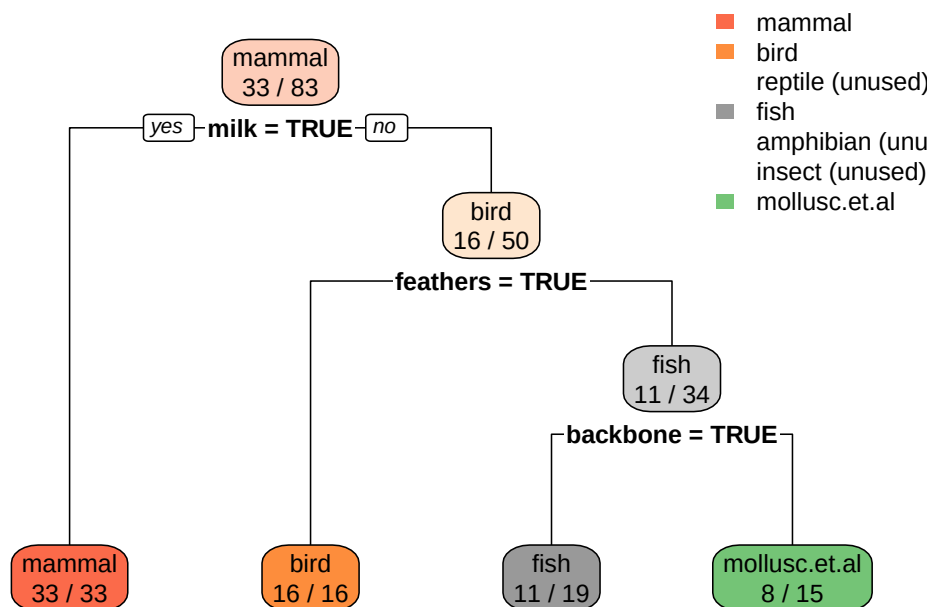


Picking the best features is called the feature ranking approach. Here we pick the 5 highest-ranked features.

```
subset <- cutoff.k(weights |>
  column_to_rownames("feature"),
  5)
subset
## [1] "feathers" "milk" "backbone" "toothed" "eggs"
```

Use only the selected features to build a model (`Fselector` provides `as.simple.formula`).

```
f <- as.simple.formula(subset, "type")
f
## type ~ feathers + milk + backbone + toothed + eggs
## <environment: 0x5b6daf0251b0>
m <- Zoo_train |> rpart(f, data = _)
rpart.plot(m, extra = 2, roundint = FALSE)
```



There are many alternative ways to calculate univariate importance scores (see package `FSelector`). Some of them (also) work for continuous features. One example is the information gain ratio based on entropy as used in decision tree induction.

```

Zoo_train |>
  gain.ratio(type ~ ., data = _) |>
  as_tibble(rownames = "feature") |>
  arrange(desc(attr_importance))
## # A tibble: 16 x 2
##   feature attr_importance
##   <chr>         <dbl>
## 1 milk           1
## 2 backbone       1
## 3 feathers       1
## 4 toothed       0.959
## 5 eggs          0.907
## 6 breathes      0.845
## 7 hair          0.781
## 8 fins          0.689
## 9 legs          0.689
## 10 airborne     0.633
## 11 tail         0.573
## 12 aquatic      0.474
## 13 venomous     0.429
## 14 catsize      0.310
  
```

```
## 15 domestic      0.115
## 16 predator      0.110
```

3.10.2 Feature Subset Selection

Often, features are related and calculating importance for each feature independently is not optimal. We can use greedy search heuristics. For example `cfs` uses correlation/entropy with best first search.

```
Zoo_train |>
  cfs(type ~ ., data = _)
## [1] "hair"      "feathers" "eggs"      "milk"      "toothed"
## [6] "backbone" "breathes" "fins"      "legs"      "tail"
```

The disadvantage of this method is that the model we want to train may not use correlation/entropy. We can use the actual model using as a black-box defined in an evaluator function to calculate a score to be maximized. This is typically the best method, since it can use the model for selection. First, we define an evaluation function that builds a model given a subset of features and calculates a quality score. We use here the average for 5 bootstrap samples (`method = "cv"` can also be used instead), no tuning (to be faster), and the average accuracy as the score.

```
evaluator <- function(subset) {
  model <- Zoo_train |>
    train(as.simple.formula(subset, "type"),
          data = _,
          method = "rpart",
          trControl = trainControl(method = "boot", number = 5),
          tuneLength = 0)

  results <- model$resample$Accuracy

  cat("Trying features:", paste(subset, collapse = " + "), "\n")

  m <- mean(results)
  cat("Accuracy:", round(m, 2), "\n\n")
  m
}
```

Start with all features (but not the class variable `type`)

```
features <- Zoo_train |>
  colnames() |>
  setdiff("type")
```

There are several (greedy) search strategies available. These run for a while so they commented out below. Remove the comment for one at a time to try these types of feature selection.

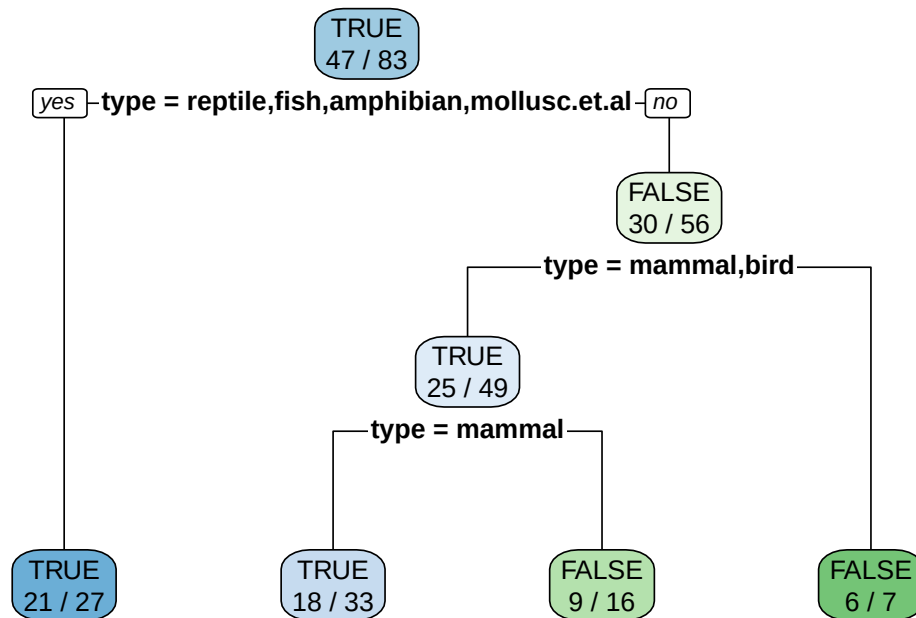
```
#subset <- backward.search(features, evaluator)
#subset <- forward.search(features, evaluator)
#subset <- best.first.search(features, evaluator)
#subset <- hill.climbing.search(features, evaluator)
#subset
```

3.11 Using Dummy Variables for Nominal Features*

Nominal features (factors) are often encoded as a series of 0-1 dummy variables. This approach in machine learning is often called one-hot encoding.

For example, let us try to predict if an animal is a predator given the type. First we use the original encoding of type as a factor with several values.

```
tree_predator <- Zoo_train |>
  rpart(predator ~ type, data = _)
rpart.plot(tree_predator, extra = 2, roundint = FALSE)
```



Note: Some splits use multiple values. Building the tree will become extremely slow if a factor has many levels (different values) since the tree has to check all possible splits into two subsets which has a time complexity of $O(2^n)$ where n is the number of different values. This situation should be avoided.

We can convert the factor `type` into a set of 0-1 dummy variables using `caret`'s `class2ind()`. See also `? dummyVars` in package `caret`.

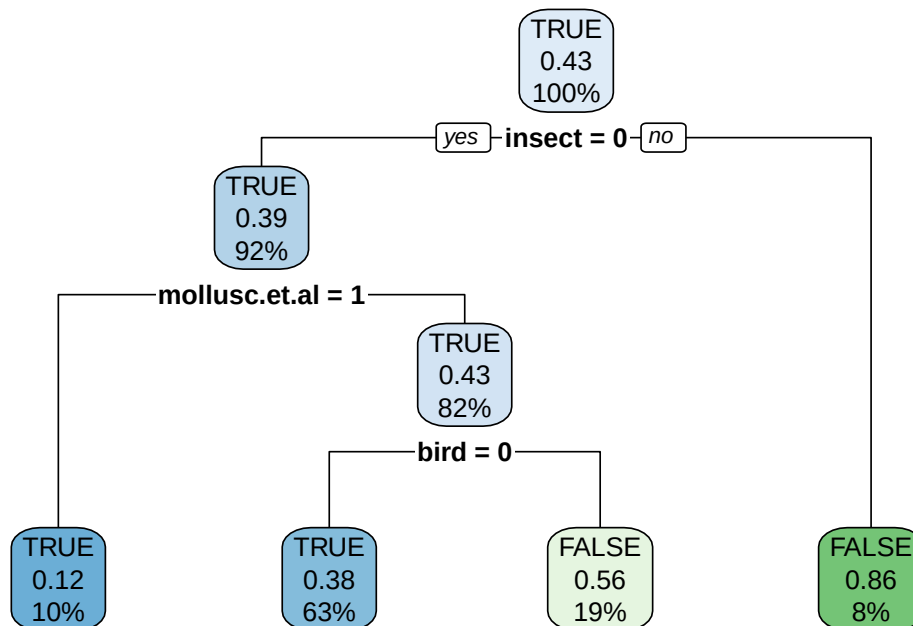
```

Zoo_train_dummy <- as_tibble(class2ind(Zoo_train$type)) |>
  mutate(across(everything(), as.factor)) |>
  add_column(predator = Zoo_train$predator)
Zoo_train_dummy
## # A tibble: 83 x 8
##   mammal bird  reptile fish  amphibian insect mollusc.et.al
##   <fct> <fct> <fct>   <fct> <fct>      <fct>   <fct>
## 1 1      0      0      0      0          0      0
## 2 0      0      0      1      0          0      0
## 3 1      0      0      0      0          0      0
## 4 1      0      0      0      0          0      0
## 5 1      0      0      0      0          0      0
## 6 1      0      0      0      0          0      0
## 7 0      0      0      1      0          0      0
## 8 0      0      0      1      0          0      0
## 9 1      0      0      0      0          0      0
## 10 1      0      0      0      0          0      0
## # i 73 more rows

```



```
## # i 1 more variable: predator <fct>
tree_predator <- Zoo_train_dummy |>
  rpart(predator ~ .,
        data = _,
        control = rpart.control(minsplit = 2, cp = 0.01))
rpart.plot(tree_predator, roundint = FALSE)
```

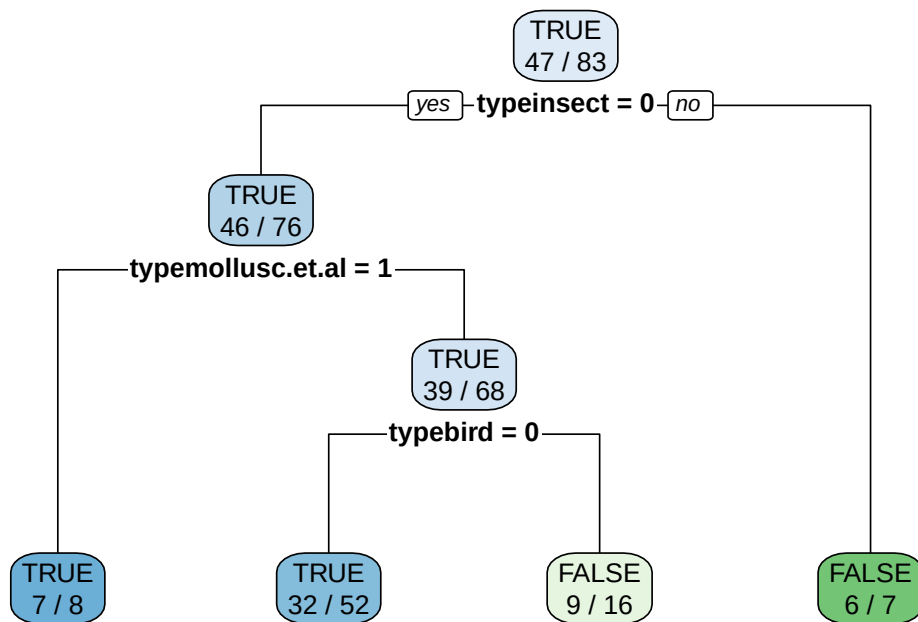


Using `caret` on the original factor encoding automatically translates factors (here type) into 0-1 dummy variables (e.g., `typeinsect = 0`). The reason is that some models cannot directly use factors and `caret` tries to consistently work with all of them.

```
fit <- Zoo_train |>
  train(predator ~ type,
        data = _,
        method = "rpart",
        control = rpart.control(minsplit = 2),
        tuneGrid = data.frame(cp = 0.01))

fit
## CART
##
## 83 samples
## 1 predictor
## 2 classes: 'TRUE', 'FALSE'
##
```

```
## No pre-processing
## Resampling: Bootstrapped (25 reps)
## Summary of sample sizes: 83, 83, 83, 83, 83, 83, ...
## Resampling results:
##
##   Accuracy   Kappa
##   0.54      0.07527
##
## Tuning parameter 'cp' was held constant at a value of 0.01
rpart.plot(fit$finalModel, extra = 2)
```



Note: To use a fixed value for the tuning parameter `cp`, we have to create a tuning grid that only contains that value.

3.12 Exercises*

We will use again the Palmer penguin data for the exercises.

```
library(palmerpenguins)
head(penguins)
## # A tibble: 6 x 8
##   species island    bill_length_mm bill_depth_mm
##   <chr>   <chr>          <dbl>         <dbl>
```

```
## 1 Adelie Torgersen      39.1      18.7
## 2 Adelie Torgersen      39.5      17.4
## 3 Adelie Torgersen      40.3       18
## 4 Adelie Torgersen      NA        NA
## 5 Adelie Torgersen      36.7      19.3
## 6 Adelie Torgersen      39.3      20.6
## # i 4 more variables: flipper_length_mm <dbl>,
## #   body_mass_g <dbl>, sex <chr>, year <dbl>
```

Create a R markdown file with the code and discussion for the following below. Remember, the complete approach is described in section Hyperparameter Tuning.

1. Split the data into a training and test set.
2. Create an rpart decision tree to predict the species. You will have to deal with missing values.
3. Experiment with setting `minsplits` for rpart and make sure `tuneLength` is at least 5. Discuss the model selection process (hyperparameter tuning) and what final model was chosen.
4. Visualize the tree and discuss what the splits mean.
5. Calculate the variable importance from the fitted model. What variables are the most important? What variables do not matter?
6. Use the test set to evaluate the generalization error and accuracy.

Chapter 4

Classification: Alternative Techniques

This chapter introduces different types of classifiers. It also discusses the important problem of class imbalance in data and options to deal with it. In addition, this chapter compares visually the decision boundaries used by different algorithms. This will provide a better understanding of the model bias that different algorithms have.

Packages Used in this Chapter

```
pkgs <- c("basemodels", "C50", "caret", "e1071", "klaR",  
         "lattice", "MASS", "mlbench", "nnet", "palmerpenguins",  
         "randomForest", "rpart", "RWeka", "scales", "tidyverse",  
         "xgboost", "sampling")  
  
pkgs_install <- pkgs[!(pkgs %in% installed.packages()[,"Package"])]  
if(length(pkgs_install)) install.packages(pkgs_install)
```

The packages used for this chapter are:

- *basemodels* (Y.-J. Chen et al. 2023)
- *C50* (Kuhn and Quinlan 2025)
- *caret* (Kuhn 2024)
- *e1071* (Meyer et al. 2024)
- *klaR* (Roeve et al. 2023)
- *lattice* (Sarkar 2025)

- *MASS* (Ripley and Venables 2025)
- *mlbench* (Leisch and Dimitriadou 2024)
- *nnet* (Ripley 2025)
- *palmerpenguins* (Horst, Hill, and Gorman 2022)
- *randomForest* (Breiman et al. 2024)
- *rpart* (Therneau and Atkinson 2025)
- *RWeka* (Hornik 2023)
- *sampling* (Tillé and Matei 2025)
- *scales* (Wickham, Pedersen, and Seidel 2025)
- *tidyverse* (Wickham 2023b)
- *xgboost* (T. Chen et al. 2025)

4.1 Types of Classifiers

Many different classification algorithms¹ have been proposed in the literature. In this chapter, we will apply some of the more popular methods.

4.1.1 Set up the Training and Test Data

We will use the Zoo dataset which is included in the R package *mlbench* (you may have to install it). The Zoo dataset containing 17 (mostly logical) variables on different 101 animals as a data frame with 17 columns (hair, feathers, eggs, milk, airborne, aquatic, predator, toothed, backbone, breathes, venomous, fins, legs, tail, domestic, catsize, type). We convert the data frame into a tidyverse tibble (optional).

```
library(tidyverse)
data(Zoo, package = "mlbench")

Zoo <- as_tibble(Zoo)
Zoo
## # A tibble: 101 x 17
##   hair feathers eggs milk airborne aquatic predator
##   <lgl> <lgl>   <lgl> <lgl> <lgl>   <lgl>   <lgl>
## 1 TRUE  FALSE   FALSE TRUE  FALSE   FALSE   TRUE
## 2 TRUE  FALSE   FALSE TRUE  FALSE   FALSE   FALSE
## 3 FALSE FALSE   TRUE  FALSE FALSE   TRUE    TRUE
## 4 TRUE  FALSE   FALSE TRUE  FALSE   FALSE   TRUE
## 5 TRUE  FALSE   FALSE TRUE  FALSE   FALSE   TRUE
## 6 TRUE  FALSE   FALSE TRUE  FALSE   FALSE   FALSE
## 7 TRUE  FALSE   FALSE TRUE  FALSE   FALSE   FALSE
## 8 FALSE FALSE   TRUE  FALSE FALSE   TRUE    FALSE
```

¹https://en.wikipedia.org/wiki/Supervised_learning

```
## 9 FALSE FALSE TRUE FALSE FALSE TRUE TRUE
## 10 TRUE FALSE FALSE TRUE FALSE FALSE FALSE
## # i 91 more rows
## # i 10 more variables: toothed <lgl>, backbone <lgl>,
## #   breathes <lgl>, venomous <lgl>, fins <lgl>, legs <int>,
## #   tail <lgl>, domestic <lgl>, catsize <lgl>, type <fct>
```

We will use the package **caret**² to make preparing training sets and building classification (and regression) models easier. A great cheat sheet can be found here³.

```
library(caret)
```

Multi-core support can be used for cross-validation. **Note:** It is commented out here because it does not work with rJava used by the RWeka-based classifiers below.

```
##library(doMC, quietly = TRUE)
##registerDoMC(cores = 4)
##getDoParWorkers()
```

Test data is not used in the model building process and needs to be set aside purely for testing the model after it is completely built. Here I use 80% for training.

```
inTrain <- createDataPartition(y = Zoo$type, p = .8)[[1]]
Zoo_train <- Zoo |> slice(inTrain)
Zoo_test <- Zoo |> slice(-inTrain)
```

For hyperparameter tuning, we will use 10-fold cross-validation.

Note: Be careful if you have many NA values in your data. **train()** and cross-validation many fail in some cases. If that is the case then you can remove features (columns) which have many NAs, omit NAs using **na.omit()** or use imputation to replace them with reasonable values (e.g., by the feature mean or via kNN). Highly imbalanced datasets are also problematic since there is a chance that a fold does not contain examples of each class leading to a hard to understand error message.

²<https://topepo.github.io/caret/>

³https://ugoproto.github.io/ugo_r_doc/pdf/caret.pdf

4.2 Rule-based classifier: PART

```
rulesFit <- Zoo_train |> train(type ~ .,
  method = "PART",
  data = _,
  tuneLength = 5,
  trControl = trainControl(method = "cv"))
rulesFit
## Rule-Based Classifier
##
## 83 samples
## 16 predictors
## 7 classes: 'mammal', 'bird', 'reptile', 'fish', 'amphibian', 'insect', 'mollusc.etc'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 74, 74, 74, 74, 75, 75, ...
## Resampling results across tuning parameters:
##
##   threshold  pruned  Accuracy  Kappa
##   0.0100     yes     0.9542    0.9382
##   0.0100     no      0.9288    0.9060
##   0.1325     yes     0.9542    0.9382
##   0.1325     no      0.9288    0.9060
##   0.2550     yes     0.9542    0.9382
##   0.2550     no      0.9288    0.9060
##   0.3775     yes     0.9542    0.9382
##   0.3775     no      0.9288    0.9060
##   0.5000     yes     0.9542    0.9382
##   0.5000     no      0.9288    0.9060
##
## Accuracy was used to select the optimal model using
## the largest value.
## The final values used for the model were threshold =
## 0.5 and pruned = yes.
```

The model selection results are shown in the table. This is the selected model:

```
rulesFit$finalModel
## PART decision list
## -----
##
## feathersTRUE <= 0 AND
## milkTRUE > 0: mammal (33.0)
```



```
##
## feathersTRUE > 0: bird (16.0)
##
## backboneTRUE <= 0 AND
## airborneTRUE <= 0: mollusc.et.al (9.0/1.0)
##
## airborneTRUE <= 0 AND
## finsTRUE > 0: fish (11.0)
##
## airborneTRUE > 0: insect (6.0)
##
## aquaticTRUE > 0: amphibian (5.0/1.0)
##
## : reptile (3.0)
##
## Number of Rules : 7
```

PART returns a decision list, i.e., an ordered rule set. For example, the first rule shows that an animal with no feathers but milk is a mammal. In ordered rule sets, the decision of the first matching rule is used.

4.3 Nearest Neighbor Classifier

K-Nearest neighbor classifiers classify a new data point by looking at the majority class labels of its k nearest neighbors in the training data set. The used kNN implementation uses Euclidean distance to determine what data points are near by, so data needs to be standardized (scaled) first. Here legs are measured between 0 and 6 while all other variables are between 0 and 1. Scaling to z-scores can be directly performed as preprocessing in `train` using the parameter `preProcess = "scale"`.

The k value is typically chosen as an odd number so we get a clear majority.

```
knnFit <- Zoo_train |> train(type ~ .,
  method = "knn",
  data = _,
  preProcess = "scale",
  tuneGrid = data.frame(k = c(1, 3, 5, 7, 9)),
  trControl = trainControl(method = "cv"))
knnFit
## k-Nearest Neighbors
##
## 83 samples
## 16 predictors
```

```
## 7 classes: 'mammal', 'bird', 'reptile', 'fish', 'amphibian', 'insect', 'mollusc.et
##
## Pre-processing: scaled (16)
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 75, 73, 74, 75, 74, 74, ...
## Resampling results across tuning parameters:
##
## k Accuracy Kappa
## 1 0.9428 0.9256
## 3 0.9410 0.9234
## 5 0.9299 0.9088
## 7 0.9031 0.8728
## 9 0.9031 0.8721
##
## Accuracy was used to select the optimal model using
## the largest value.
## The final value used for the model was k = 1.
```

```
knnFit$finalModel
## 1-nearest neighbor model
## Training set outcome distribution:
##
##      mammal      bird      reptile      fish
##      33      16      4      11
##      amphibian      insect mollusc.et.al
##      4      7      8
```

kNN classifiers are lazy models meaning that instead of learning, they just keep the complete dataset. This is why the final model just gives us a summary statistic for the class labels in the training data.

4.4 Naive Bayes Classifier

Caret's train formula interface translates logicals and factors into dummy variables which the classifier interprets as numbers so it would use a Gaussian naive Bayes⁴ estimation. To avoid this, I directly specify `x` and `y`.

```
NBFit <- train(x = as.data.frame(Zoo_train[, -ncol(Zoo_train)]),
               y = pull(Zoo_train, "type"),
               method = "nb",
               tuneGrid = data.frame(fL = c(.2, .5, 1, 5),
                                     usekernel = TRUE, adjust = 1),
```

⁴https://en.wikipedia.org/wiki/Naive_Bayes_classifier#Gaussian_naive_Bayes

```

trControl = trainControl(method = "cv")
NBFit
## Naive Bayes
##
## 83 samples
## 16 predictors
## 7 classes: 'mammal', 'bird', 'reptile', 'fish', 'amphibian', 'insect', 'mollusc.et.al'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 74, 75, 74, 76, 74, 75, ...
## Resampling results across tuning parameters:
##
##   fL   Accuracy   Kappa
##   0.2  0.9274     0.9057
##   0.5  0.9274     0.9057
##   1.0  0.9163     0.8904
##   5.0  0.8927     0.8550
##
## Tuning parameter 'usekernel' was held constant at a
##   value of TRUE
## Tuning parameter 'adjust' was held
##   constant at a value of 1
## Accuracy was used to select the optimal model using
##   the largest value.
## The final values used for the model were fL =
##   0.2, usekernel = TRUE and adjust = 1.

```

The final model contains the prior probabilities for each class.

```

NBFit$finalModel$apriori
## grouping
##      mammal      bird      reptile      fish
##      0.39759      0.19277      0.04819      0.13253
##      amphibian      insect mollusc.et.al
##      0.04819      0.08434      0.09639

```

And the conditional probabilities as a table for each feature. For brevity, we only show the tables for the first three features. For example, the condition probability $P(\text{hair} = \text{TRUE} | \text{class} = \text{mammal})$ is 0.9641.

```

NBFit$finalModel$tables[1:3]
## $hair
##      var

```

```
## grouping      FALSE    TRUE
## mammal        0.03593 0.96407
## bird          0.98780 0.01220
## reptile       0.95455 0.04545
## fish          0.98246 0.01754
## amphibian     0.95455 0.04545
## insect        0.43243 0.56757
## mollusc.et.al 0.97619 0.02381
##
## $feathers
##               var
## grouping      FALSE    TRUE
## mammal        0.994012 0.005988
## bird          0.012195 0.987805
## reptile       0.954545 0.045455
## fish          0.982456 0.017544
## amphibian     0.954545 0.045455
## insect        0.972973 0.027027
## mollusc.et.al 0.976190 0.023810
##
## $eggs
##               var
## grouping      FALSE    TRUE
## mammal        0.96407 0.03593
## bird          0.01220 0.98780
## reptile       0.27273 0.72727
## fish          0.01754 0.98246
## amphibian     0.04545 0.95455
## insect        0.02703 0.97297
## mollusc.et.al 0.14286 0.85714
```

4.5 Bayesian Network

Bayesian networks are not covered here. R has very good support for modeling with Bayesian Networks. An example is the package `bnlearn`⁵.

4.6 Logistic Regression

Logistic regression is a very powerful classification method and should always be tried as one of the first models. A detailed discussion with more code is available in section Logistic Regression in the Appendix.

⁵<https://www.bnlearn.com/>

Regular logistic regression predicts only one outcome coded as a binary variable. Since we have data with several classes, we use multinomial logistic regression⁶, also called a log-linear model which is an extension of logistic regresses for multi-class problems. Caret uses `nnet::multinom()` which implements penalized multinomial regression.

```
logRegFit <- Zoo_train |> train(type ~ .,
  method = "multinom",
  data = _,
  trace = FALSE, # suppress some output
  trControl = trainControl(method = "cv"))
logRegFit
## Penalized Multinomial Regression
##
## 83 samples
## 16 predictors
## 7 classes: 'mammal', 'bird', 'reptile', 'fish', 'amphibian', 'insect', 'mollusc.et.al'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 75, 76, 75, 74, 74, 74, ...
## Resampling results across tuning parameters:
##
##      decay  Accuracy  Kappa
##      0e+00  0.8704    0.8306
##      1e-04  0.9038    0.8749
##      1e-01  0.9056    0.8777
##
## Accuracy was used to select the optimal model using
## the largest value.
## The final value used for the model was decay = 0.1.
```

```
logRegFit$finalModel
## Call:
## nnet::multinom(formula = .outcome ~ ., data = dat, decay = param$decay,
##      trace = FALSE)
##
## Coefficients:
##              (Intercept) hairTRUE feathersTRUE eggsTRUE
## bird              -0.33330  -1.0472           2.9696   0.8278
## reptile             0.01303  -2.0808          -1.0891   0.6731
## fish              -0.17462  -0.2762          -0.1135   1.8817
## amphibian          -1.28295  -1.5165          -0.2698   0.6801
## insect             -0.75300  -0.3903          -0.1445   0.8980
```

⁶https://en.wikipedia.org/wiki/Multinomial_logistic_regression

```

## mollusc.et.al      1.52104 -1.2287      -0.2492      0.9320
##                  milkTRUE airborneTRUE aquaticTRUE
## bird              -1.2523      1.17310    -0.1594
## reptile           -2.1800      -0.51796    -1.0890
## fish              -1.3571      -0.09009     0.5093
## amphibian         -1.6014      -0.36649     1.6271
## insect            -1.0130      1.37404     -1.0752
## mollusc.et.al     -0.9035      -1.17882     0.7160
##                  predatorTRUE toothedTRUE backboneTRUE
## bird              0.22312      -1.7846      0.4736
## reptile           0.04172      -0.2003      0.8968
## fish             -0.33094      0.4118      0.2768
## amphibian        -0.13993      0.7399      0.2557
## insect           -1.11743      -1.1852     -1.5725
## mollusc.et.al     0.83070      -1.7390     -2.6045
##                  breathesTRUE venomousTRUE finsTRUE      legs
## bird              0.1337      -0.3278 -0.545979 -0.59910
## reptile          -0.5039      1.2776 -1.192197 -0.24200
## fish            -1.9709      -0.4204  1.472416 -1.15775
## amphibian        0.4594      0.1611 -0.628746  0.09302
## insect           0.1341      -0.2567 -0.002527  0.59118
## mollusc.et.al    -0.6287      0.8411 -0.206104  0.12091
##                  tailTRUE domesticTRUE catsizeTRUE
## bird              0.5947      0.14176    -0.1182
## reptile           1.1863      -0.40893    -0.4305
## fish              0.3226      0.08636    -0.3132
## amphibian        -1.3529      -0.40545    -1.3581
## insect           -1.6908      -0.24924    -1.0416
## mollusc.et.al    -0.7353      -0.22601    -0.7079
##
## Residual Deviance: 35.46
## AIC: 239.5

```

The coefficients are log odds ratios measured against the default class (here mammal). A negative log odds ratio means that the odds go down with an increase in the value of the predictor. A predictor with a positive log-odds ratio increases the odds. For example, in the model above, `hair=TRUE` has a negative coefficient for bird but `feathers=TRUE` has a large positive coefficient.

4.7 Artificial Neural Network (ANN)

Standard networks have an input layer, an output layer and in between a single hidden layer.

```

nnetFit <- Zoo_train |> train(type ~ .,
  method = "nnet",
  data = _,
  tuneLength = 5,
  trControl = trainControl(method = "cv"),
  trace = FALSE # no progress output
)
nnetFit
## Neural Network
##
## 83 samples
## 16 predictors
## 7 classes: 'mammal', 'bird', 'reptile', 'fish', 'amphibian', 'insect', 'mollusc.etal'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 75, 74, 76, 76, 75, 74, ...
## Resampling results across tuning parameters:
##
##   size  decay  Accuracy  Kappa
##   1     0e+00  0.7536    0.6369
##   1     1e-04  0.6551    0.4869
##   1     1e-03  0.8201    0.7562
##   1     1e-02  0.7979    0.7285
##   1     1e-01  0.7263    0.6265
##   3     0e+00  0.8344    0.7726
##   3     1e-04  0.8219    0.7622
##   3     1e-03  0.8798    0.8420
##   3     1e-02  0.9063    0.8752
##   3     1e-01  0.8809    0.8405
##   5     0e+00  0.8319    0.7768
##   5     1e-04  0.8448    0.7922
##   5     1e-03  0.9020    0.8660
##   5     1e-02  0.9131    0.8816
##   5     1e-01  0.9020    0.8680
##   7     0e+00  0.8405    0.7893
##   7     1e-04  0.9031    0.8689
##   7     1e-03  0.9020    0.8678
##   7     1e-02  0.9131    0.8816
##   7     1e-01  0.8909    0.8539
##   9     0e+00  0.9145    0.8847
##   9     1e-04  0.9131    0.8804
##   9     1e-03  0.9242    0.8974
##   9     1e-02  0.9131    0.8818
##   9     1e-01  0.9020    0.8680

```

```
##
## Accuracy was used to select the optimal model using
## the largest value.
## The final values used for the model were size = 9 and
## decay = 0.001.
```

The input layer has a size of 16, one for each input feature and the output layer has a size of 7 representing the 7 classes. Model selection chose a network architecture with a hidden layer with 9 units resulting in 223 learned weights. Since the model is considered a black-box model only the network architecture and the used variables are shown as a summary.

```
nnetFit$finalModel
## a 16-9-7 network with 223 weights
## inputs: hairTRUE feathersTRUE eggsTRUE milkTRUE airborneTRUE aquaticTRUE predatorTR
## output(s): .outcome
## options were - softmax modelling decay=0.001
```

For deep Learning, R offers packages for using tensorflow⁷ and Keras⁸.

4.8 Support Vector Machines

```
svmFit <- Zoo_train |> train(type ~.,
  method = "svmLinear",
  data = _,
  tuneLength = 5,
  trControl = trainControl(method = "cv"))
svmFit
## Support Vector Machines with Linear Kernel
##
## 83 samples
## 16 predictors
## 7 classes: 'mammal', 'bird', 'reptile', 'fish', 'amphibian', 'insect', 'mollusc.et
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 77, 75, 74, 76, 75, 74, ...
## Resampling results:
##
```

⁷<https://tensorflow.rstudio.com/>

⁸<https://keras3.posit.co/>


```
## Accuracy Kappa
## 0.9317 0.9105
##
## Tuning parameter 'C' was held constant at a value of 1
```

We use a linear support vector machine. Support vector machines can use kernels to create non-linear decision boundaries. `method` above can be changed to "svmPoly" or "svmRadial" to use kernels. The choice of kernel is typically made by experimentation.

The support vectors determining the decision boundary are stored in the model.

```
svmFit$finalModel
## Support Vector Machine object of class "ksvm"
##
## SV type: C-svc (classification)
## parameter : cost C = 1
##
## Linear (vanilla) kernel function.
##
## Number of Support Vectors : 42
##
## Objective Function Value : -0.1432 -0.22 -0.1501 -0.1756 -0.0943 -0.1047 -0.2804 -0.0808 -0.15
## Training error : 0
```

4.9 Ensemble Methods

Many ensemble methods are available in R. We only cover here code for two popular methods.

4.9.1 Random Forest

```
randomForestFit <- Zoo_train |> train(type ~ .,
  method = "rf",
  data = _,
  tuneLength = 5,
  trControl = trainControl(method = "cv"))
randomForestFit
## Random Forest
##
## 83 samples
```

```
## 16 predictors
## 7 classes: 'mammal', 'bird', 'reptile', 'fish', 'amphibian', 'insect', 'mollusc.et
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 73, 72, 76, 75, 73, 77, ...
## Resampling results across tuning parameters:
##
##      mtry  Accuracy  Kappa
##      2    0.9287    0.9054
##      5    0.9432    0.9246
##      9    0.9398    0.9197
##     12    0.9498    0.9331
##     16    0.9498    0.9331
##
## Accuracy was used to select the optimal model using
## the largest value.
## The final value used for the model was mtry = 12.
```

The default number of trees is 500 and `mtry` determines the number of variables randomly sampled as candidates at each split. This number is a tradeoff where a larger number allows each tree to pick better splits, but a smaller number increases the independence between trees.

```
randomForestFit$finalModel
##
## Call:
## randomForest(x = x, y = y, mtry = param$mtry)
##              Type of random forest: classification
##              Number of trees: 500
## No. of variables tried at each split: 12
##
##              OOB estimate of error rate: 6.02%
## Confusion matrix:
##              mammal bird reptile fish amphibian insect
## mammal          33    0      0    0          0      0
## bird             0   16      0    0          0      0
## reptile          0    0      3    0          1      0
## fish             0    0      0   11          0      0
## amphibian        0    0      1    0          3      0
## insect           0    0      0    0          0      6
## mollusc.et.al    0    0      0    0          0      2
##              mollusc.et.al class.error
## mammal                0      0.0000
## bird                   0      0.0000
```

```
## reptile          0      0.2500
## fish            0      0.0000
## amphibian       0      0.2500
## insect          1      0.1429
## mollusc.et.al   6      0.2500
```

The model is a set of 500 trees and the prediction is made by applying all trees and then using the majority vote.

Since random forests use bagging (bootstrap sampling to train trees), the remaining data can be used like a test set. The resulting error is called out-of-bag (OOB) error and gives an estimate for the generalization error. The model above also shows the confusion matrix based on the OOB error.

4.9.2 Gradient Boosted Decision Trees (xgboost)

The idea of gradient boosting is to learn a base model and then to learn successive models to predict and correct the error of all previous models. Typically, tree models are used.

```
xgboostFit <- Zoo_train |> train(type ~ .,
  method = "xgbTree",
  data = _,
  tuneLength = 5,
  trControl = trainControl(method = "cv"),
  tuneGrid = expand.grid(
    nrounds = 20,
    max_depth = 3,
    colsample_bytree = .6,
    eta = 0.1,
    gamma=0,
    min_child_weight = 1,
    subsample = .5
  ))
xgboostFit
## eXtreme Gradient Boosting
##
## 83 samples
## 16 predictors
## 7 classes: 'mammal', 'bird', 'reptile', 'fish', 'amphibian', 'insect', 'mollusc.et.al'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 74, 74, 76, 73, 76, 74, ...
## Resampling results:
```

```
##
## Accuracy Kappa
## 0.92      0.893
##
## Tuning parameter 'nrounds' was held constant at a value
## held constant at a value of 1
## Tuning parameter
## 'subsample' was held constant at a value of 0.5
```

The final model is a complicated set of trees, so only some summary information is shown.

```
xgboostFit$finalModel
## ##### xgb.Booster
## raw: 111.6 Kb
## call:
## xgboost::xgb.train(params = list(eta = param$eta, max_depth = param$max_depth,
## gamma = param$gamma, colsample_bytree = param$colsample_bytree,
## min_child_weight = param$min_child_weight, subsample = param$subsample),
## data = x, nrounds = param$nrounds, num_class = length(lev),
## objective = "multi:softprob")
## params (as set within xgb.train):
## eta = "0.1", max_depth = "3", gamma = "0", colsample_bytree = "0.6", min_child_weight = "1"
## xgb.attributes:
## niter
## callbacks:
## cb.print.evaluation(period = print_every_n)
## # of features: 16
## niter: 20
## nfeatures : 16
## xNames : hairTRUE feathersTRUE eggsTRUE milkTRUE airborneTRUE aquaticTRUE predatorTRUE
## problemType : Classification
## tuneValue :
## nrounds max_depth eta gamma colsample_bytree
## 1 20 3 0.1 0 0.6
## min_child_weight subsample
## 1 1 0.5
## obsLevels : mammal bird reptile fish amphibian insect mollusc.et.al
## param :
## list()
```

4.10 Model Comparison

We first create a weak baseline model that always predicts the the majority class mammal.

```
baselineFit <- Zoo_train |> train(type ~ .,
  method = basemodels::dummyClassifier,
  data = _,
  strategy = "constant",
  constant = "mammal",
  trControl = trainControl(method = "cv"
    ))

baselineFit
## dummyClassifier
##
## 83 samples
## 16 predictors
## 7 classes: 'mammal', 'bird', 'reptile', 'fish', 'amphibian', 'insect', 'mollusc.et.al'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 76, 75, 75, 74, 76, 76, ...
## Resampling results:
##
## Accuracy Kappa
## 0.4025 0
```

The kappa of 0 clearly indicates that the baseline model has no power.

We collect the performance metrics from the models trained on the same data.

```
resamps <- resamples(list(
  baseline = baselineFit,
  PART = rulesFit,
  kNearestNeighbors = knnFit,
  NBayes = NBFit,
  logReg = logRegFit,
  ANN = nnetFit,
  SVM = svmFit,
  RandomForest = randomForestFit,
  XGBoost = xgboostFit
))

resamps
##
## Call:
```

```
## resamples.default(x = list(baseline = baselineFit, PART
## svmFit, RandomForest = randomForestFit, XGBoost
## = xgboostFit))
##
## Models: baseline, PART, kNearestNeighbors, NBayes, logReg, ANN, SVM, RandomForest,
## Number of resamples: 10
## Performance metrics: Accuracy, Kappa
## Time estimates for: everything, final model fit
```

The summary statistics shows the performance. We see that all methods do well on this easy data set, only the baseline model performs as expected poorly.

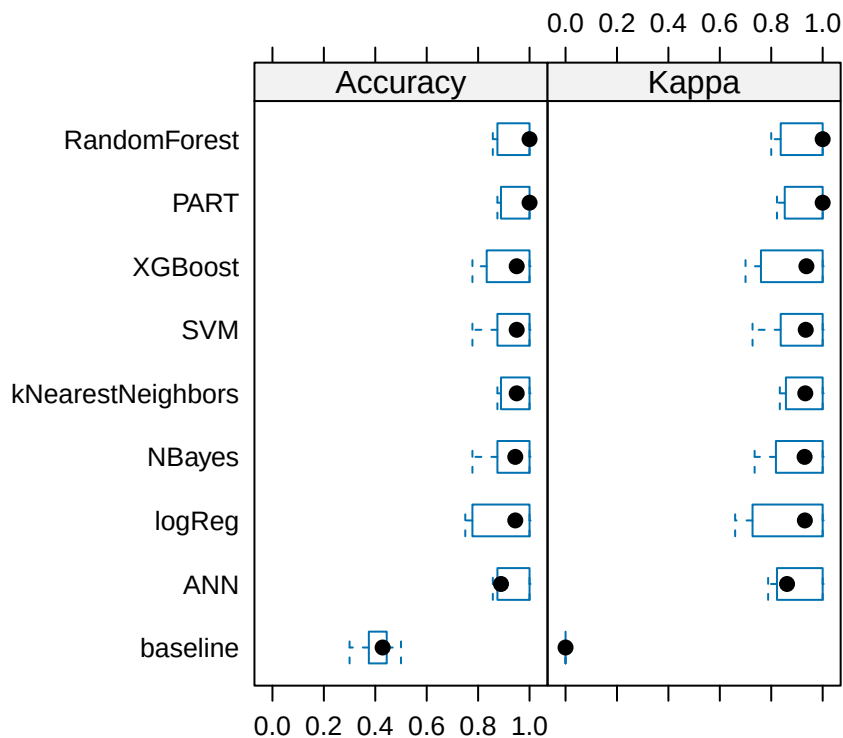
```
summary(resamps)
##
## Call:
## summary.resamples(object = resamps)
##
## Models: baseline, PART, kNearestNeighbors, NBayes, logReg, ANN, SVM, RandomForest,
## Number of resamples: 10
##
## Accuracy
##
```

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
## baseline	0.3000	0.3750	0.4286	0.4025	0.4405	0.5
## PART	0.8750	0.8889	1.0000	0.9542	1.0000	1.0
## kNearestNeighbors	0.8750	0.8889	0.9500	0.9428	1.0000	1.0
## NBayes	0.7778	0.8750	0.9444	0.9274	1.0000	1.0
## logReg	0.7500	0.8056	0.9444	0.9056	1.0000	1.0
## ANN	0.8571	0.8750	0.8889	0.9242	1.0000	1.0
## SVM	0.7778	0.8785	0.9500	0.9317	1.0000	1.0
## RandomForest	0.8571	0.8835	1.0000	0.9498	1.0000	1.0
## XGBoost	0.7778	0.8472	0.9500	0.9200	1.0000	1.0

```
##
## NA's
## baseline      0
## PART          0
## kNearestNeighbors 0
## NBayes        0
## logReg        0
## ANN           0
## SVM           0
## RandomForest  0
## XGBoost       0
##
## Kappa
##
```

	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
## baseline	0.0000	0.0000	0.0000	0.0000	0	0

```
## PART 0.8222 0.8542 1.0000 0.9382 1 1
## kNearestNeighbors 0.8333 0.8588 0.9324 0.9256 1 1
## NBayes 0.7353 0.8220 0.9297 0.9057 1 1
## logReg 0.6596 0.7560 0.9308 0.8777 1 1
## ANN 0.7879 0.8241 0.8615 0.8974 1 1
## SVM 0.7273 0.8413 0.9342 0.9105 1 1
## RandomForest 0.8000 0.8483 1.0000 0.9331 1 1
## XGBoost 0.7000 0.7848 0.9367 0.8930 1 1
## NA's
## baseline 0
## PART 0
## kNearestNeighbors 0
## NBayes 0
## logReg 0
## ANN 0
## SVM 0
## RandomForest 0
## XGBoost 0
library(lattice)
bwplot(resamps, layout = c(3, 1))
```



Perform inference about differences between models. For each metric, all pair-

wise differences are computed and tested to assess if the difference is equal to zero. By default Bonferroni correction for multiple comparison is used. Differences are shown in the upper triangle and p-values are in the lower triangle.

```
difs <- diff(resamps)
summary(difs)
##
## Call:
## summary.diff.resamples(object = difs)
##
## p-value adjustment: bonferroni
## Upper diagonal: estimates of the difference
## Lower diagonal: p-value for H0: difference = 0
##
## Accuracy
##
```

	baseline	PART	kNearestNeighbors
## baseline		-0.55171	-0.54032
## PART	8.37e-07		0.01139
## kNearestNeighbors	3.20e-07	1	
## NBayes	7.48e-06	1	1
## logReg	1.81e-05	1	1
## ANN	9.66e-06	1	1
## SVM	2.87e-06	1	1
## RandomForest	9.73e-07	1	1
## XGBoost	2.04e-05	1	1

```
##
```

	NBayes	logReg	ANN	SVM
## baseline	-0.52492	-0.50310	-0.52175	-0.52921
## PART	0.02679	0.04861	0.02996	0.02250
## kNearestNeighbors	0.01540	0.03722	0.01857	0.01111
## NBayes		0.02183	0.00317	-0.00429
## logReg	1		-0.01865	-0.02611
## ANN	1	1		-0.00746
## SVM	1	1	1	
## RandomForest	1	1	1	1
## XGBoost	1	1	1	1

```
##
```

	RandomForest	XGBoost
## baseline	-0.54738	-0.51754
## PART	0.00433	0.03417
## kNearestNeighbors	-0.00706	0.02278
## NBayes	-0.02246	0.00738
## logReg	-0.04428	-0.01444
## ANN	-0.02563	0.00421
## SVM	-0.01817	0.01167
## RandomForest		0.02984
## XGBoost	1	

```
##
```



```
## Kappa
##          baseline PART      kNearestNeighbors
## baseline          -0.93815 -0.92557
## PART          1.41e-09      0.01258
## kNearestNeighbors 1.37e-09 1
## NBayes          1.94e-08 1      1
## logReg          4.17e-07 1      1
## ANN          6.28e-09 1      1
## SVM          1.45e-08 1      1
## RandomForest    3.67e-09 1      1
## XGBoost        1.03e-07 1      1
##          NBayes logReg ANN      SVM
## baseline    -0.90570 -0.87768 -0.89737 -0.91054
## PART         0.03245  0.06047  0.04078  0.02761
## kNearestNeighbors 0.01987  0.04789  0.02820  0.01503
## NBayes              0.02802  0.00833 -0.00485
## logReg             1              -0.01969 -0.03287
## ANN             1      1              -0.01317
## SVM             1      1      1
## RandomForest    1      1      1      1
## XGBoost         1      1      1      1
##          RandomForest XGBoost
## baseline    -0.93305      -0.89296
## PART         0.00510      0.04519
## kNearestNeighbors -0.00748      0.03261
## NBayes       -0.02735      0.01274
## logReg       -0.05538     -0.01529
## ANN         -0.03568      0.00441
## SVM         -0.02251      0.01758
## RandomForest    0.04009
## XGBoost         1
```

All perform similarly well except the baseline model (differences in the first row are negative and the p-values in the first column are $<.05$ indicating that the null-hypothesis of a difference of 0 can be rejected). Most models do similarly well on the data. We choose here the random forest model and evaluate its generalization performance on the held-out test set.

```
pr <- predict(randomForestFit, Zoo_test)
pr
## [1] mammal      fish      mollusc.et.al fish
## [5] mammal      insect    mammal      mammal
## [9] mammal      mammal    bird        mammal
## [13] mammal      bird      reptile     bird
## [17] mollusc.et.al bird
```

```
## 7 Levels: mammal bird reptile fish amphibian ... mollusc.et.al
```

Calculate the confusion matrix for the held-out test data.

```
confusionMatrix(pr, reference = Zoo_test$type)
```

```
## Confusion Matrix and Statistics
```

```
##
```

```
##           Reference
## Prediction  mammal bird reptile fish amphibian insect
## mammal      8     0     0     0         0     0
## bird        0     4     0     0         0     0
## reptile     0     0     1     0         0     0
## fish        0     0     0     2         0     0
## amphibian   0     0     0     0         0     0
## insect      0     0     0     0         0     1
## mollusc.et.al 0     0     0     0         0     0
```

```
##           Reference
## Prediction  mollusc.et.al
## mammal      0
## bird        0
## reptile     0
## fish        0
## amphibian   0
## insect      0
## mollusc.et.al 2
```

```
## Overall Statistics
```

```
##
```

```
##           Accuracy : 1
##           95% CI : (0.815, 1)
##    No Information Rate : 0.444
##    P-Value [Acc > NIR] : 4.58e-07
```

```
##
```

```
##           Kappa : 1
```

```
##
```

```
##    McNemar's Test P-Value : NA
```

```
##
```

```
## Statistics by Class:
```

```
##
```

```
##           Class: mammal Class: bird
## Sensitivity           1.000           1.000
## Specificity           1.000           1.000
## Pos Pred Value         1.000           1.000
## Neg Pred Value         1.000           1.000
## Prevalence             0.444           0.222
```

```

## Detection Rate          0.444      0.222
## Detection Prevalence    0.444      0.222
## Balanced Accuracy       1.000      1.000
##                          Class: reptile Class: fish
## Sensitivity             1.0000     1.000
## Specificity             1.0000     1.000
## Pos Pred Value          1.0000     1.000
## Neg Pred Value          1.0000     1.000
## Prevalence              0.0556     0.111
## Detection Rate          0.0556     0.111
## Detection Prevalence    0.0556     0.111
## Balanced Accuracy       1.0000     1.000
##                          Class: amphibian Class: insect
## Sensitivity             NA         1.0000
## Specificity             1         1.0000
## Pos Pred Value          NA         1.0000
## Neg Pred Value          NA         1.0000
## Prevalence              0         0.0556
## Detection Rate          0         0.0556
## Detection Prevalence    0         0.0556
## Balanced Accuracy       NA         1.0000
##                          Class: mollusc.et.al
## Sensitivity             1.000
## Specificity             1.000
## Pos Pred Value          1.000
## Neg Pred Value          1.000
## Prevalence              0.111
## Detection Rate          0.111
## Detection Prevalence    0.111
## Balanced Accuracy       1.000

```

4.11 Class Imbalance

Classifiers have a hard time to learn from data where we have much more observations for one class (called the majority class). This is called the **class imbalance problem** which is especially problematic when it is important to identify members of the minority class. In this setting the minority class is also often called the positive class since it has the property that we want to identify. An example would be that we want to identify the patients that are positive for a rare disease.

Here is a very good article about the problem and solutions.⁹

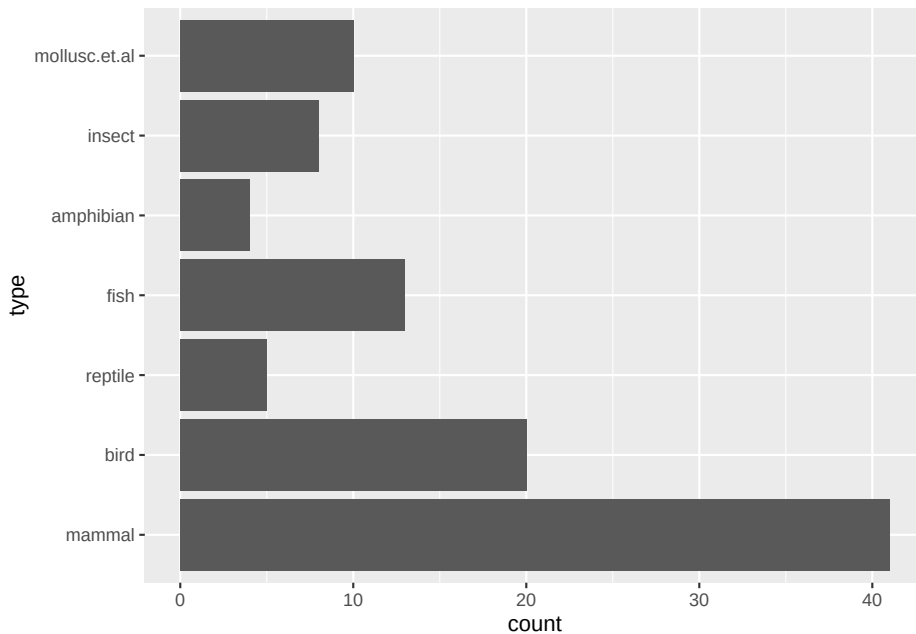
⁹<http://www.kdnuggets.com/2016/08/learning-from-imbalanced-classes.html>

For the examples, we will use the Zoo dataset.

```
library(rpart)
library(rpart.plot)
data(Zoo, package = "mlbench")
```

You always need to check if you have a class imbalance problem. Just look at the class distribution.

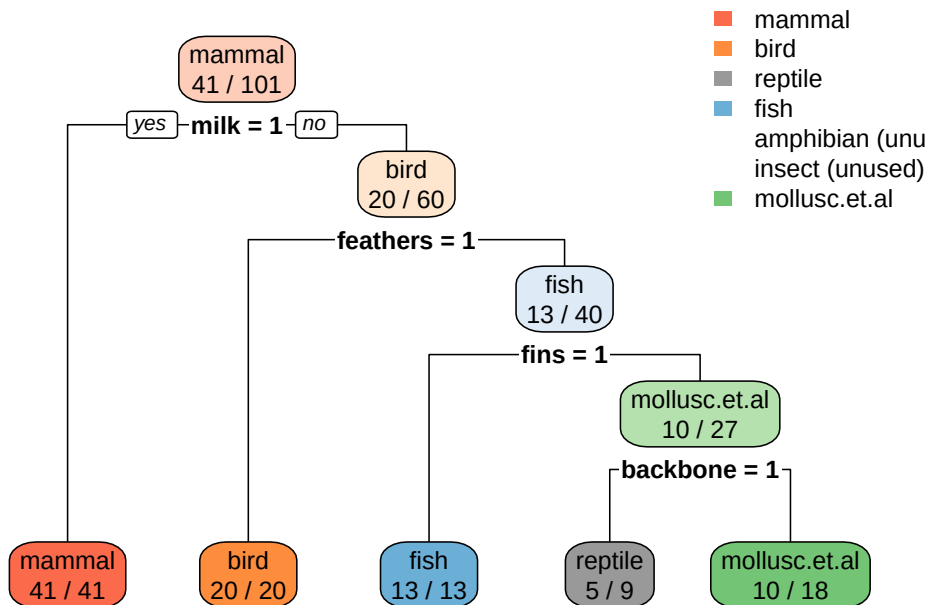
```
ggplot(Zoo, aes(y = type)) + geom_bar()
```



We see that there is moderate class imbalance with very many mammals and few amphibians and reptiles in the data set. If we build a classifier, then it is very likely that it will not perform well for identifying amphibians or reptiles. This effect was seen when building a simple decision tree.

```
tree_default <- Zoo |>
  rpart(type ~ ., data = _)
tree_default
## n= 101
##
## node), split, n, loss, yval, (yprob)
##      * denotes terminal node
##
```

```
## 1) root 101 60 mammal (0.41 0.2 0.05 0.13 0.04 0.079 0.099)
## 2) milk >= 0.5 41 0 mammal (1 0 0 0 0 0) *
## 3) milk < 0.5 60 40 bird (0 0.33 0.083 0.22 0.067 0.13 0.17)
## 6) feathers >= 0.5 20 0 bird (0 1 0 0 0 0) *
## 7) feathers < 0.5 40 27 fish (0 0 0.12 0.33 0.1 0.2 0.25)
## 14) fins >= 0.5 13 0 fish (0 0 0 1 0 0) *
## 15) fins < 0.5 27 17 mollusc.et.al (0 0 0.19 0 0.15 0.3 0.37)
## 30) backbone >= 0.5 9 4 reptile (0 0 0.56 0 0.44 0 0) *
## 31) backbone < 0.5 18 8 mollusc.et.al (0 0 0 0 0 0.44 0.56) *
library(rpart.plot)
rpart.plot(tree_default, extra = 2)
```



The resulting tree has no leaf nodes for these rare animal types meaning that it cannot identify them.

Deciding if an animal is a reptile or not is a very class imbalanced problem. We set up this problem by changing the class variable to make it into a binary reptile/non-reptile classification problem.

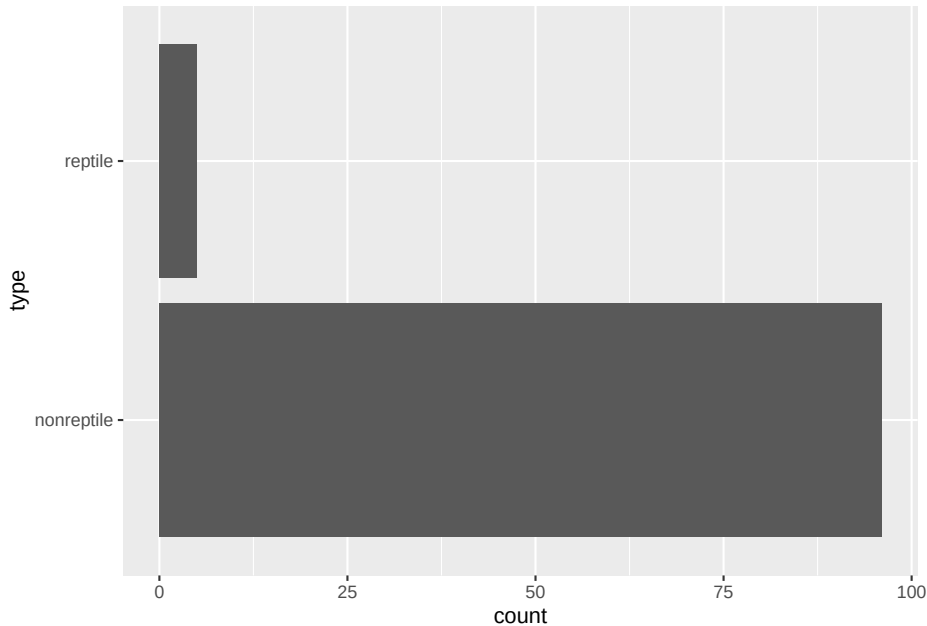
```
Zoo_reptile <- Zoo |>
  mutate(type = factor(Zoo$type == "reptile",
    levels = c(FALSE, TRUE),
    labels = c("nonreptile", "reptile")))
```

Do not forget to make the class variable a factor (a nominal variable) or you will get a regression tree instead of a classification tree.

By checking the class distribution, we see that classification model is now highly imbalances with only 5 reptiles vs. 96 non-reptiles.

```
summary(Zoo_reptile)
##      hair      feathers      eggs
## Mode :logical Mode :logical Mode :logical
## FALSE:58      FALSE:81      FALSE:42
## TRUE :43      TRUE :20      TRUE :59
##
##
##
##      milk      airborne      aquatic
## Mode :logical Mode :logical Mode :logical
## FALSE:60      FALSE:77      FALSE:65
## TRUE :41      TRUE :24      TRUE :36
##
##
##
##      predator      toothed      backbone
## Mode :logical Mode :logical Mode :logical
## FALSE:45      FALSE:40      FALSE:18
## TRUE :56      TRUE :61      TRUE :83
##
##
##
##      breathes      venomous      fins
## Mode :logical Mode :logical Mode :logical
## FALSE:21      FALSE:93      FALSE:84
## TRUE :80      TRUE :8      TRUE :17
##
##
##
##      legs      tail      domestic
## Min. :0.00 Mode :logical Mode :logical
## 1st Qu.:2.00 FALSE:26      FALSE:88
## Median :4.00 TRUE :75      TRUE :13
## Mean :2.84
## 3rd Qu.:4.00
## Max. :8.00
##      catsize      type
## Mode :logical nonreptile:96
## FALSE:57      reptile : 5
## TRUE :44
##
##
##
```

```
ggplot(Zoo_reptile, aes(y = type)) + geom_bar()
```



The graph shows that the data is highly imbalanced with encountering a non-reptile being about 20 times more likely than encountering a reptile. We will try several different options in the next few subsections.

We split the data into test and training data so we can evaluate how well the classifiers for the different options work. I use here a 50/50 split to make sure that the test set has some samples of the rare reptile class.

```
set.seed(1234)

inTrain <- createDataPartition(y = Zoo_reptile$type, p = .5)[[1]]
training_reptile <- Zoo_reptile |> slice(inTrain)
testing_reptile <- Zoo_reptile |> slice(-inTrain)
```

4.11.1 Option 1: Use the Data As Is and Hope For The Best

```
fit <- training_reptile |>
  train(type ~ .,
        data = _,
        method = "rpart",
```

```

trControl = trainControl(method = "cv"))
## Warning in nominalTrainWorkflow(x = x, y = y, wts =
## weights, info = trainInfo, : There were missing values in
## resampled performance measures.

```

Warnings: “There were missing values in resampled performance measures.” means that some test folds used in hyper parameter tuning did not contain examples of both classes. This is very likely with strong class imbalance and small datasets.

```

fit
## CART
##
## 51 samples
## 16 predictors
## 2 classes: 'nonreptile', 'reptile'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 46, 47, 46, 46, 45, 46, ...
## Resampling results:
##
## Accuracy Kappa
## 0.9467 0
##
## Tuning parameter 'cp' was held constant at a value of 0
rpart.plot(fit$finalModel, extra = 2)

```

nonreptile
48 / 51

The tree predicts everything as non-reptile. Have a look at the error on the test set.

```

confusionMatrix(data = predict(fit, testing_reptile),
  ref = testing_reptile$type, positive = "reptile")
## Confusion Matrix and Statistics
##
##              Reference
## Prediction nonreptile reptile
## nonreptile      48      2
## reptile         0      0
##
##              Accuracy : 0.96

```



```
##          95% CI : (0.863, 0.995)
##      No Information Rate : 0.96
##      P-Value [Acc > NIR] : 0.677
##
##          Kappa : 0
##
##      McNemar's Test P-Value : 0.480
##
##          Sensitivity : 0.00
##          Specificity : 1.00
##          Pos Pred Value : NaN
##          Neg Pred Value : 0.96
##          Prevalence : 0.04
##          Detection Rate : 0.00
##      Detection Prevalence : 0.00
##          Balanced Accuracy : 0.50
##
##          'Positive' Class : reptile
##
```

Accuracy is high, but it is exactly the same as the no-information rate and kappa is zero. Sensitivity is also zero, meaning that we do not identify any positive (reptile). If the cost of missing a positive is much larger than the cost associated with misclassifying a negative, then accuracy is not a good measure! By dealing with imbalance, we are **not** concerned with accuracy, but we want to increase the sensitivity, i.e., the chance to identify positive examples. **This approach does not work!**

Note: The positive class value (the one that you want to detect) is set manually to reptile using `positive = "reptile"`. Otherwise sensitivity/specificity will not be correctly calculated.

4.11.2 Option 2: Balance Data With Resampling

A popular choice is to balance the class distribution by using stratified sampling¹⁰ with replacement to oversample the minority/positive class. Other options include SMOTE¹¹ (in package **DMwR**) or other sampling strategies (e.g., from package **unbalanced**).

Here, we use stratified sampling with 50 observations per class. **Note:** many samples of the positive/minority class will be chosen several times.

¹⁰https://en.wikipedia.org/wiki/Stratified_sampling

¹¹https://en.wikipedia.org/wiki/Synthetic_minority_oversampling_technique

```

library(sampling)
##
## Attaching package: 'sampling'
## The following object is masked from 'package:caret':
##
##      cluster
set.seed(1000) # for repeatability

id <- strata(training_reptile, stratanames = "type",
             size = c(50, 50), method = "srswr")
training_reptile_balanced <- training_reptile |>
  slice(id$ID_unit)

table(training_reptile_balanced$type)
##
## nonreptile    reptile
##          50         50

```

Build a tree using the balanced data.

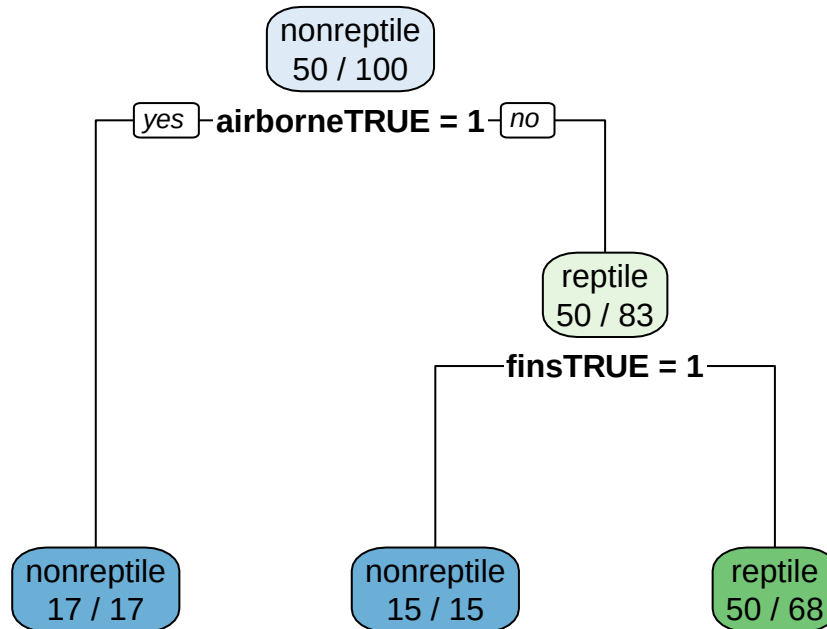
```

fit <- training_reptile_balanced |>
  train(type ~ .,
        data = _,
        method = "rpart",
        trControl = trainControl(method = "cv"),
        control = rpart.control(minsplit = 5))

fit
## CART
##
## 100 samples
## 16 predictor
## 2 classes: 'nonreptile', 'reptile'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 90, 90, 90, 90, 90, 90, ...
## Resampling results across tuning parameters:
##
##   cp    Accuracy  Kappa
## 0.18  0.81      0.62
## 0.30  0.63      0.26
## 0.34  0.53      0.06
##
## Accuracy was used to select the optimal model using

```

```
## the largest value.
## The final value used for the model was cp = 0.18.
rpart.plot(fit$finalModel, extra = 2)
```



The resulting tree has a reasonable accuracy/kappa on the balanced data. However, the real data does not have the distribution of the balanced data, so we check the model on the unbalanced testing data.

```
confusionMatrix(data = predict(fit, testing_reptile),
  ref = testing_reptile$type, positive = "reptile")
## Confusion Matrix and Statistics
##
##              Reference
## Prediction  nonreptile reptile
## nonreptile      19      0
## reptile         29      2
##
##              Accuracy : 0.42
##              95% CI : (0.282, 0.568)
##      No Information Rate : 0.96
##      P-Value [Acc > NIR] : 1
##
##              Kappa : 0.05
##
##      McNemar's Test P-Value : 2e-07
```

```
##
##          Sensitivity : 1.0000
##          Specificity : 0.3958
##          Pos Pred Value : 0.0645
##          Neg Pred Value : 1.0000
##          Prevalence : 0.0400
##          Detection Rate : 0.0400
##          Detection Prevalence : 0.6200
##          Balanced Accuracy : 0.6979
##
##          'Positive' Class : reptile
##
```

The accuracy is below the no information rate and kappa is close to 0! However, sensitivity (the ability to identify reptiles) is now one meaning that the classifier is able to identify all reptiles.

We have to make a tradeoff between sensitivity (identifying reptiles) and specificity (how many of the identified animals are really reptiles). The tradeoff can be controlled using the sample proportions. We can sample more reptiles to increase sensitivity at the cost of lower specificity (this effect cannot be seen in the data since the test set has only a few reptiles).

```
id <- strata(training_reptile, stratanames = "type",
             size = c(50, 500), method = "srswr")
training_reptile_balanced <- training_reptile |>
  slice(id$ID_unit)
table(training_reptile_balanced$type)
##
## nonreptile   reptile
##          50      500
fit <- training_reptile_balanced |>
  train(type ~ .,
        data = _,
        method = "rpart",
        trControl = trainControl(method = "cv"),
        control = rpart.control(minsplit = 5))

confusionMatrix(data = predict(fit, testing_reptile),
                 ref = testing_reptile$type, positive = "reptile")
## Confusion Matrix and Statistics
##
##              Reference
## Prediction nonreptile reptile
## nonreptile      33      0
## reptile         15      2
```

```
##
##           Accuracy : 0.7
##           95% CI : (0.554, 0.821)
##       No Information Rate : 0.96
##       P-Value [Acc > NIR] : 1.000000
##
##           Kappa : 0.15
##
##  McNemar's Test P-Value : 0.000301
##
##       Sensitivity : 1.000
##       Specificity : 0.688
##       Pos Pred Value : 0.118
##       Neg Pred Value : 1.000
##       Prevalence : 0.040
##       Detection Rate : 0.040
##       Detection Prevalence : 0.340
##       Balanced Accuracy : 0.844
##
##       'Positive' Class : reptile
##
```

Note: We have used a sample ratio of 1:10 meaning that we value identifying 1 reptile as much as missclassifying 10 non-reptiles. This is related to the cost used by the cost-sensitive classifier introduced in Option 4 below.

4.11.3 Option 3: Build A Larger Tree and use Predicted Probabilities

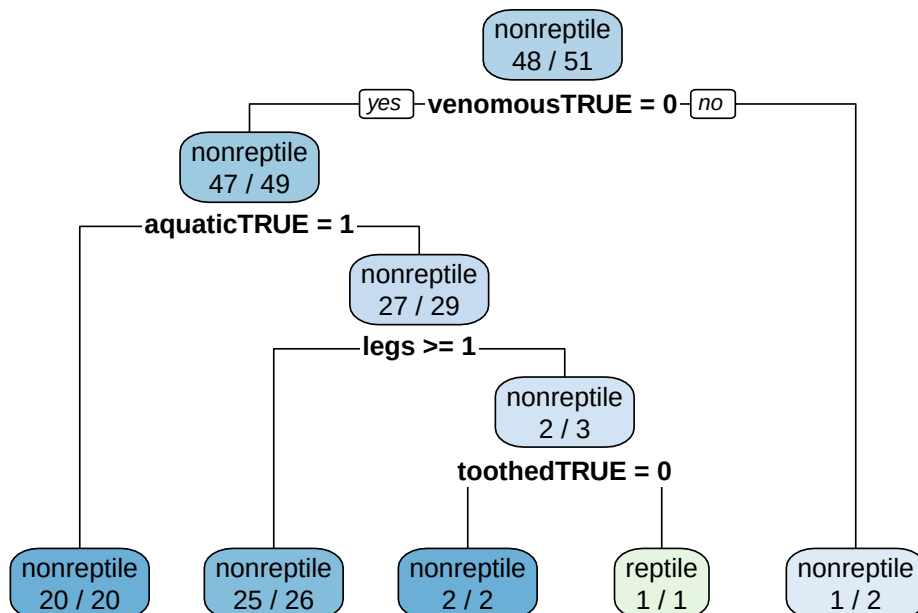
Increase complexity and require less data for splitting a node. Here I also use AUC (area under the ROC) as the tuning metric. You need to specify the two class summary function. Note that the tree still trying to improve accuracy on the data and not AUC! I also enable class probabilities since I want to predict probabilities later.

```
fit <- training_reptile |>
  train(type ~ .,
        data = _,
        method = "rpart",
        tuneLength = 10,
        trControl = trainControl(
          method = "cv",
          classProbs = TRUE, ## for predict with type="prob"
          summaryFunction=twoClassSummary), ## for ROC
```

```

metric = "ROC",
control = rpart.control(minsplit = 3))
## Warning in nominalTrainWorkflow(x = x, y = y, wts =
## weights, info = trainInfo, : There were missing values in
## resampled performance measures.
fit
## CART
##
## 51 samples
## 16 predictors
## 2 classes: 'nonreptile', 'reptile'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 46, 45, 46, 45, 47, 47, ...
## Resampling results:
##
##   ROC      Sens  Spec
## 0.4333 0.92 0
##
## Tuning parameter 'cp' was held constant at a value of 0
rpart.plot(fit$finalModel, extra = 2)

```



```

confusionMatrix(data = predict(fit, testing_reptile),
                  ref = testing_reptile$type, positive = "reptile")
## Confusion Matrix and Statistics
##
##              Reference
## Prediction  nonreptile reptile
## nonreptile      48      2
## reptile         0      0
##
##              Accuracy : 0.96
##              95% CI : (0.863, 0.995)
##      No Information Rate : 0.96
##      P-Value [Acc > NIR] : 0.677
##
##              Kappa : 0
##
##  Mcnemar's Test P-Value : 0.480
##
##              Sensitivity : 0.00
##              Specificity : 1.00
##      Pos Pred Value : NaN
##      Neg Pred Value : 0.96
##              Prevalence : 0.04
##      Detection Rate : 0.00
##      Detection Prevalence : 0.00
##      Balanced Accuracy : 0.50
##
##      'Positive' Class : reptile
##

```

Note: Accuracy is high, but it is close or below to the no-information rate!

4.11.3.1 Create A Biased Classifier

We can create a classifier which will detect more reptiles at the expense of misclassifying non-reptiles. This is equivalent to increasing the cost of misclassifying a reptile as a non-reptile. The usual rule is to predict in each node the majority class from the test data in the node. For a binary classification problem that means a probability of $>50\%$. In the following, we reduce this threshold to 1% or more. This means that if the new observation ends up in a leaf node with 1% or more reptiles from training then the observation will be classified as a reptile. The data set is small and this works better with more data.

```

prob <- predict(fit, testing_reptile, type = "prob")
tail(prob)
##      nonreptile reptile
## tuna      1.0000 0.00000
## vole      0.9615 0.03846
## wasp      0.5000 0.50000
## wolf      0.9615 0.03846
## worm      1.0000 0.00000
## wren      0.9615 0.03846
pred <- ifelse(prob[, "reptile"] >= 0.01, "reptile", "nonreptile") |>
  as.factor()

confusionMatrix(data = pred,
  ref = testing_reptile$type, positive = "reptile")
## Confusion Matrix and Statistics
##
##              Reference
## Prediction  nonreptile reptile
## nonreptile      13      0
## reptile         35      2
##
##              Accuracy : 0.3
##              95% CI : (0.179, 0.446)
##      No Information Rate : 0.96
##      P-Value [Acc > NIR] : 1
##
##              Kappa : 0.029
##
##  McNemar's Test P-Value : 9.08e-09
##
##              Sensitivity : 1.0000
##              Specificity : 0.2708
##      Pos Pred Value : 0.0541
##      Neg Pred Value : 1.0000
##      Prevalence : 0.0400
##      Detection Rate : 0.0400
##      Detection Prevalence : 0.7400
##      Balanced Accuracy : 0.6354
##
##      'Positive' Class : reptile
##

```

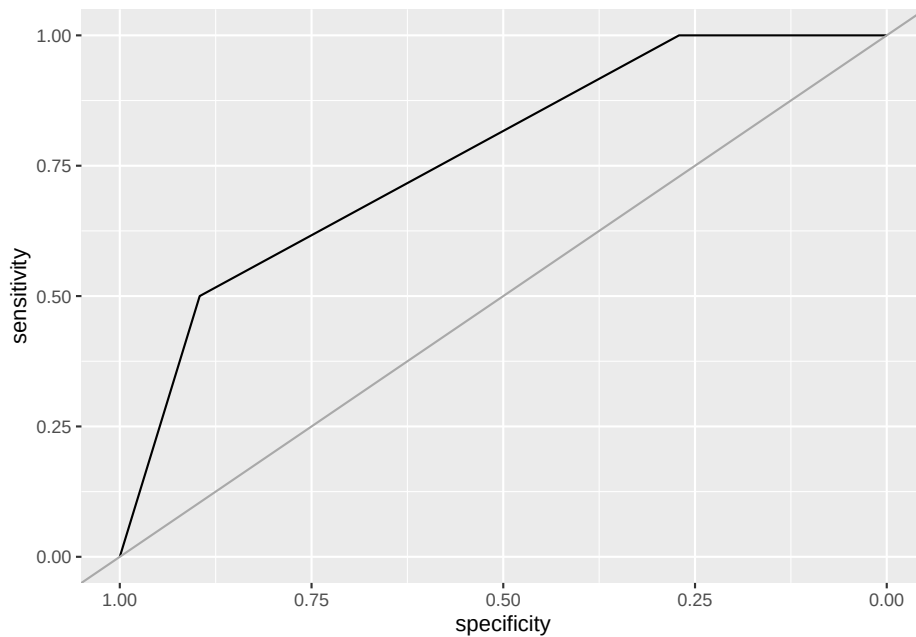
Note that accuracy goes down and is below the no information rate. However, both measures are based on the idea that all errors have the same cost. What is important is that we are now able to find more reptiles.

4.11.3.2 Plot the ROC Curve

Since we have a binary classification problem and a classifier that predicts a probability for an observation to be a reptile, we can also use a receiver operating characteristic (ROC)¹² curve. For the ROC curve all different cutoff thresholds for the probability are used and then connected with a line. The area under the curve represents a single number for how well the classifier works (the closer to one, the better).

```
library("pROC")
## Type 'citation("pROC")' for a citation.
##
## Attaching package: 'pROC'
## The following objects are masked from 'package:stats':
##
##      cov, smooth, var
r <- roc(testing_reptile$type == "reptile", prob[, "reptile"])
## Setting levels: control = FALSE, case = TRUE
## Setting direction: controls < cases
r
##
## Call:
## roc.default(response = testing_reptile$type == "reptile", predictor = prob[,      "reptile"])
##
## Data: prob[, "reptile"] in 48 controls (testing_reptile$type == "reptile" FALSE) < 2 cases (t
## Area under the curve: 0.766
ggroc(r) + geom_abline(intercept = 1, slope = 1, color = "darkgrey")
```

¹²https://en.wikipedia.org/wiki/Receiver_operating_characteristic



4.11.4 Option 4: Use a Cost-Sensitive Classifier

The implementation of CART in `rpart` can use a cost matrix for making splitting decisions (as parameter `loss`). The matrix has the form

```
TP FP
FN TN
```

where TP and TN have to be 0. We make FN very expensive with a cost of 100 compared to the cost for FP of 1.

```
cost <- matrix(c(
  0, 1,
  100, 0
), byrow = TRUE, nrow = 2)
cost
##      [,1] [,2]
## [1,]    0    1
## [2,]  100    0
fit <- training_reptile |>
  train(type ~ .,
        data = _,
        method = "rpart",
```

```

    parms = list(loss = cost),
    trControl = trainControl(method = "cv"))
## Warning in nominalTrainWorkflow(x = x, y = y, wts =
## weights, info = trainInfo, : There were missing values in
## resampled performance measures.

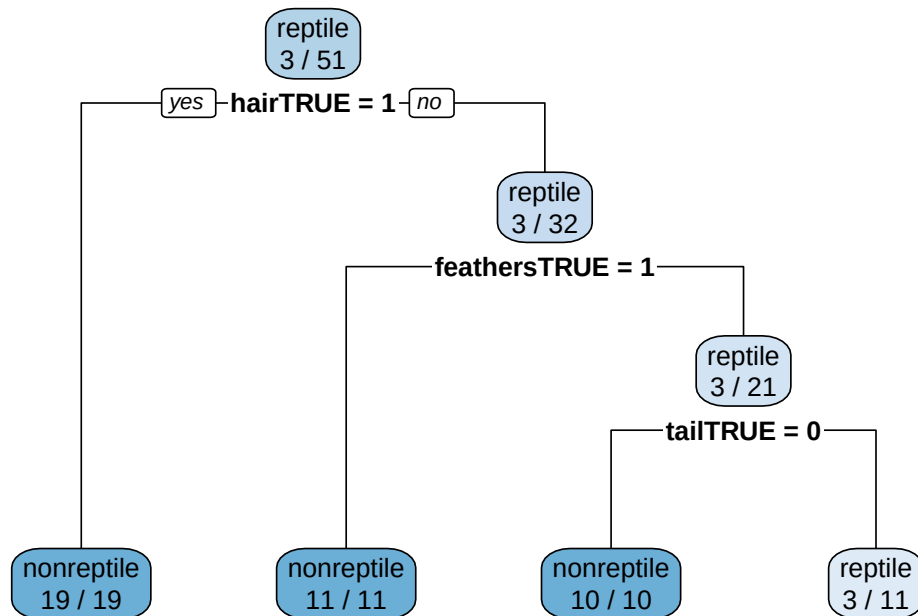
```

The warning “*There were missing values in resampled performance measures*” means that some folds did not contain any reptiles (because of the class imbalance) and thus the performance measures could not be calculated.

```

fit
## CART
##
## 51 samples
## 16 predictors
## 2 classes: 'nonreptile', 'reptile'
##
## No pre-processing
## Resampling: Cross-Validated (10 fold)
## Summary of sample sizes: 46, 46, 46, 46, 46, 45, ...
## Resampling results:
##
##   Accuracy   Kappa
## 0.5417      -0.0409
##
## Tuning parameter 'cp' was held constant at a value of 0
rpart.plot(fit$finalModel, extra = 2)

```



```

confusionMatrix(data = predict(fit, testing_reptile),
  ref = testing_reptile$type, positive = "reptile")
## Confusion Matrix and Statistics
##
##           Reference
## Prediction  nonreptile reptile
## nonreptile      39      0
## reptile         9      2
##
##           Accuracy : 0.82
##           95% CI : (0.686, 0.914)
##       No Information Rate : 0.96
##       P-Value [Acc > NIR] : 0.99998
##
##           Kappa : 0.257
##
##  Mcnemar's Test P-Value : 0.00766
##
##           Sensitivity : 1.000
##           Specificity : 0.812
##       Pos Pred Value : 0.182
##       Neg Pred Value : 1.000
##           Prevalence : 0.040
##       Detection Rate : 0.040
##       Detection Prevalence : 0.220

```

```
##      Balanced Accuracy : 0.906
##
##      'Positive' Class : reptile
##
```

The high cost for false negatives results in a classifier that does not miss any reptile.

Note: Using a cost-sensitive classifier is often the best option. Unfortunately, most classification algorithms (or their implementation) do not have the ability to consider misclassification cost. Cost-sensitivity can be added to ANNs by changing the optimized loss function from pure cross-entropy loss to a weighted version. Using a custom loss function is possible in most deep learning libraries.

4.12 Comparing Decision Boundaries of Popular Classification Techniques*

Classifiers create decision boundaries to discriminate between classes. Different classifiers are able to create different shapes of decision boundaries (e.g., some are strictly linear) and thus some classifiers may perform better for certain datasets. In this section, we visualize the decision boundaries found by several popular classification methods.

The following function defines a plot that adds the decision boundary (black lines) and the difference between the classification probability of the two best classes (color intensity; indifference is 0 shown as white) by evaluating the classifier at evenly spaced grid points. Note that low resolution will make evaluation faster but it also will make the decision boundary look like it has small steps even if it is a straight line.

```
library(tidyverse)
library(ggplot2)
library(scales)
library(caret)
```

```
decisionplot <- function(model, data, class_var,
  predict_type = c("class", "prob"), resolution = 3 * 72) {
  # resolution is set to 72 dpi for 3 inches wide images.

  y <- data |> pull(class_var)
  x <- data |> dplyr::select(-all_of(class_var))

  # resubstitution accuracy
```

```

prediction <- predict(model, x, type = predict_type[1])

# LDA returns a list
if(is.list(prediction)) prediction <- prediction$class

prediction <- factor(prediction, levels = levels(y))
cm <- confusionMatrix(data = prediction,
                      reference = y)
acc <- cm$overall["Accuracy"]

# evaluate model on a grid
r <- sapply(x[, 1:2], range, na.rm = TRUE)
xs <- seq(r[1,1], r[2,1], length.out = resolution)
ys <- seq(r[1,2], r[2,2], length.out = resolution)
g <- cbind(rep(xs, each = resolution), rep(ys,
                                           time = resolution))

colnames(g) <- colnames(r)
g <- as_tibble(g)

# guess how to get class labels from predict
# (unfortunately not very consistent between models)
cl <- predict(model, g, type = predict_type[1])

prob <- NULL
if(is.list(cl)) { # LDA returns a list
  prob <- cl$posterior
  cl <- cl$class
} else if (inherits(model, "svm"))
  prob <- attr(predict(model, g, probability = TRUE), "probabilities")
else
  if(!is.na(predict_type[2]))
    try(prob <- predict(model, g, type = predict_type[2]))

# We visualize the difference in probability/score between
# the winning class and the second best class. We only use
# probability if the classifier's predict function supports it.
delta_prob <- 1
if(!is.null(prob))

  try({
    if (any(rowSums(prob) != 1))
      prob <- cbind(prob, 1 - rowSums(prob))

    max_prob <- t(apply(prob, MARGIN = 1, sort, decreasing = TRUE))
    delta_prob <- max_prob[,1] - max_prob[,2]
  })

```

```

    }, silent = TRUE)

cl <- factor(cl, levels = levels(y))

g <- g |> add_column(prediction = cl,
                     delta_prob = delta_prob)

ggplot(g, mapping = aes(
  x = .data[[colnames(g)[1]]],
  y = .data[[colnames(g)[2]]]) +
  geom_raster(mapping = aes(fill = prediction,
                           alpha = delta_prob)) +
  geom_contour(mapping = aes(z = as.numeric(prediction)),
               bins = length(levels(cl)),
               linewidth = .5,
               color = "black") +
  geom_point(data = data, mapping = aes(
    x = .data[[colnames(data)[1]]],
    y = .data[[colnames(data)[2]]],
    shape = .data[[class_var]],
    alpha = .7) +
  scale_alpha_continuous(range = c(0,1),
                        limits = c(0,1)) +
  labs(subtitle = paste("Training accuracy:", round(acc, 2)))
}

```

4.12.1 Iris Dataset

For easier visualization, we use two dimensions of the Iris dataset.

```

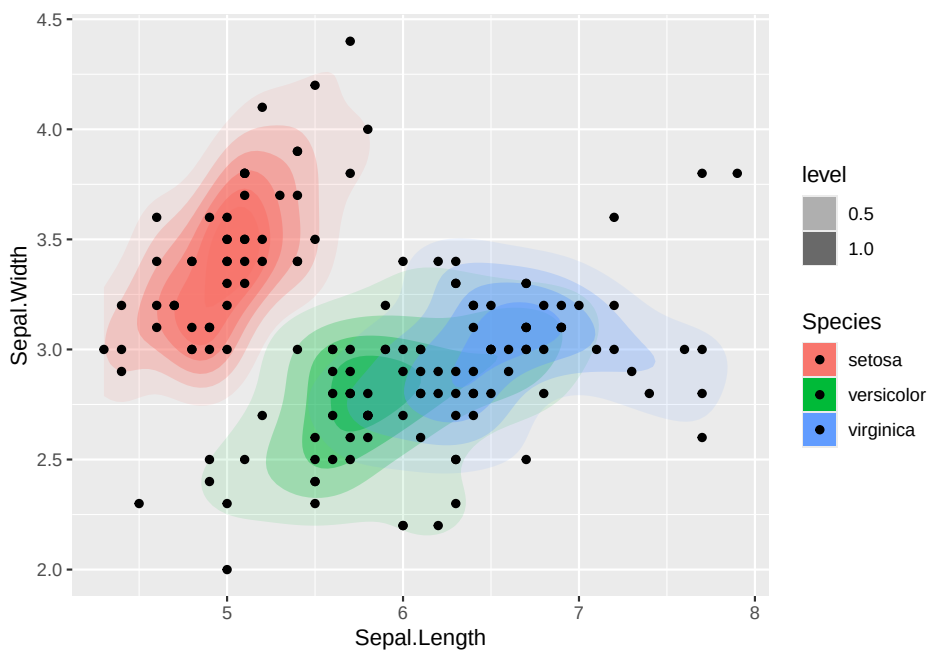
set.seed(1000)
data(iris)
iris <- as_tibble(iris)

x <- iris |> dplyr::select(Sepal.Length, Sepal.Width, Species)
# Note: package MASS overwrites the select function.

x
## # A tibble: 150 x 3
##   Sepal.Length Sepal.Width Species
##         <dbl>         <dbl> <fct>
## 1         5.1         3.5 setosa
## 2         4.9         3   setosa
## 3         4.7         3.2 setosa

```

```
## 4      4.6      3.1 setosa
## 5      5      3.6 setosa
## 6      5.4      3.9 setosa
## 7      4.6      3.4 setosa
## 8      5      3.4 setosa
## 9      4.4      2.9 setosa
## 10     4.9      3.1 setosa
## # i 140 more rows
ggplot(x, aes(x = Sepal.Length,
              y = Sepal.Width,
              fill = Species)) +
  stat_density_2d(geom = "polygon",
                 aes(alpha = after_stat(level))) +
  geom_point()
```



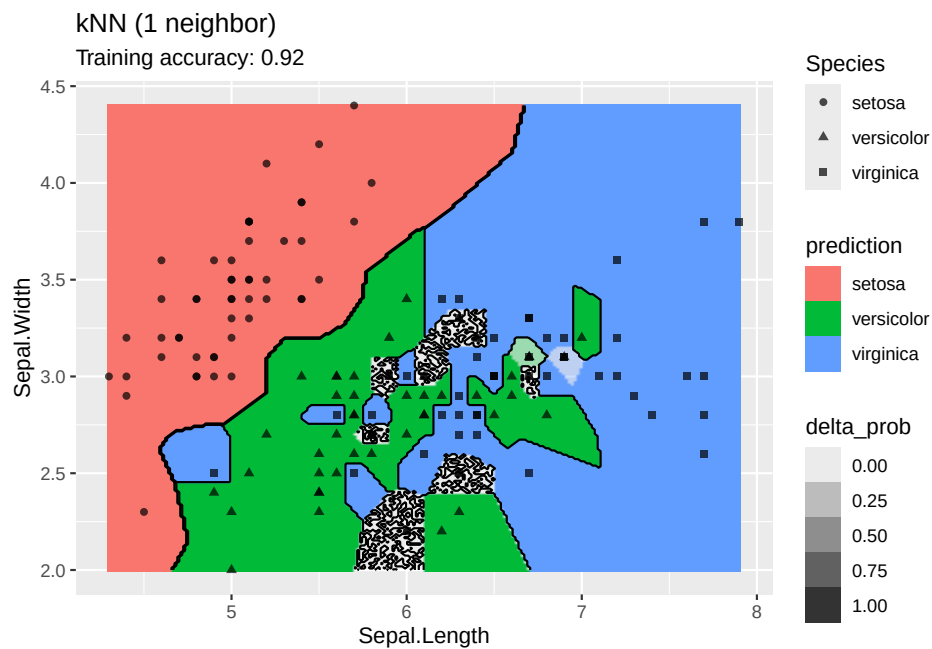
This is the original data. Color is used to show the density. Note that there is some overplotting with several points in the same position. You could use `geom_jitter()` instead of `geom_point()`.

4.12.1.1 Nearest Neighbor Classifier

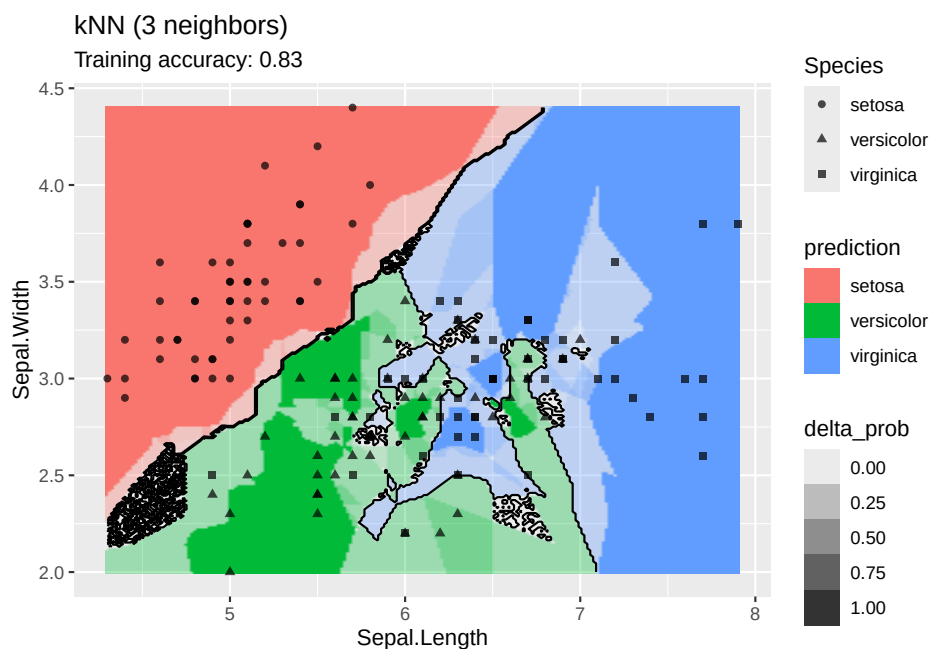
We try several values for k .

4.12. COMPARING DECISION BOUNDARIES OF POPULAR CLASSIFICATION TECHNIQUES*149

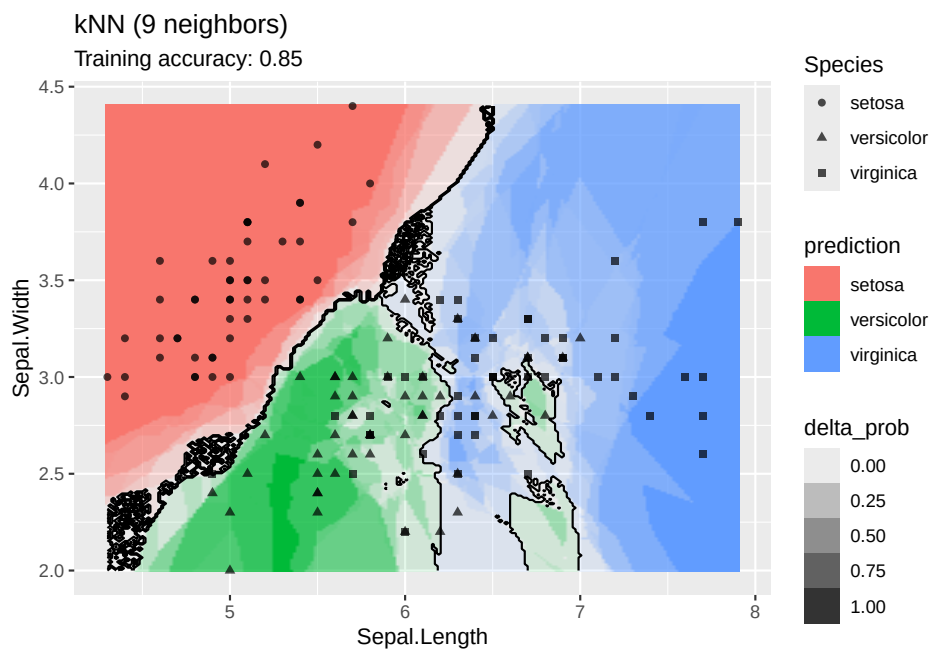
```
model <- x |> caret::knn3(Species ~ ., data = _, k = 1)
decisionplot(model, x, class_var = "Species") +
  labs(title = "kNN (1 neighbor)")
```



```
model <- x |> caret::knn3(Species ~ ., data = _, k = 3)
decisionplot(model, x, class_var = "Species") +
  labs(title = "kNN (3 neighbors)")
```



```
model <- x |> caret::knn3(Species ~ ., data = _, k = 9)
decisionplot(model, x, class_var = "Species") +
  labs(title = "kNN (9 neighbors)")
```



4.12. COMPARING DECISION BOUNDARIES OF POPULAR CLASSIFICATION TECHNIQUES*151

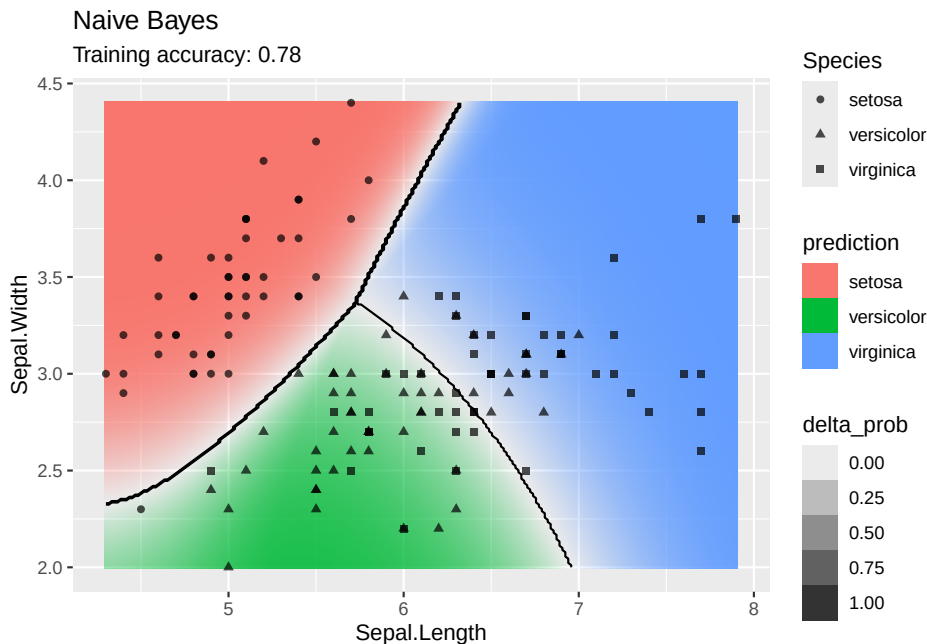
Increasing k smooths the decision boundary. At $k = 1$, we see white areas around points where flowers of two classes are in the same spot. Here, the algorithm randomly chooses a class during prediction resulting in the meandering decision boundary. The predictions in that area are not stable and every time we ask for a class, we may get a different class.

Note: The crazy lines in white areas are an artifact of the visualization. Here the classifier randomly selects a class.

4.12.1.2 Naive Bayes Classifier

Use a Gaussian naive Bayes classifier.

```
model <- x |> e1071::naiveBayes(Species ~ ., data = _)
decisionplot(model, x, class_var = "Species",
             predict_type = c("class", "raw")) +
  labs(title = "Naive Bayes")
```

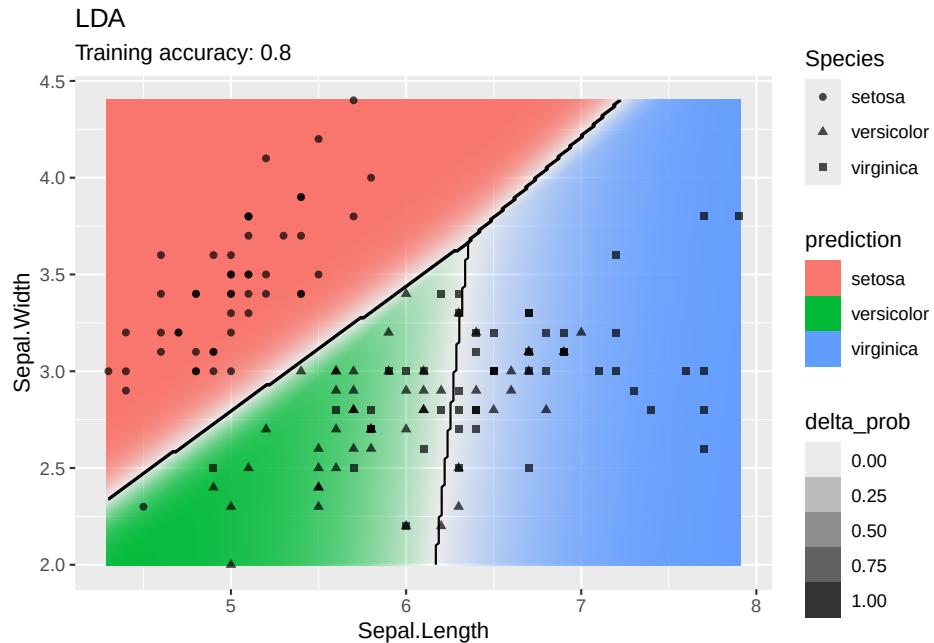


The GBN finds a good model with the advantage that no hyperparameters are needed.

4.12.1.3 Linear Discriminant Analysis

LDA finds linear decision boundaries.

```
model <- x |> MASS::lda(Species ~ ., data = _)
decisionplot(model, x, class_var = "Species") +
  labs(title = "LDA")
```



Linear decision boundaries work for this dataset so LDA works well.

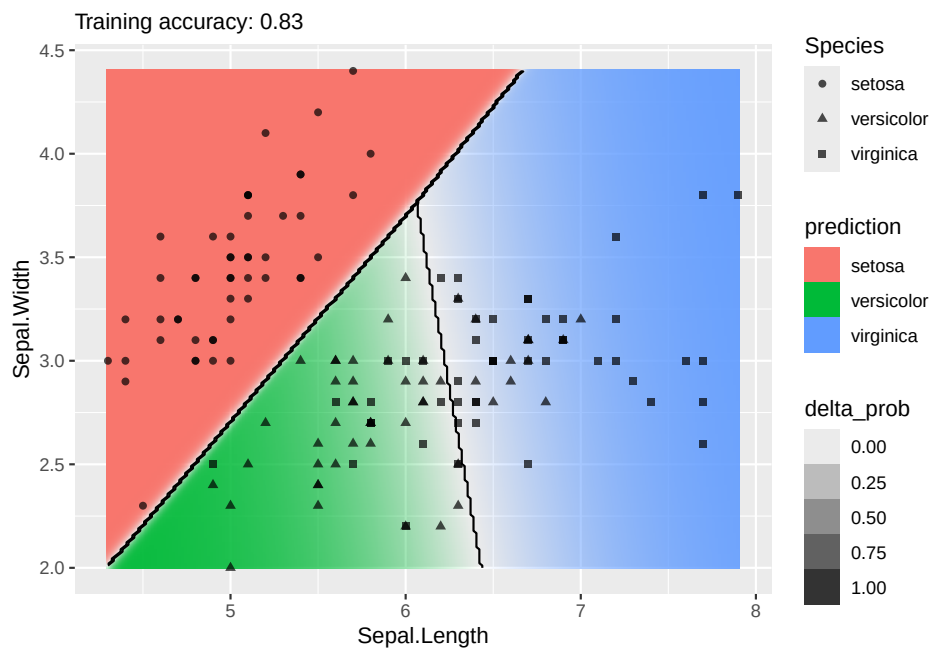
4.12.1.4 Multinomial Logistic Regression

Multinomial logistic regression is an extension of logistic regression to problems with more than two classes.

```
model <- x |> nnet::multinom(Species ~ ., data = _)
## # weights: 12 (6 variable)
## initial value 164.791843
## iter 10 value 62.715967
## iter 20 value 59.808291
## iter 30 value 55.445984
## iter 40 value 55.375704
## iter 50 value 55.346472
## iter 60 value 55.301707
## iter 70 value 55.253532
## iter 80 value 55.243230
## iter 90 value 55.230241
## iter 100 value 55.212479
```

4.12. COMPARING DECISION BOUNDARIES OF POPULAR CLASSIFICATION TECHNIQUES*153

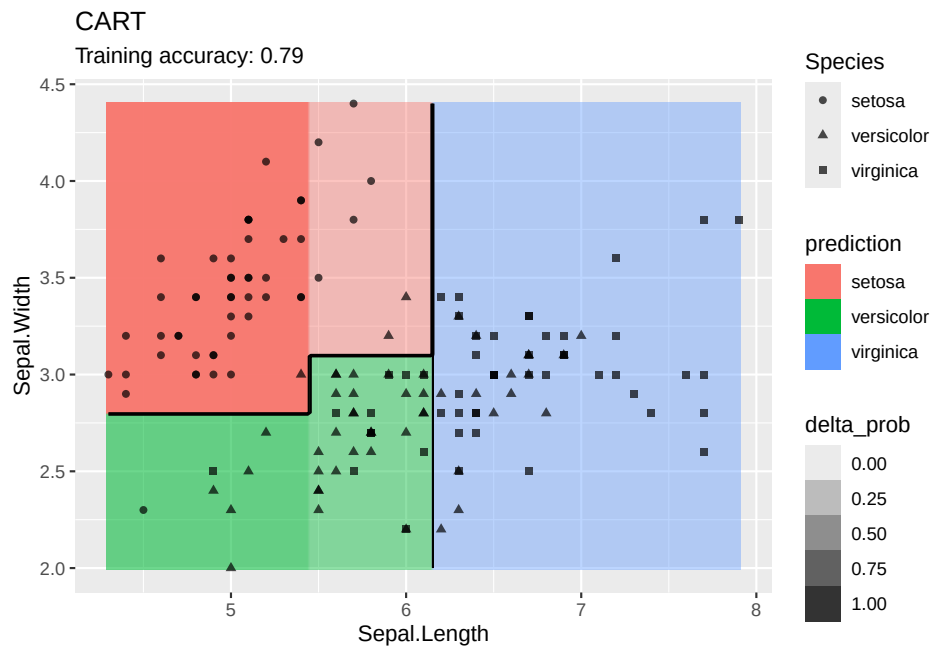
```
## final value 55.212479
## stopped after 100 iterations
decisionplot(model, x, class_var = "Species") +
  labs(titel = "Multinomial Logistic Regression")
## Ignoring unknown labels:
## * titel : "Multinomial Logistic Regression"
```



4.12.1.5 Decision Trees

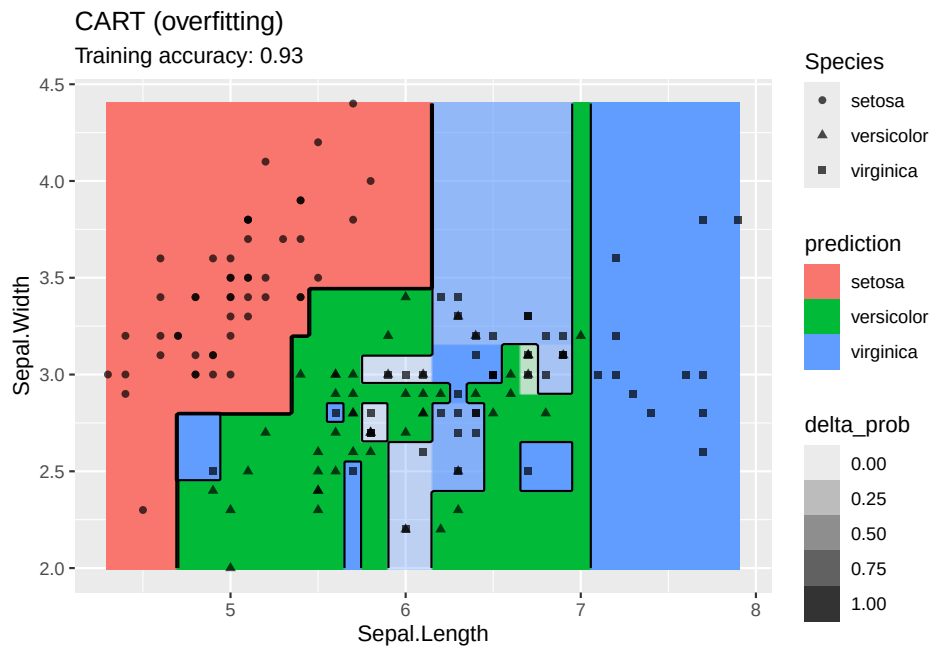
Compare different types of decision trees.

```
model <- x |> rpart::rpart(Species ~ ., data = _)
decisionplot(model, x, class_var = "Species") +
  labs(title = "CART")
```

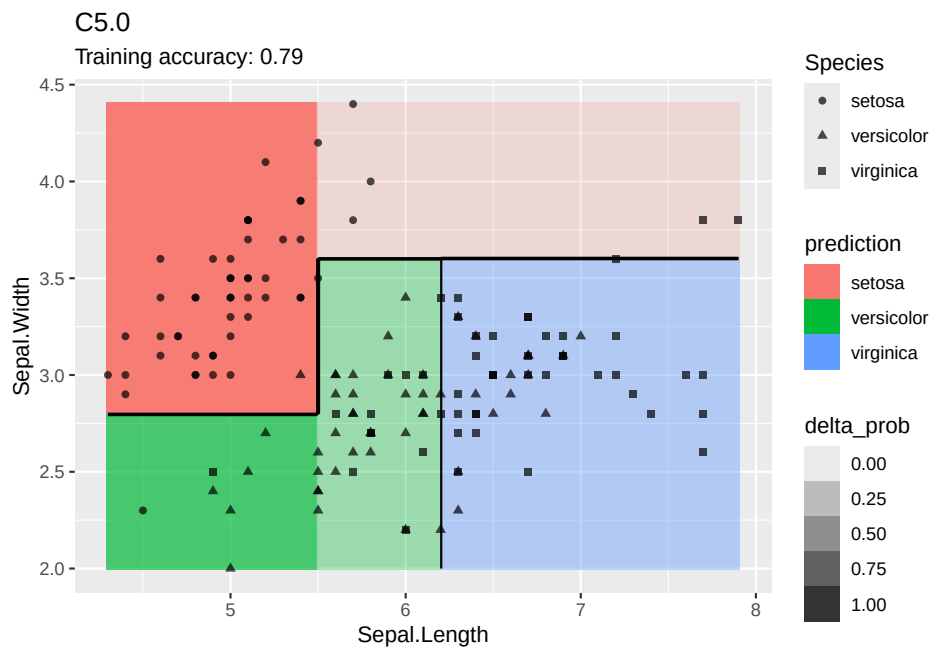


```
model <- x |> rpart::rpart(Species ~ ., data = _,
  control = rpart::rpart.control(cp = 0.001, minsplit = 1))
decisionplot(model, x, class_var = "Species") +
  labs(title = "CART (overfitting)")
```

4.12. COMPARING DECISION BOUNDARIES OF POPULAR CLASSIFICATION TECHNIQUES*155



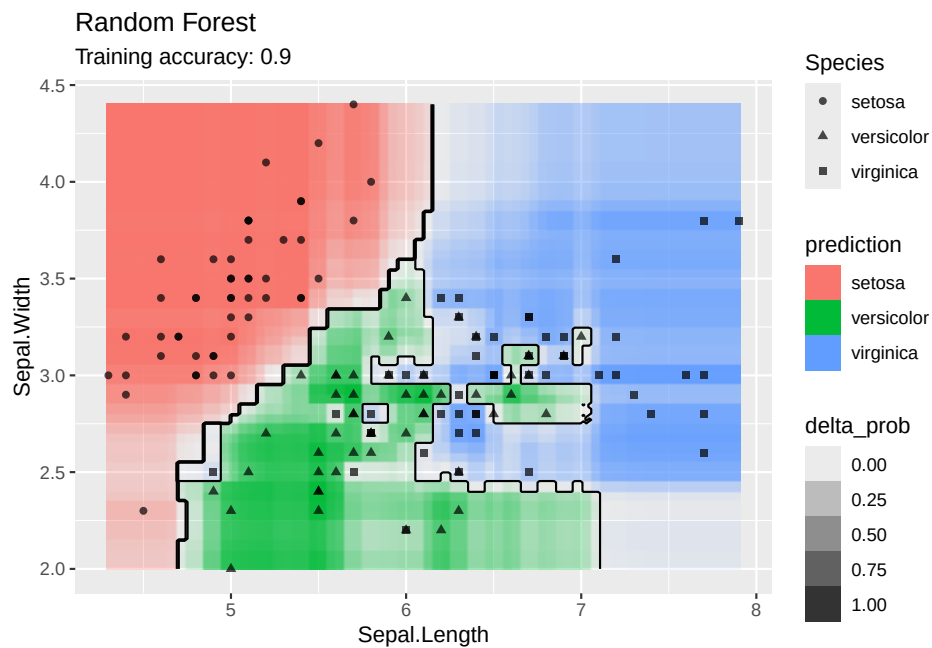
```
model <- x |> C50::C5.0(Species ~ ., data = _)
decisionplot(model, x, class_var = "Species") +
  labs(title = "C5.0")
```



4.12.1.6 Ensemble: Random Forest

Use an ensemble method.

```
model <- x |> randomForest::randomForest(Species ~ ., data = _)
decisionplot(model, x, class_var = "Species") +
  labs(title = "Random Forest")
```



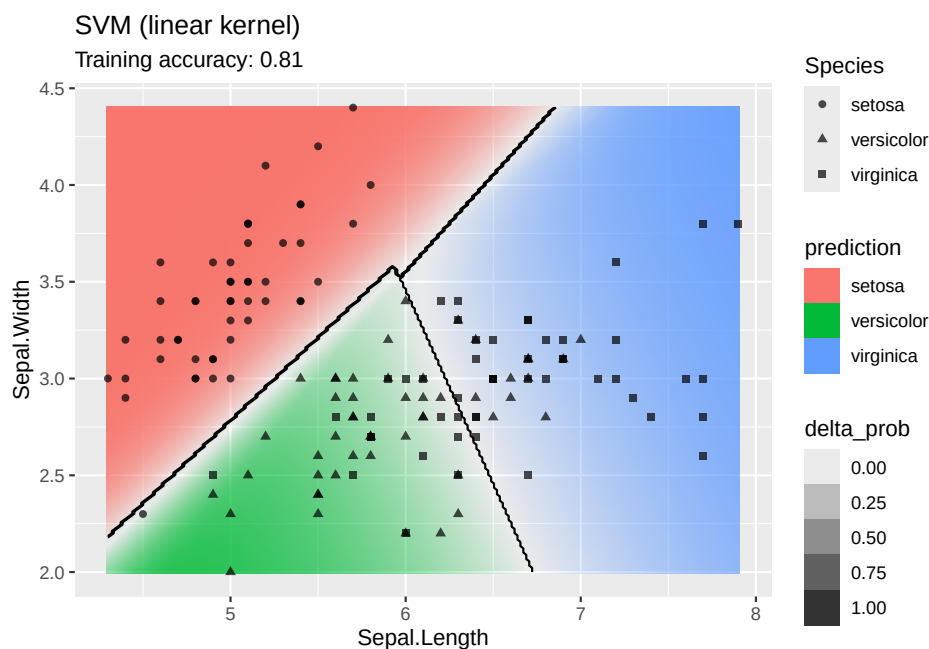
For the default settings for Random forest, the model seems to overfit the training data. More data would probably alleviate this issue.

4.12.1.7 Support Vector Machine

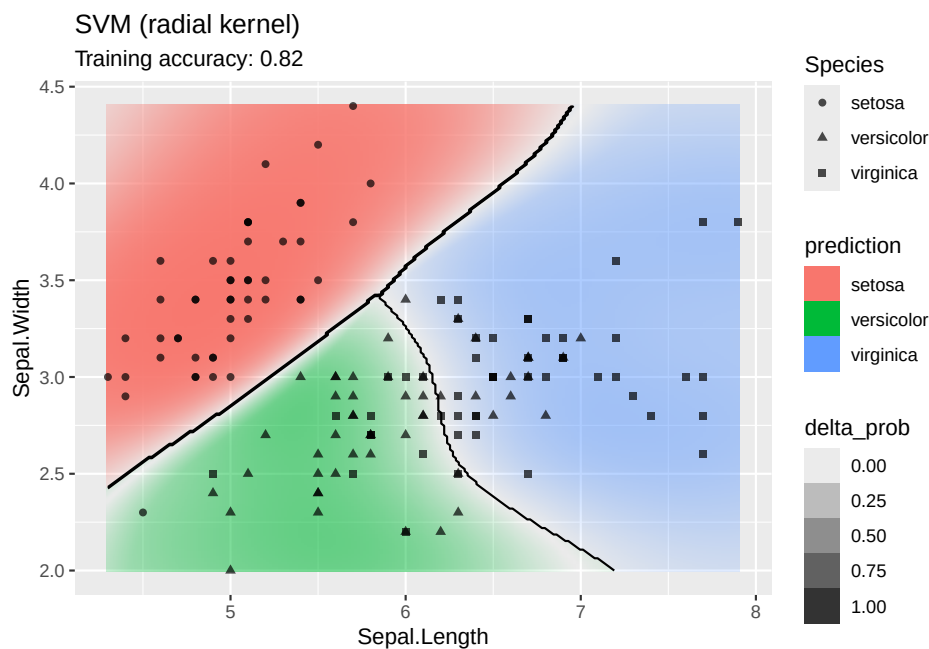
Compare SVMs with different kernel functions.

```
model <- x |> e1071::svm(Species ~ ., data = _,
  kernel = "linear", probability = TRUE)
decisionplot(model, x, class_var = "Species") +
  labs(title = "SVM (linear kernel)")
```


4.12. COMPARING DECISION BOUNDARIES OF POPULAR CLASSIFICATION TECHNIQUES*157



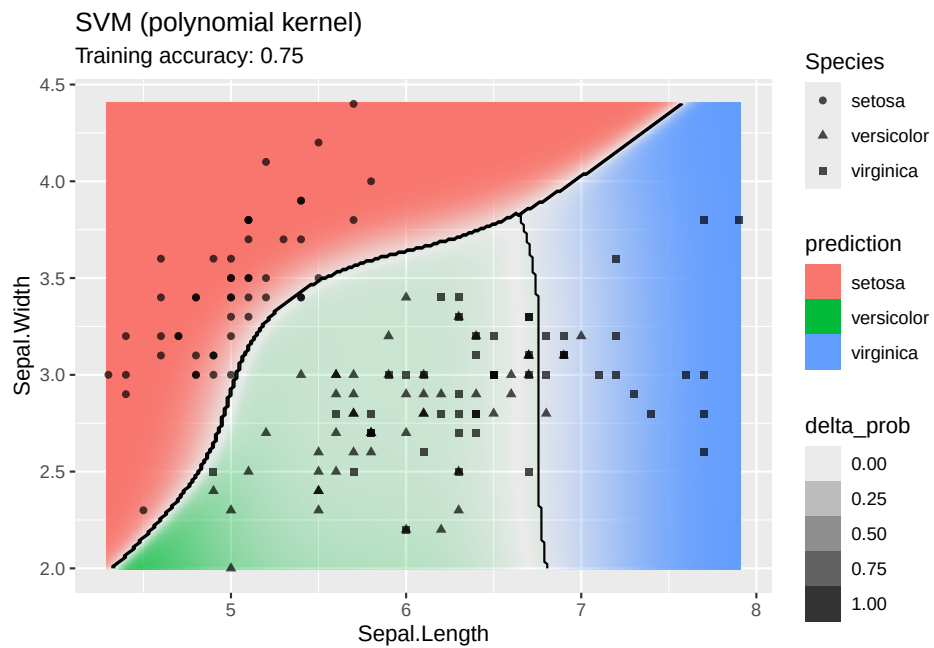
```
model <- x |> e1071::svm(Species ~ ., data = _,
                        kernel = "radial", probability = TRUE)
decisionplot(model, x, class_var = "Species") +
  labs(title = "SVM (radial kernel)")
```



```

model <- x |> e1071::svm(Species ~ ., data = _,
                        kernel = "polynomial", probability = TRUE)
decisionplot(model, x, class_var = "Species") +
  labs(title = "SVM (polynomial kernel)")

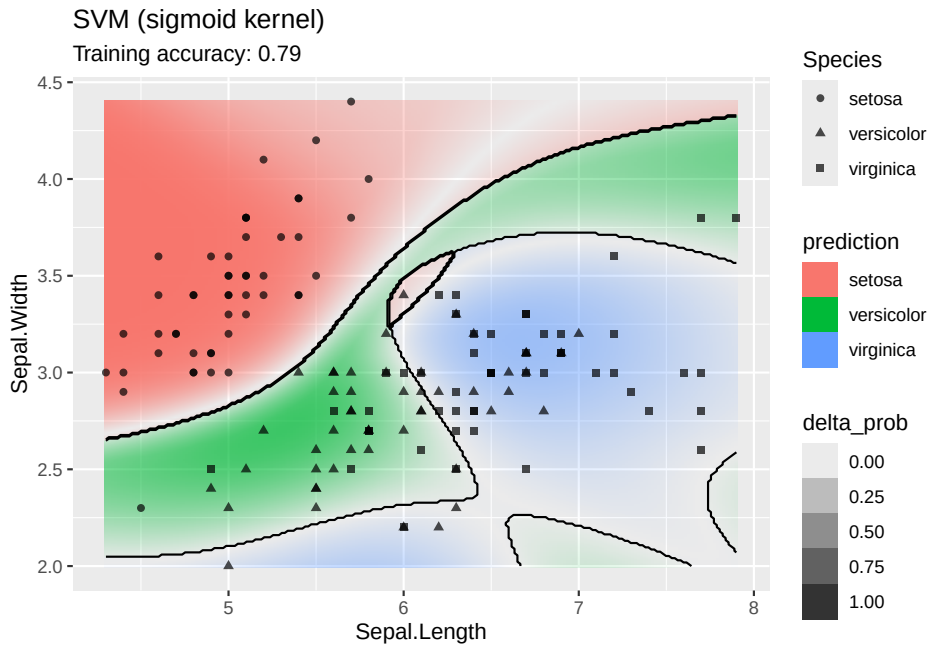
```



```

model <- x |> e1071::svm(Species ~ ., data = _,
                        kernel = "sigmoid", probability = TRUE)
decisionplot(model, x, class_var = "Species") +
  labs(title = "SVM (sigmoid kernel)")

```



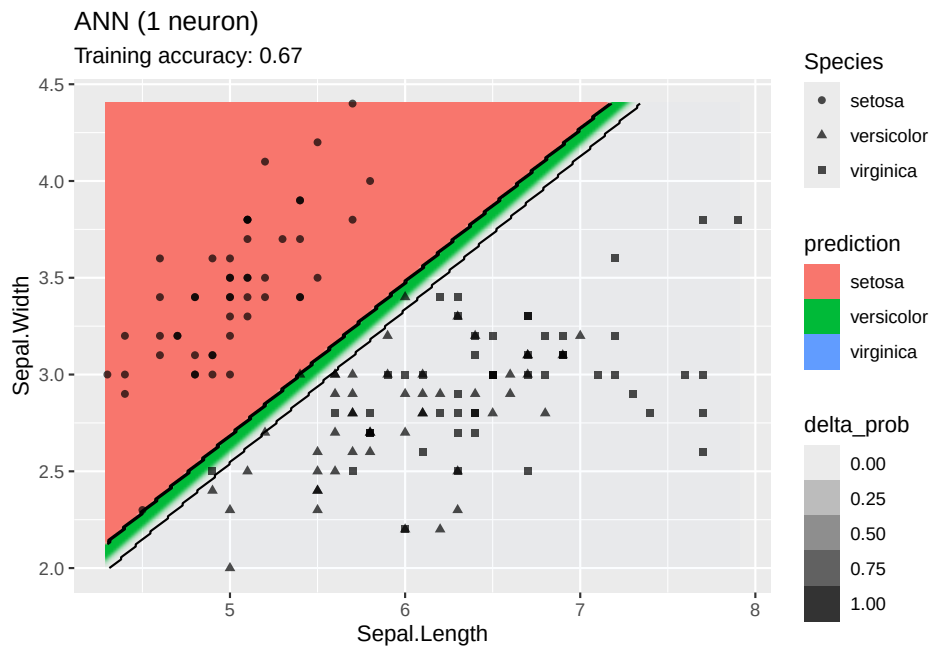
The linear SVM (without a kernel) produces straight lines and works well on the iris data. Most kernels do also well, only the sigmoid kernel seems to find a very strange decision boundary which indicates that the data does not have a linear decision boundary in the projected space.

4.12.1.8 Single Layer Feed-forward Neural Networks

Use a simple network with one hidden layer. We will try a different number of neurons for the hidden layer.

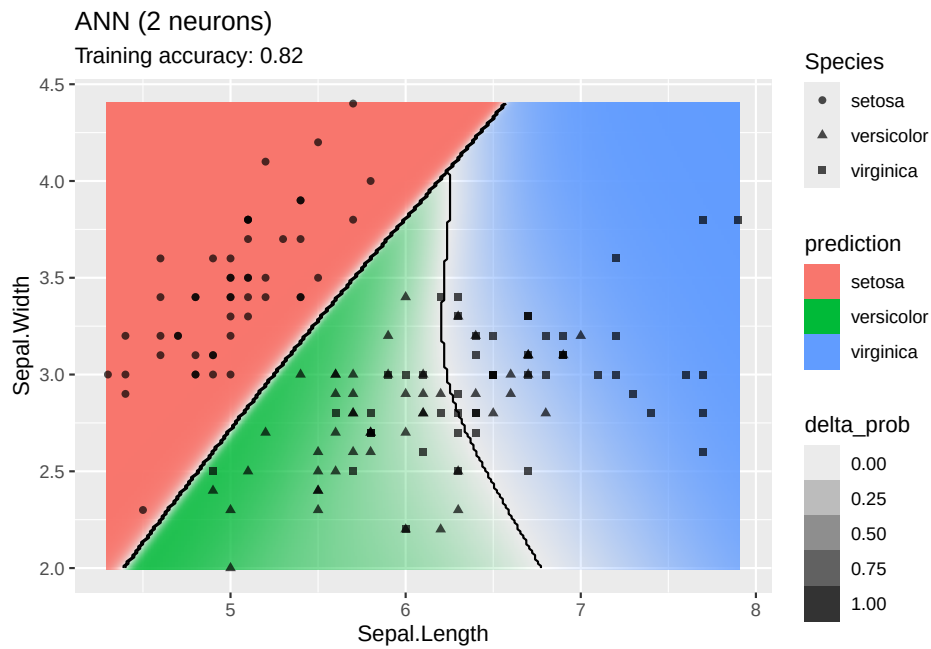
```
set.seed(1234)

model <- x |> nnet::nnet(Species ~ ., data = _,
                        size = 1, trace = FALSE)
decisionplot(model, x, class_var = "Species",
             predict_type = c("class", "raw")) +
  labs(title = "ANN (1 neuron)")
```

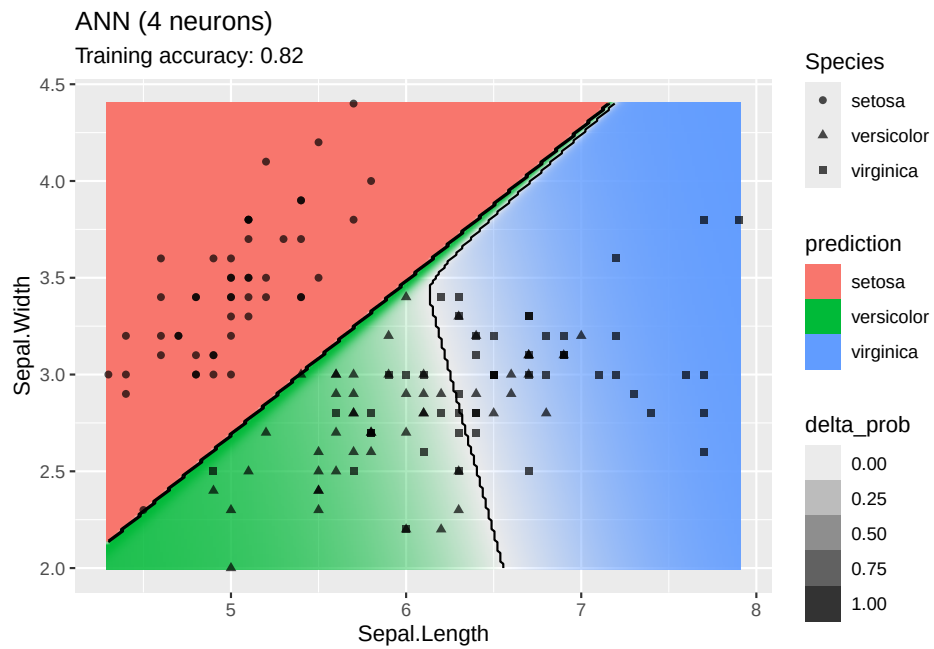


```
model <- x |> nnet::nnet(Species ~ ., data = _,
                        size = 2, trace = FALSE)
decisionplot(model, x, class_var = "Species",
             predict_type = c("class", "raw")) +
  labs(title = "ANN (2 neurons)")
```

4.12. COMPARING DECISION BOUNDARIES OF POPULAR CLASSIFICATION TECHNIQUES*161

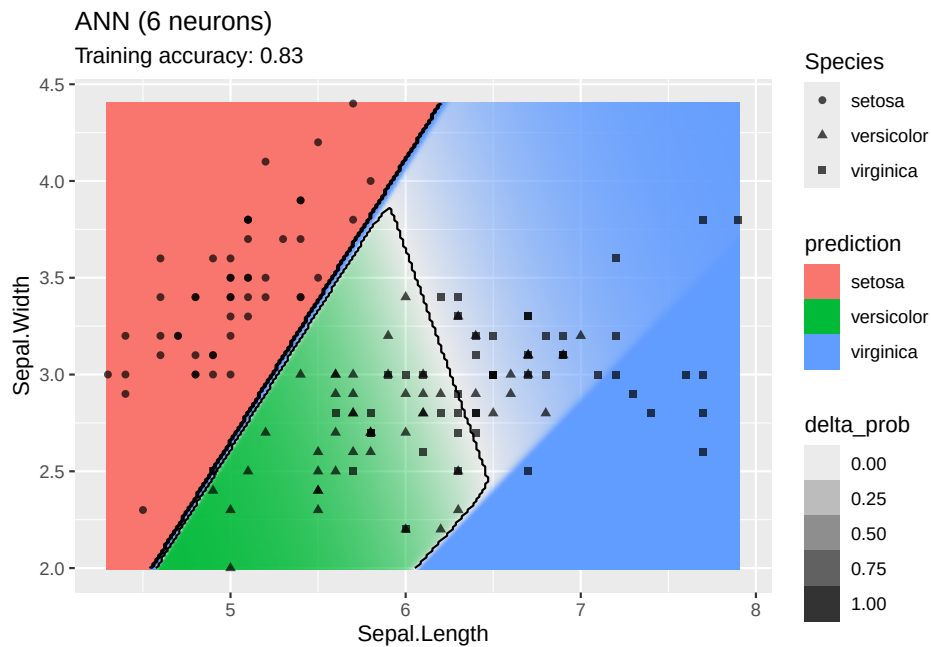


```
model <- x |> nnet::nnet(Species ~ ., data = _,
                        size = 4, trace = FALSE)
decisionplot(model, x, class_var = "Species",
             predict_type = c("class", "raw")) +
  labs(title = "ANN (4 neurons)")
```

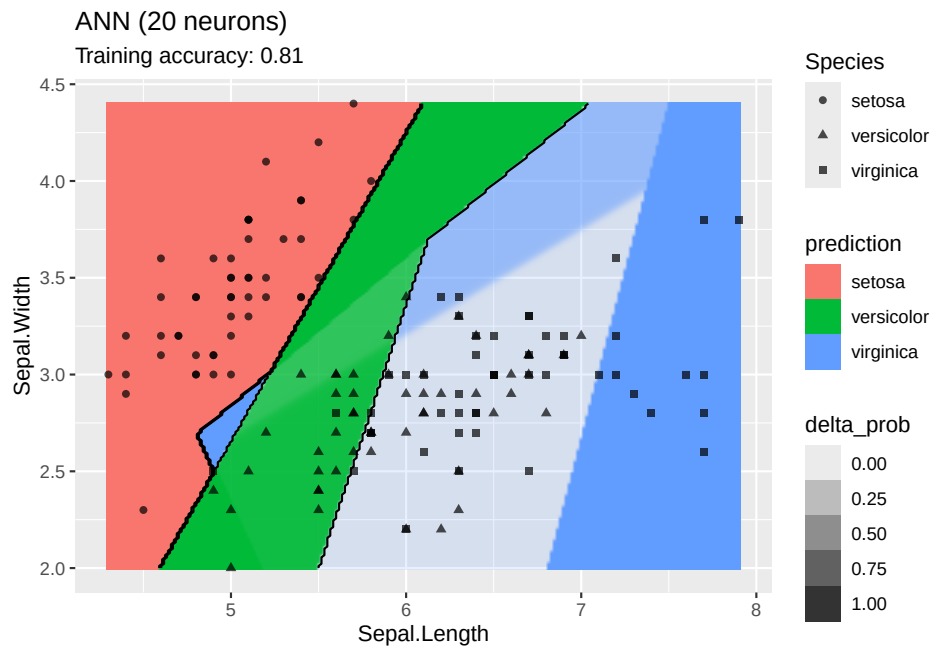


```
model <- x |> nnet::nnet(Species ~ ., data = _,
                        size = 6, trace = FALSE)
decisionplot(model, x, class_var = "Species",
             predict_type = c("class", "raw")) +
  labs(title = "ANN (6 neurons)")
```

4.12. COMPARING DECISION BOUNDARIES OF POPULAR CLASSIFICATION TECHNIQUES*163

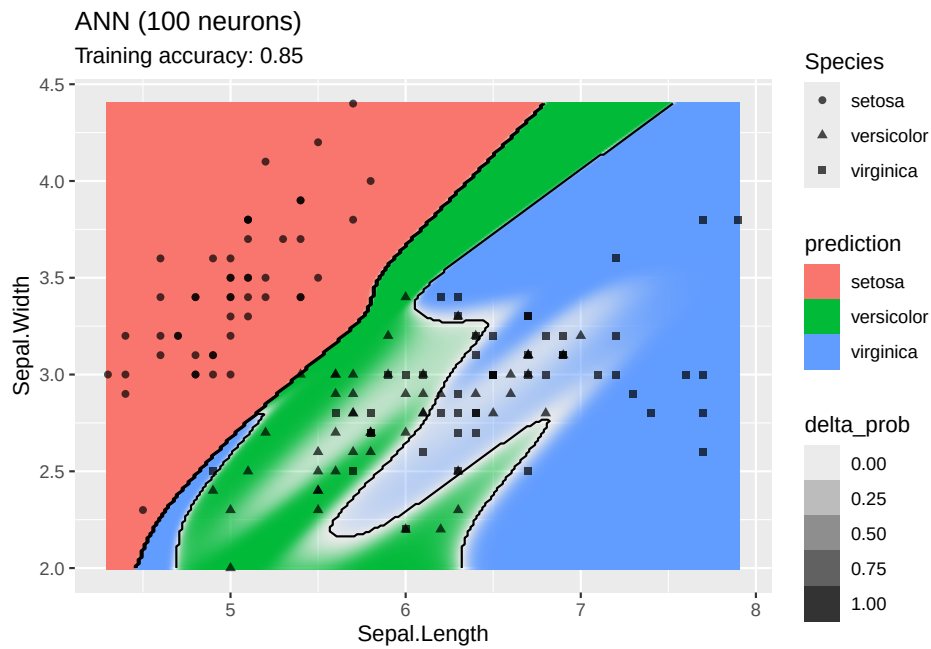


```
model <- x |> nnet::nnet(Species ~ ., data = _,
                        size = 20, trace = FALSE)
decisionplot(model, x, class_var = "Species",
             predict_type = c("class", "raw")) +
  labs(title = "ANN (20 neurons)")
```

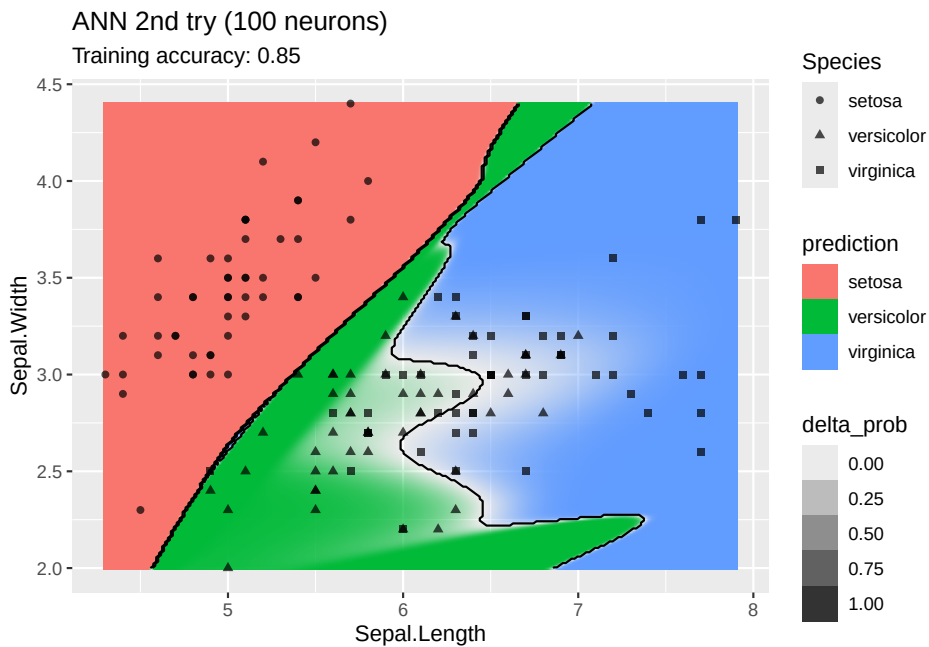


```
model <- x |> nnet::nnet(Species ~ ., data = _,
                        size = 100, trace = FALSE)
decisionplot(model, x, class_var = "Species",
             predict_type = c("class", "raw")) +
  labs(title = "ANN (100 neurons)")
```


4.12. COMPARING DECISION BOUNDARIES OF POPULAR CLASSIFICATION TECHNIQUES*165



```
model <- x |> nnet::nnet(Species ~ ., data = _,
                        size = 100, trace = FALSE)
decisionplot(model, x, class_var = "Species",
             predict_type = c("class", "raw")) +
  labs(title = "ANN 2nd try (100 neurons)")
```



For this simple data set, 2 neurons produce the best classifier. The model starts to overfit with 6 or more neurons. I ran the ANN twice with 100 neurons, the two different decision boundaries indicate that the variability of the ANN with so many neurons is high.

4.12.2 Circle Dataset

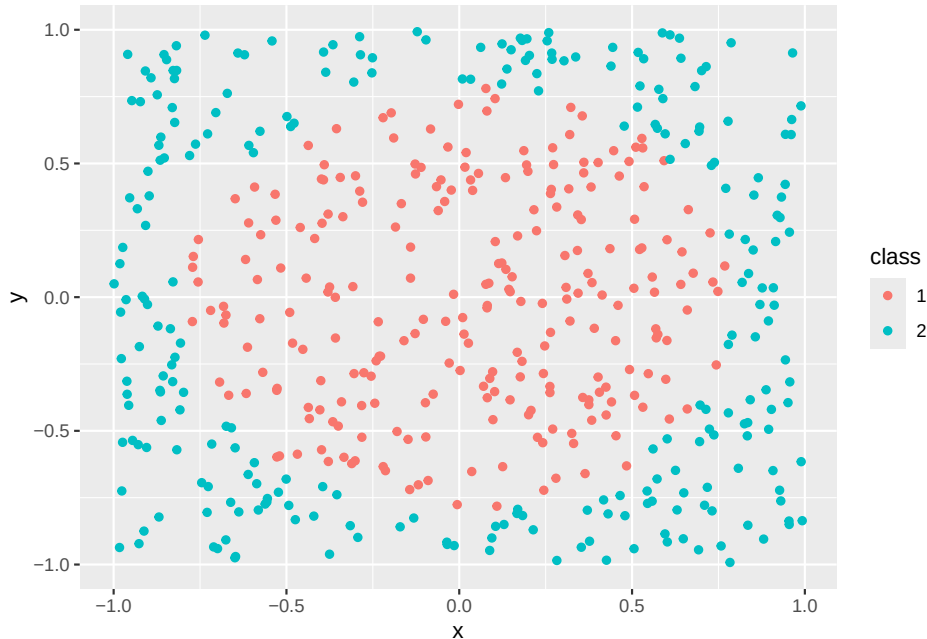
```
set.seed(1000)

x <- mlbench::mlbench.circle(500)

# You can also experiment with the following datasets.
#x <- mlbench::mlbench.cassini(500)
#x <- mlbench::mlbench.spirals(500, sd = .1)
#x <- mlbench::mlbench.smiley(500)

x <- cbind(as.data.frame(x$x), factor(x$classes))
colnames(x) <- c("x", "y", "class")
x <- as_tibble(x)
x
## # A tibble: 500 x 3
##       x       y class
##   <dbl> <dbl> <fct>
```

```
## 1 -0.344 0.448 1
## 2 0.518 0.915 2
## 3 -0.772 -0.0913 1
## 4 0.382 0.412 1
## 5 0.0328 0.438 1
## 6 -0.865 -0.354 2
## 7 0.477 0.640 2
## 8 0.167 -0.809 2
## 9 -0.568 -0.281 1
## 10 -0.488 0.638 2
## # i 490 more rows
ggplot(x, aes(x = x, y = y, color = class)) +
  geom_point()
```

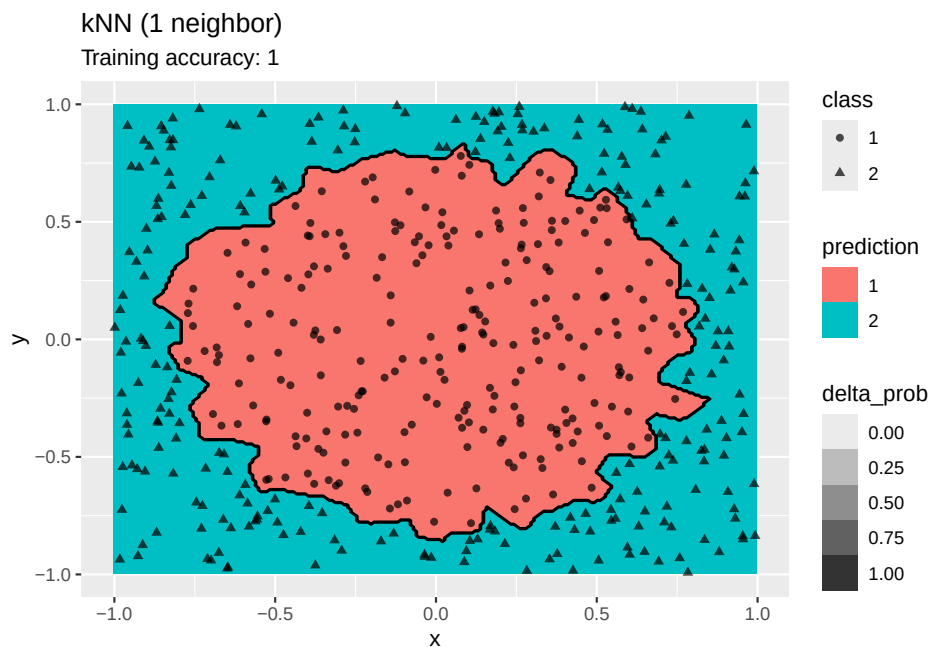


This dataset is challenging for some classification algorithms since the optimal decision boundary is a circle around the class in the center.

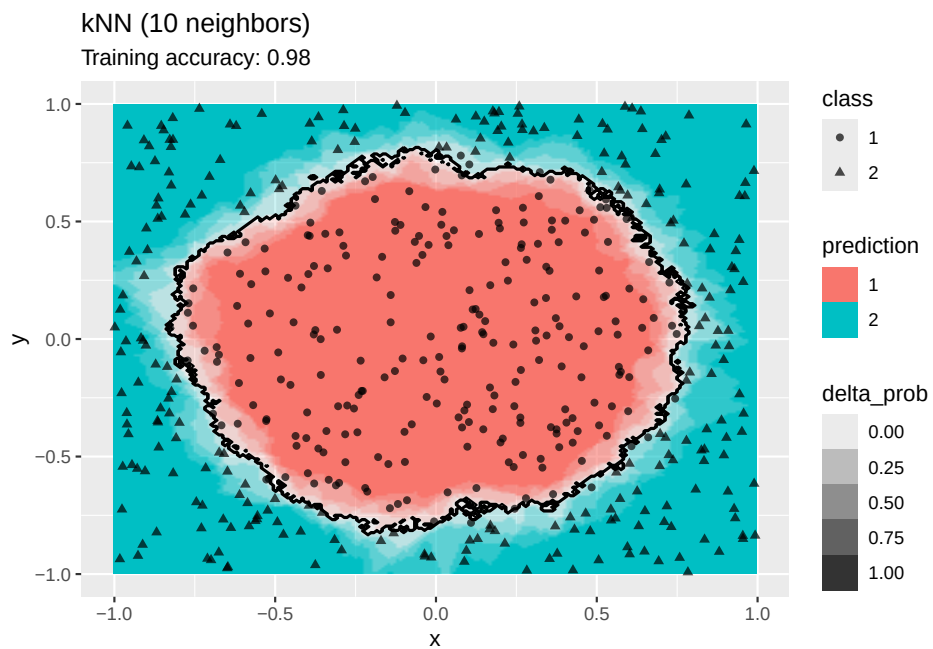
4.12.2.1 Nearest Neighbor Classifier

Compare kNN classifiers with different values for k .

```
model <- x |> caret::knn3(class ~ ., data = _, k = 1)
decisionplot(model, x, class_var = "class") +
  labs(title = "kNN (1 neighbor)")
```



```
model <- x |> caret::knn3(class ~ ., data = _, k = 10)
decisionplot(model, x, class_var = "class") +
  labs(title = "kNN (10 neighbors)")
```

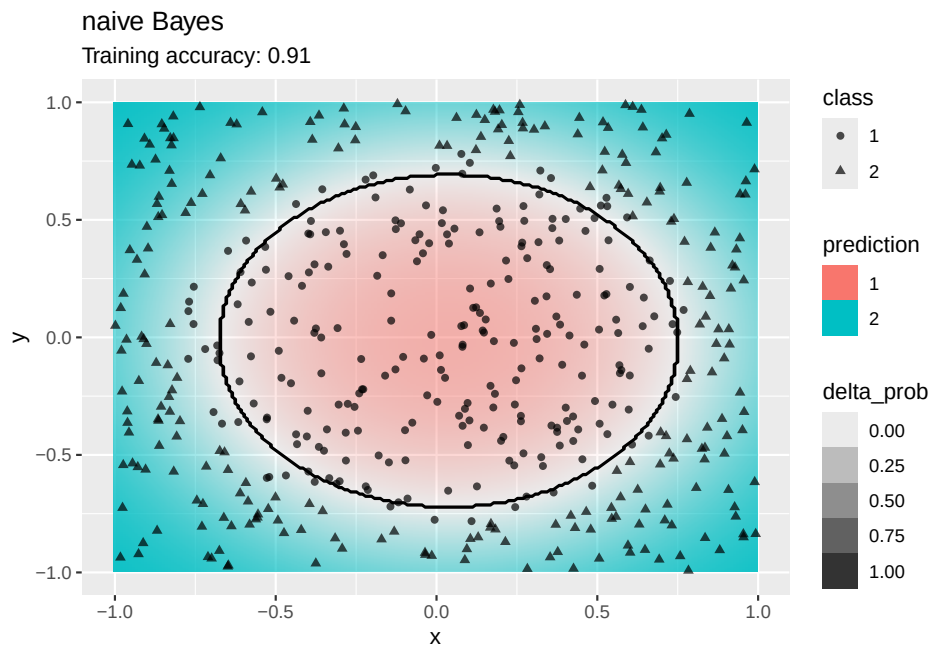


k-Nearest does not find a smooth decision boundary, but tends to overfit the training data at low values for k .

4.12.2.2 Naive Bayes Classifier

The Gaussian naive Bayes classifier works very well on the data.

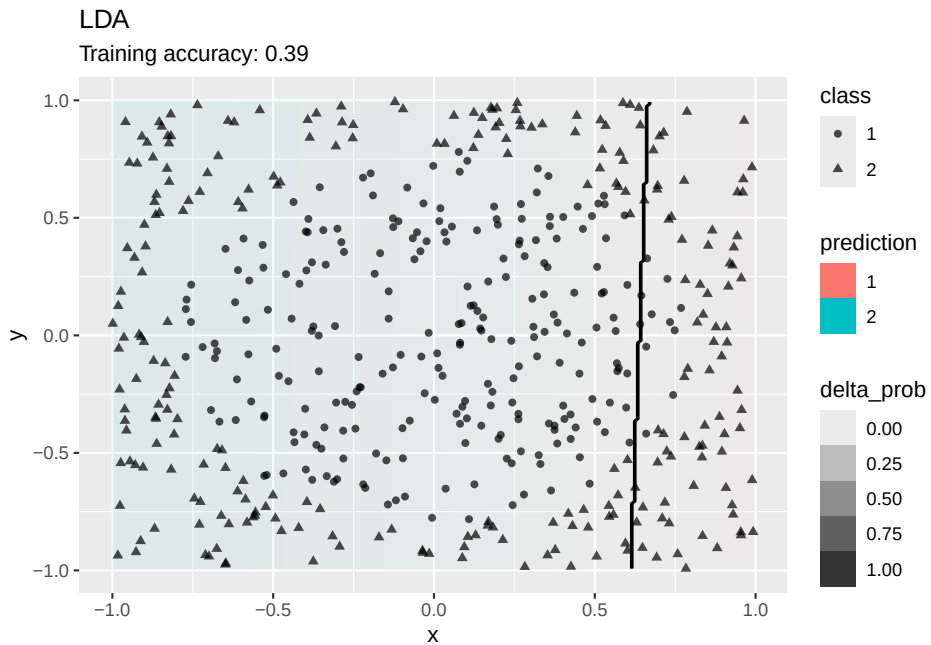
```
model <- x |> e1071::naiveBayes(class ~ ., data = _)
decisionplot(model, x, class_var = "class",
  predict_type = c("class", "raw")) +
  labs(title = "naive Bayes")
```



4.12.2.3 Linear Discriminant Analysis

LDA cannot find a good model since the true decision boundary is not linear.

```
model <- x |> MASS::lda(class ~ ., data = _)
decisionplot(model, x, class_var = "class") + labs(title = "LDA")
```

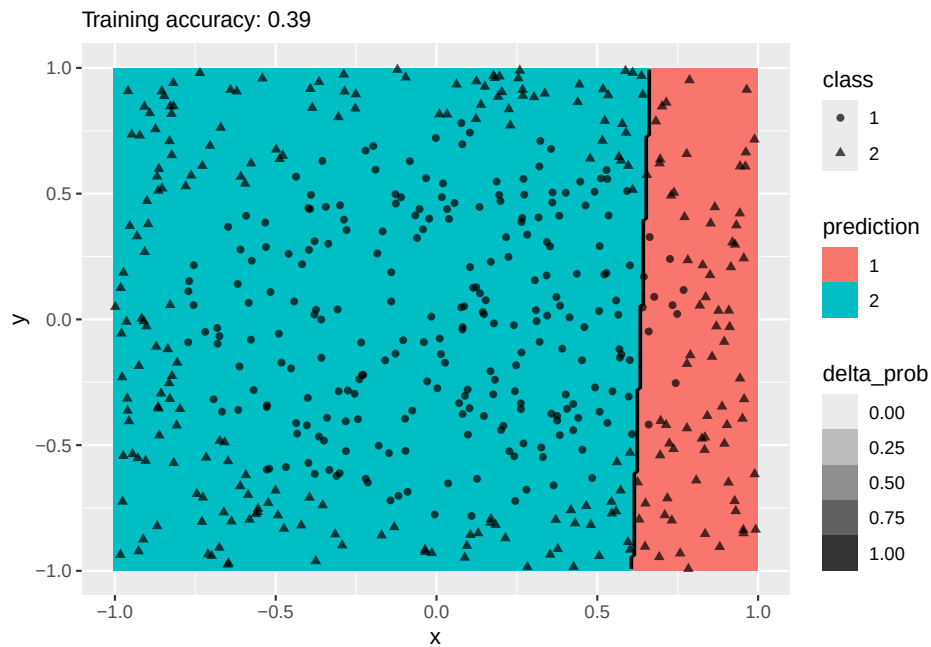


4.12.2.4 Multinomial Logistic Regression

Multinomial logistic regression is an extension of logistic regression to problems with more than two classes.

```
model <- x |> nnet::multinom(class ~., data = _)
## # weights:  4 (3 variable)
## initial value 346.573590
## final value 346.308371
## converged
decisionplot(model, x, class_var = "class") +
  labs(title = "Multinomial Logistic Regression")
## Ignoring unknown labels:
## * title : "Multinomial Logistic Regression"
```

4.12. COMPARING DECISION BOUNDARIES OF POPULAR CLASSIFICATION TECHNIQUES*171

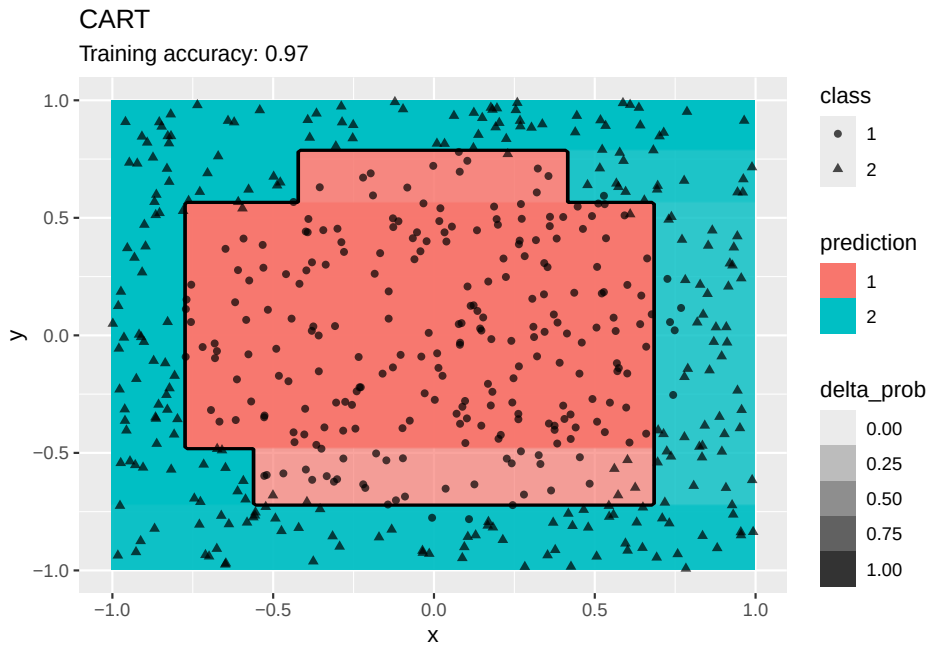


Logistic regression also tries to find a linear decision boundary and fails.

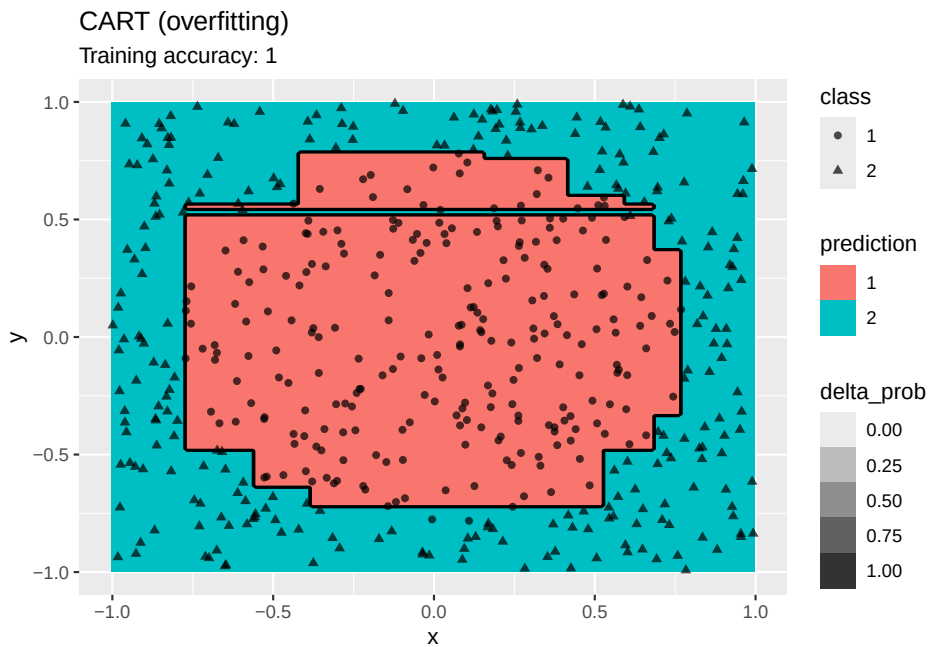
4.12.2.5 Decision Trees

Compare different decision tree algorithms.

```
model <- x |> rpart::rpart(class ~ ., data = _)
decisionplot(model, x, class_var = "class") +
  labs(title = "CART")
```

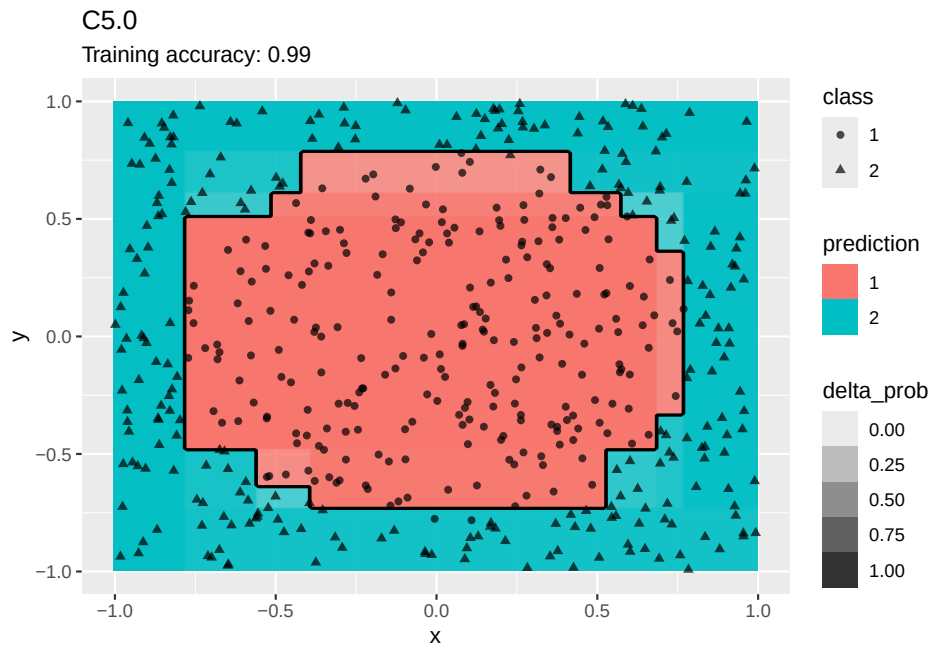


```
model <- x |> rpart::rpart(class ~ ., data = _,
  control = rpart::rpart.control(cp = 0, minsplit = 1))
decisionplot(model, x, class_var = "class") +
  labs(title = "CART (overfitting)")
```



4.12. COMPARING DECISION BOUNDARIES OF POPULAR CLASSIFICATION TECHNIQUES*173

```
model <- x |> C50::C5.0(class ~ ., data = _)
decisionplot(model, x, class_var = "class") +
  labs(title = "C5.0")
```



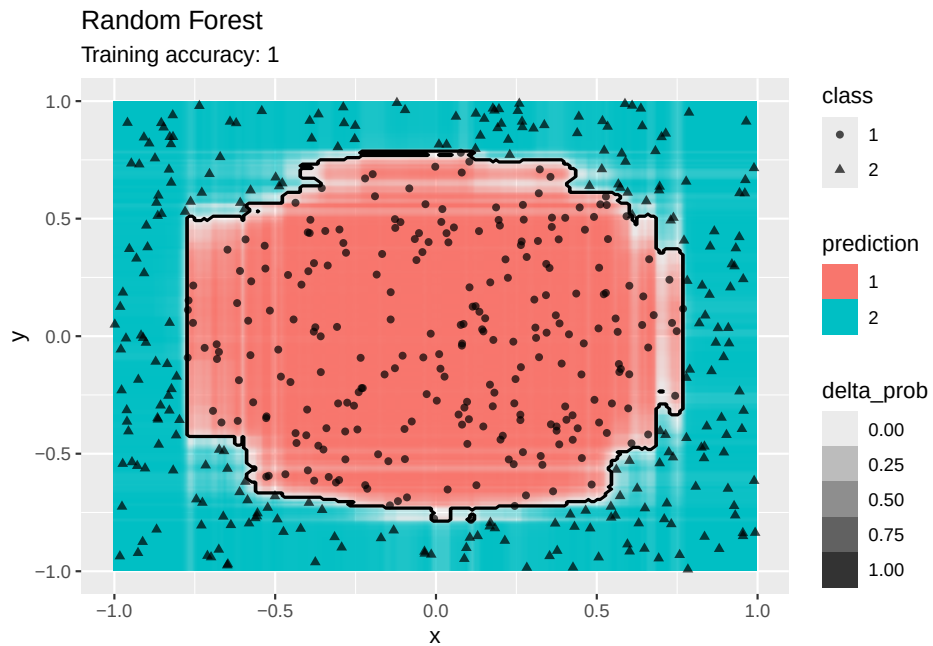
Decision trees do well with the restriction that they can only create cuts parallel to the axes.

4.12.2.6 Ensemble: Random Forest

Try random forest on the dataset.

```
library(randomForest)
## randomForest 4.7-1.2
## 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:ggplot2':
##
##   margin
model <- x |> randomForest(class ~ ., data = _)
```

```
decisionplot(model, x, class_var = "class") +
  labs(title = "Random Forest")
```

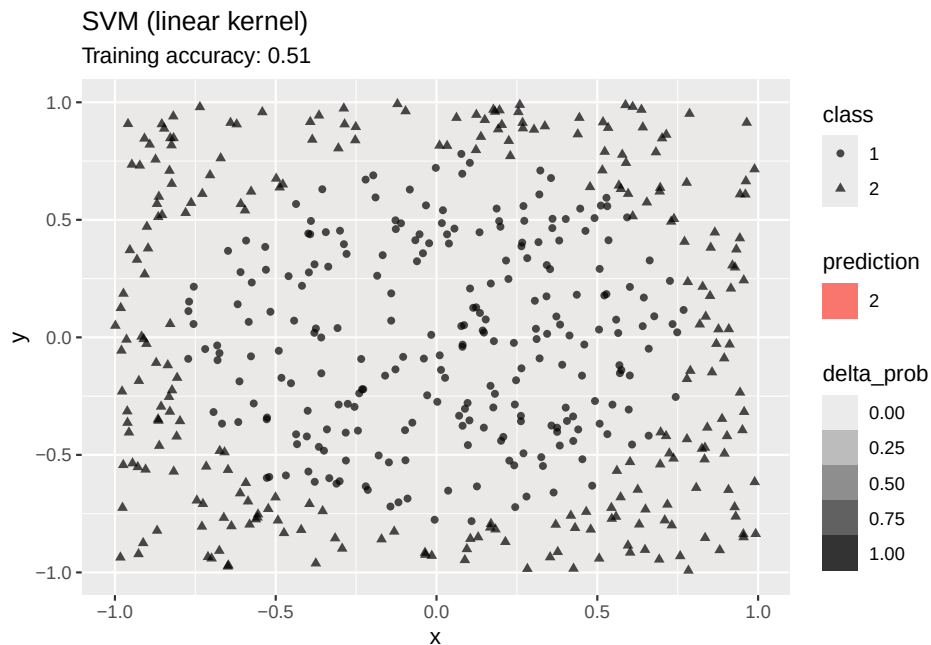


4.12.2.7 Support Vector Machine

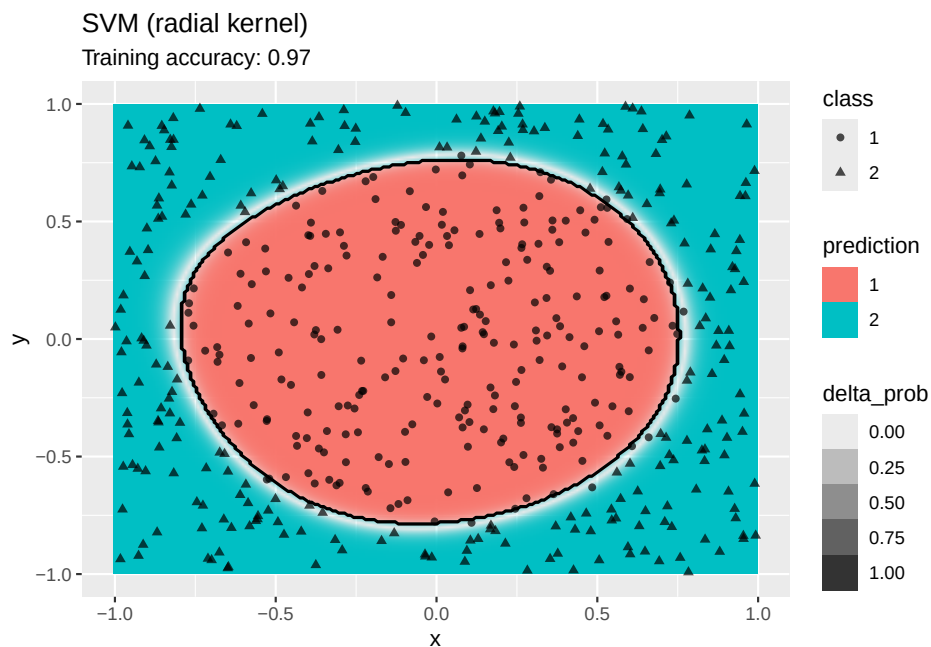
Compare SVMs with different kernels.

```
model <- x |> e1071::svm(class ~ ., data = _,
  kernel = "linear", probability = TRUE)
decisionplot(model, x, class_var = "class") +
  labs(title = "SVM (linear kernel)")
## Warning: Computation failed in `stat_contour()`.
## Caused by error in `zero_range()`:
## ! `x` must be length 1 or 2
```

4.12. COMPARING DECISION BOUNDARIES OF POPULAR CLASSIFICATION TECHNIQUES*175



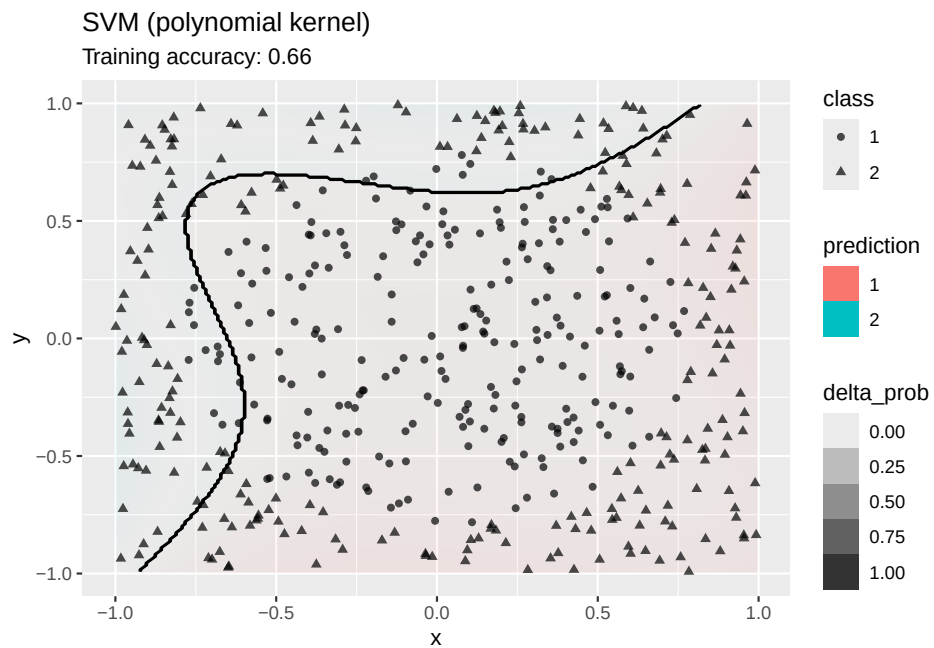
```
model <- x |> e1071::svm(class ~ ., data = _,  
                        kernel = "radial", probability = TRUE)  
decisionplot(model, x, class_var = "class") +  
  labs(title = "SVM (radial kernel)")
```



```

model <- x |> e1071::svm(class ~ ., data = _,
                        kernel = "polynomial", probability = TRUE)
decisionplot(model, x, class_var = "class") +
  labs(title = "SVM (polynomial kernel)")

```

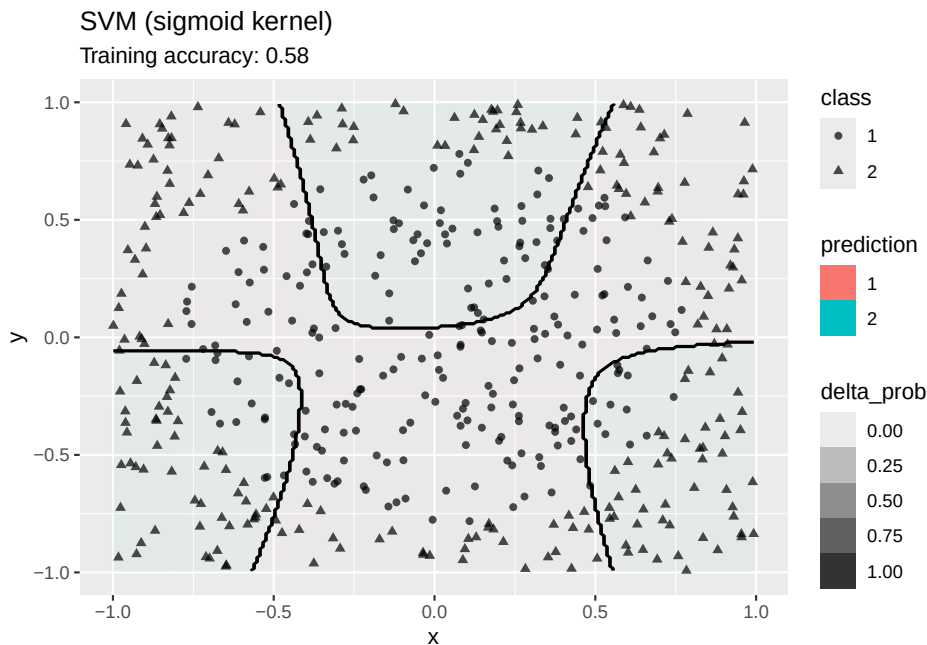


```

model <- x |> e1071::svm(class ~ ., data = _,
                        kernel = "sigmoid", probability = TRUE)
decisionplot(model, x, class_var = "class") +
  labs(title = "SVM (sigmoid kernel)")

```

4.12. COMPARING DECISION BOUNDARIES OF POPULAR CLASSIFICATION TECHNIQUES*177

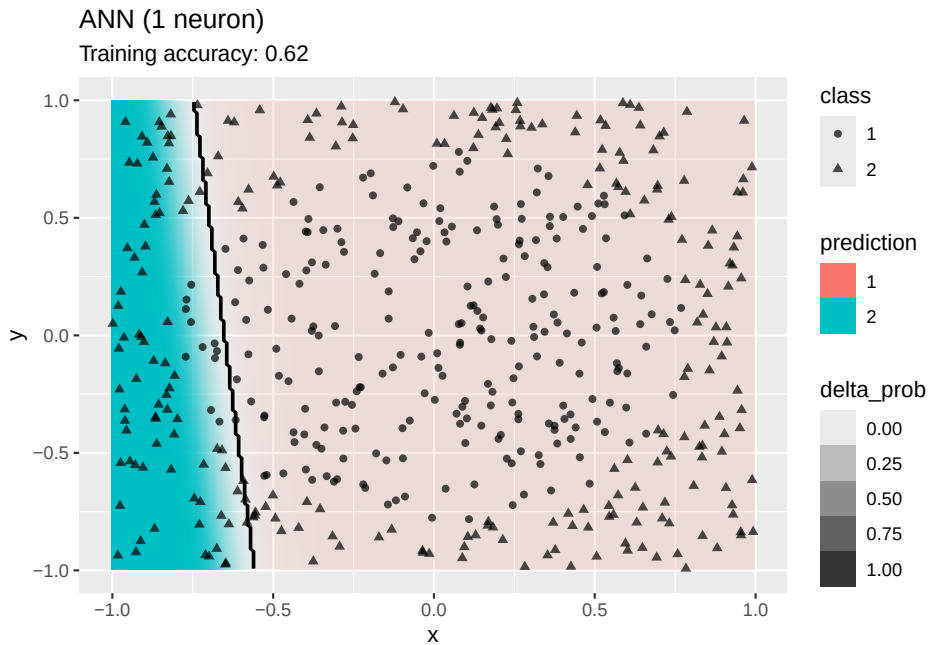


A linear SVM does not work on this data set. An SMV with a radial kernel performs well, the other kernels have issues with finding a linear decision boundary in the projected space.

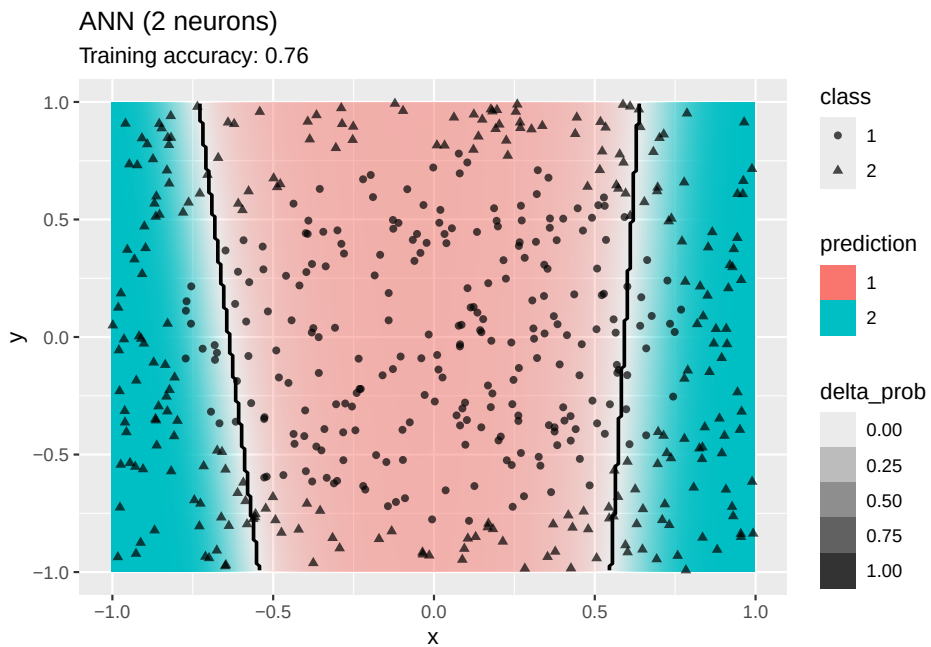
4.12.2.8 Single Layer Feed-forward Neural Networks

Use a simple network with one hidden layer. We will try a different number of neurons for the hidden layer.

```
model <- x |> nnet::nnet(class ~ ., data = _, size = 1, trace = FALSE)
decisionplot(model, x, class_var = "class",
  predict_type = c("class", "raw")) + labs(title = "ANN (1 neuron)")
```

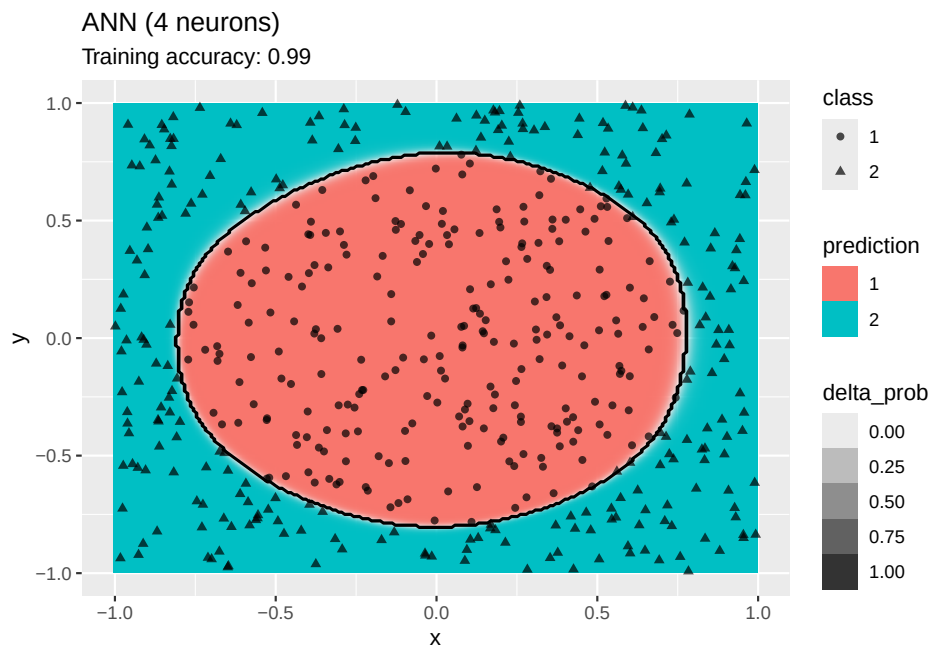


```
model <- x |> nnet::nnet(class ~ ., data = _, size = 2, trace = FALSE)
decisionplot(model, x, class_var = "class",
  predict_type = c("class", "raw")) + labs(title = "ANN (2 neurons)")
```

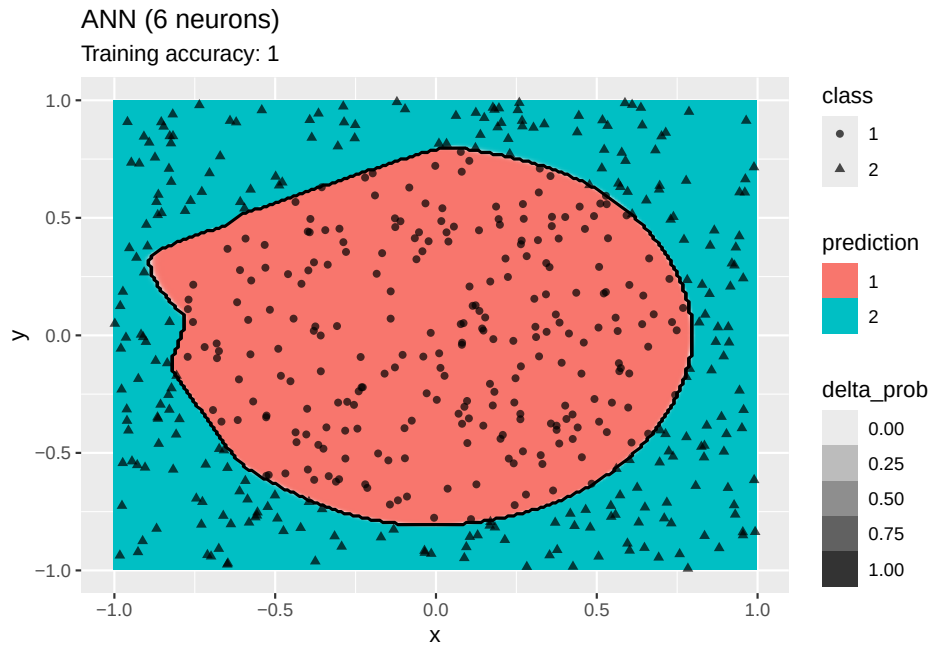


4.12. COMPARING DECISION BOUNDARIES OF POPULAR CLASSIFICATION TECHNIQUES*179

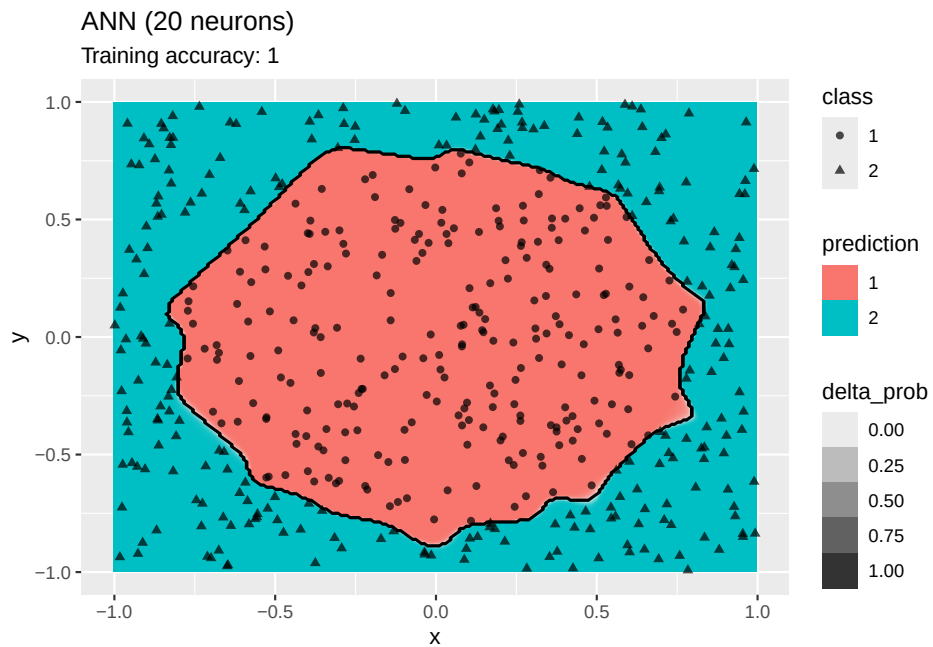
```
model <- x |> nnet::nnet(class ~ ., data = _, size = 4, trace = FALSE)
decisionplot(model, x, class_var = "class",
  predict_type = c("class", "raw")) + labs(title = "ANN (4 neurons)")
```



```
model <- x |> nnet::nnet(class ~ ., data = _, size = 6, trace = FALSE)
decisionplot(model, x, class_var = "class",
  predict_type = c("class", "raw")) + labs(title = "ANN (6 neurons)")
```



```
model <- x |> nnet::nnet(class ~ ., data = _, size = 20, trace = FALSE)
decisionplot(model, x, class_var = "class",
  predict_type = c("class", "raw")) + labs(title = "ANN (20 neurons)")
```



The plots show that a network with 4 and 6 neurons performs well, while a larger number of neurons leads to overfitting the training data.

4.13 More Information on Classification with R

- Package caret: <http://topepo.github.io/caret/index.html>
- Tidymodels (machine learning with tidyverse): <https://www.tidymodels.org/>
- R taskview on machine learning: <http://cran.r-project.org/web/views/MachineLearning.html>

4.14 Exercises*

We will again use the Palmer penguin data for the exercises.

```
library(palmerpenguins)
head(penguins)
## # A tibble: 6 x 8
##   species island   bill_length_mm bill_depth_mm
##   <chr>   <chr>         <dbl>         <dbl>
## 1 Adelie  Torgersen         39.1           18.7
## 2 Adelie  Torgersen         39.5           17.4
## 3 Adelie  Torgersen         40.3           18
## 4 Adelie  Torgersen         NA             NA
## 5 Adelie  Torgersen         36.7           19.3
## 6 Adelie  Torgersen         39.3           20.6
## # i 4 more variables: flipper_length_mm <dbl>,
## #   body_mass_g <dbl>, sex <chr>, year <dbl>
```

Create a R markdown file with the code and do the following below.

1. Apply at least 3 different classification models to the data.
2. Compare the models and a simple baseline model. Which model performs the best? Does it perform significantly better than the other models?

Chapter 5

Association Analysis: Basic Concepts

This chapter introduces association rules mining using the APRIORI algorithm. In addition, analyzing sets of association rules using visualization techniques is demonstrated.

The corresponding chapter of the data mining textbook is available online: Chapter 5: Association Analysis: Basic Concepts and Algorithms.¹

Packages Used in this Chapter

```
pkgs <- c("arules", "arulesViz", "mlbench",
          "palmerpenguins", "tidyverse")

pkgs_install <- pkgs[!(pkgs %in% installed.packages()[,"Package"])]
if(length(pkgs_install)) install.packages(pkgs_install)
```

The packages used for this chapter are:

- *arules* (Hahsler et al. 2025)
- *arulesViz* (Hahsler 2025)
- *mlbench* (Leisch and Dimitriadou 2024)
- *palmerpenguins* (Horst, Hill, and Gorman 2022)
- *tidyverse* (Wickham 2023b)

¹https://mhahsler.github.io/Introduction_to_Data_Mining_R_Examples/DM_chapters/ch5_association_analysis.pdf

5.1 Preliminaries

Association rule mining² plays a vital role in discovering hidden patterns and relationships within large transactional datasets. Applications range from exploratory data analysis in marketing to building rule-based classifiers. Agrawal, Imielinski, and Swami (1993) introduced the problem of mining association rules from transaction data as follows (the definition is taken from Hahsler, Grün, and Hornik (2005)):

Let $I = \{i_1, i_2, \dots, i_n\}$ be a set of n binary attributes called items. Let $D = \{t_1, t_2, \dots, t_m\}$ be a set of transactions called the database. Each transaction in D has a unique transaction ID and contains a subset of the items in I . A rule is defined as an implication of the form $X \Rightarrow Y$ where $X, Y \subseteq I$ and $X \cap Y = \emptyset$ are called itemsets. On itemsets and rules several quality measures can be defined. The most important measures are support and confidence. The support $\text{supp}(X)$ of an itemset X is defined as the proportion of transactions in the data set which contain the itemset. Itemsets with a support which surpasses a user-defined threshold σ are called frequent itemsets. The confidence of a rule is defined as $\text{conf}(X \Rightarrow Y) = \text{supp}(X \cup Y) / \text{supp}(X)$. Association rules are rules with $\text{supp}(X \cup Y) \geq \sigma$ and $\text{conf}(X) \geq \delta$ where σ and δ are user-defined thresholds. The found set of association rules is then used reason about the data.

You can read the free sample chapter from the textbook (Tan, Steinbach, and Kumar 2005): Chapter 5. Association Analysis: Basic Concepts and Algorithms³

5.1.1 The arules Package

Association rule mining in R is implemented in the package **arules**.

```
library(tidyverse)
library(arules)
library(arulesViz)
```

For information about the **arules** package try: `help(package="arules")` and `vignette("arules")` (also available at CRAN⁴)

arules uses the S4 object system to implement classes and methods. Standard R objects use the S3 object system⁵ which do not use formal class definitions and are usually implemented as a list with a class attribute. **arules** and many other

²https://en.wikipedia.org/wiki/Association_rule_learning

³https://mhahsler.github.io/Introduction_to_Data_Mining_R_Examples/DM_chapters/ch5_association_analysis.pdf

⁴<http://cran.r-project.org/web/packages/arules/vignettes/arules.pdf>

⁵<http://adv-r.had.co.nz/S3.html>

R packages use the S4 object system⁶ which is based on formal class definitions with member variables and methods (similar to object-oriented programming languages like Java and C++). Some important differences of using S4 objects compared to the usual S3 objects are:

- coercion (casting): `as(from, "class_name")`
- help for classes: `class? class_name`

5.1.2 Transactions

5.1.2.1 Create Transactions

We will use the Zoo dataset from `mlbench`.

```
data(Zoo, package = "mlbench")
head(Zoo)
```

##	hair	feathers	eggs	milk	airborne	aquatic
## aardvark	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE
## antelope	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE
## bass	FALSE	FALSE	TRUE	FALSE	FALSE	TRUE
## bear	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE
## boar	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE
## buffalo	TRUE	FALSE	FALSE	TRUE	FALSE	FALSE
##	predator	toothed	backbone	breathes	venomous	fins
## aardvark	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
## antelope	FALSE	TRUE	TRUE	TRUE	FALSE	FALSE
## bass	TRUE	TRUE	TRUE	FALSE	FALSE	TRUE
## bear	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
## boar	TRUE	TRUE	TRUE	TRUE	FALSE	FALSE
## buffalo	FALSE	TRUE	TRUE	TRUE	FALSE	FALSE
##	legs	tail	domestic	catsize	type	
## aardvark	4	FALSE	FALSE	TRUE	mammal	
## antelope	4	TRUE	FALSE	TRUE	mammal	
## bass	0	TRUE	FALSE	FALSE	fish	
## bear	4	FALSE	FALSE	TRUE	mammal	
## boar	4	TRUE	FALSE	TRUE	mammal	
## buffalo	4	TRUE	FALSE	TRUE	mammal	

The data in the `data.frame` need to be converted into a set of transactions where each row represents a transaction and each column is translated into items. This is done using the constructor `transactions()`. For the Zoo data set this means that we consider animals as transactions and the different traits (features) will

⁶<http://adv-r.had.co.nz/S4.html>

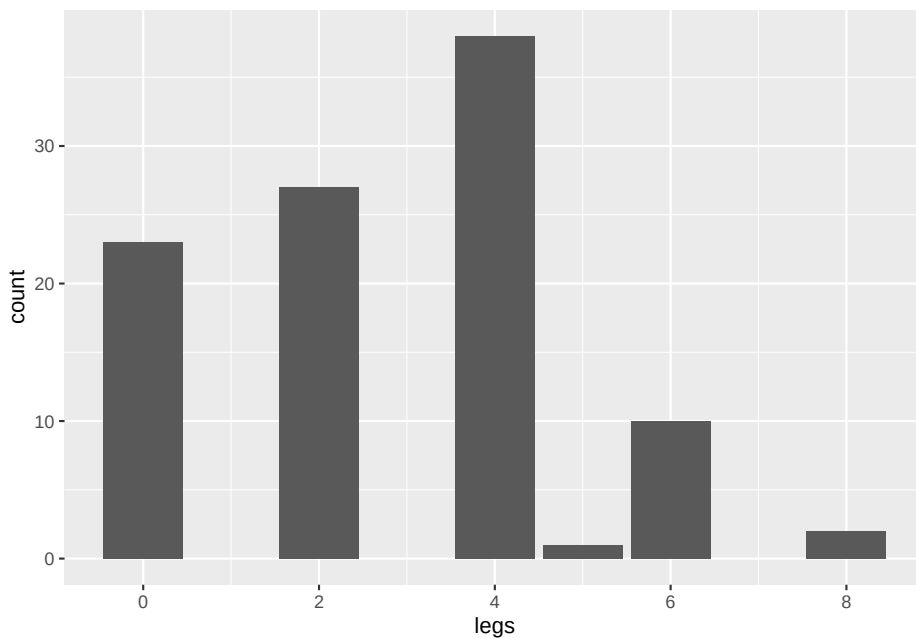
become items that each animal has. For example the animal *antelope* has the item *hair* in its transaction.

```
trans <- transactions(Zoo)
## Warning: Column(s) 13 not logical or factor. Applying
## default discretization (see '? discretizeDF').
```

The conversion gives a warning because only discrete features (**factor** and **logical**) can be directly translated into items. Continuous features need to be discretized first.

What is column 13?

```
summary(Zoo[13])
##      legs
##  Min.   :0.00
## 1st Qu.:2.00
##  Median:4.00
##   Mean  :2.84
## 3rd Qu.:4.00
##   Max.  :8.00
ggplot(Zoo, aes(legs)) + geom_bar()
```



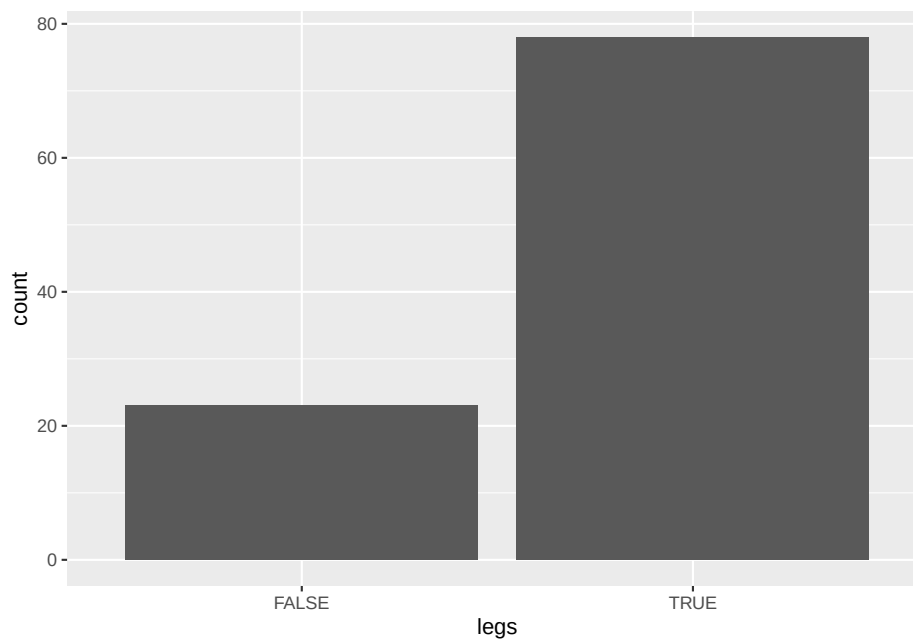
```
Zoo$legs |> table()
##
##  0  2  4  5  6  8
## 23 27 38  1 10  2
```

There are several ways how we can convert a continuous value into discrete items

5.1.2.2 Alternative Encodings for Continuous Values

Alternative 1: Make legs into has/does not have legs

```
Zoo_has_legs <- Zoo |> mutate(legs = legs > 0)
ggplot(Zoo_has_legs, aes(legs)) + geom_bar()
```



```
Zoo_has_legs$legs |> table()
##
## FALSE  TRUE
##   23   78
```

Alternative 2: Use each unique value as an item.

```
Zoo_unique_leg_values <- Zoo |> mutate(legs = factor(legs))
Zoo_unique_leg_values$legs |> head()
## [1] 4 4 0 4 4 4
## Levels: 0 2 4 5 6 8
```

Alternative 3: Use the `discretize` function (see ? `discretize`⁷ and discretization in the code for Chapter 2⁸):

```
Zoo_discretized_legs <- Zoo |> mutate(
  legs = discretize(legs, breaks = 2, method="interval")
)
table(Zoo_discretized_legs$legs)
##
## [0,4) [4,8]
##    50    51
```

In the following we will use **Alternative 1** where the `legs` item indicates if the animal has legs. We convert this data set into a set of transactions.

```
trans <- transactions(Zoo_has_legs)
trans
## transactions in sparse format with
## 101 transactions (rows) and
## 23 items (columns)
```

5.1.2.3 Inspecting Transactions

It is very important to check that the conversion to transactions worked as intended.

```
summary(trans)
## transactions as itemMatrix in sparse format with
## 101 rows (elements/itemsets/transactions) and
## 23 columns (items) and a density of 0.3612
##
## most frequent items:
## backbone breathes legs tail toothed (Other)
##      83      80      78      75      61      462
##
## element (itemset/transaction) length distribution:
## sizes
```

⁷<https://www.rdocumentation.org/packages/arules/topics/discretize>

⁸chap2.html#discretize-features


```
## 3 4 5 6 7 8 9 10 11 12
## 3 2 6 5 8 21 27 25 3 1
##
##      Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##      3.00   8.00   9.00   8.31  10.00  12.00
##
## includes extended item information - examples:
##      labels variables levels
## 1      hair      hair    TRUE
## 2 feathers feathers    TRUE
## 3      eggs      eggs    TRUE
##
## includes extended transaction information - examples:
##      transactionID
## 1      aardvark
## 2      antelope
## 3          bass
```

Look at the created items. They are also called columns names since the transactions are stored as a large, sparse logical matrix.

```
colnames(trans)
## [1] "hair"          "feathers"
## [3] "eggs"          "milk"
## [5] "airborne"      "aquatic"
## [7] "predator"      "toothed"
## [9] "backbone"      "breathes"
## [11] "venomous"      "fins"
## [13] "legs"          "tail"
## [15] "domestic"      "catsize"
## [17] "type=mammal"   "type=bird"
## [19] "type=reptile"  "type=fish"
## [21] "type=amphibian" "type=insect"
## [23] "type=mollusc.et.al"
```

Compare this to the original features (column names) from Zoo.

```
colnames(Zoo)
## [1] "hair"          "feathers" "eggs"      "milk"      "airborne"
## [6] "aquatic"      "predator" "toothed"   "backbone"  "breathes"
## [11] "venomous"     "fins"     "legs"      "tail"      "domestic"
## [16] "catsize"      "type"
```

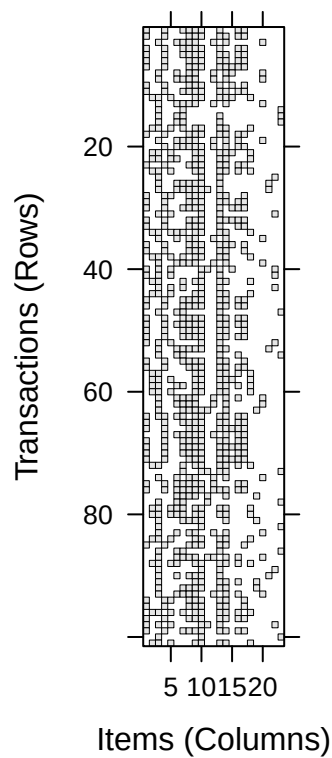
We see that the nominal variable type was converted into several items, one for each value.

We can also look at a (first) few transactions as a logical matrix. TRUE indicates the presence of an item.

```
as(trans, "matrix")[1:3,]
##           hair feathers  eggs  milk airborne aquatic
## aardvark  TRUE      FALSE FALSE  TRUE    FALSE  FALSE
## antelope  TRUE      FALSE FALSE  TRUE    FALSE  FALSE
## bass      FALSE     FALSE  TRUE  FALSE    FALSE  TRUE
##           predator toothed backbone breathes venomous  fins
## aardvark   TRUE      TRUE      TRUE    TRUE    FALSE FALSE
## antelope   FALSE     TRUE      TRUE    TRUE    FALSE FALSE
## bass       TRUE      TRUE      TRUE    FALSE    FALSE  TRUE
##           legs  tail domestic catsize type=mammal type=bird
## aardvark  TRUE FALSE     FALSE    TRUE      TRUE    FALSE
## antelope  TRUE  TRUE     FALSE    TRUE      TRUE    FALSE
## bass      FALSE  TRUE     FALSE    FALSE     FALSE    FALSE
##           type=reptile type=fish type=amphibian type=insect
## aardvark      FALSE     FALSE      FALSE      FALSE
## antelope      FALSE     FALSE      FALSE      FALSE
## bass          FALSE     TRUE      FALSE      FALSE
##           type=mollusc.et.al
## aardvark                FALSE
## antelope                FALSE
## bass                   FALSE
```

The matrix is large. We can get a visual impression by plot the binary matrix as an image. Dark dots represent items (TRUE values in the matrix).

```
image(trans)
```



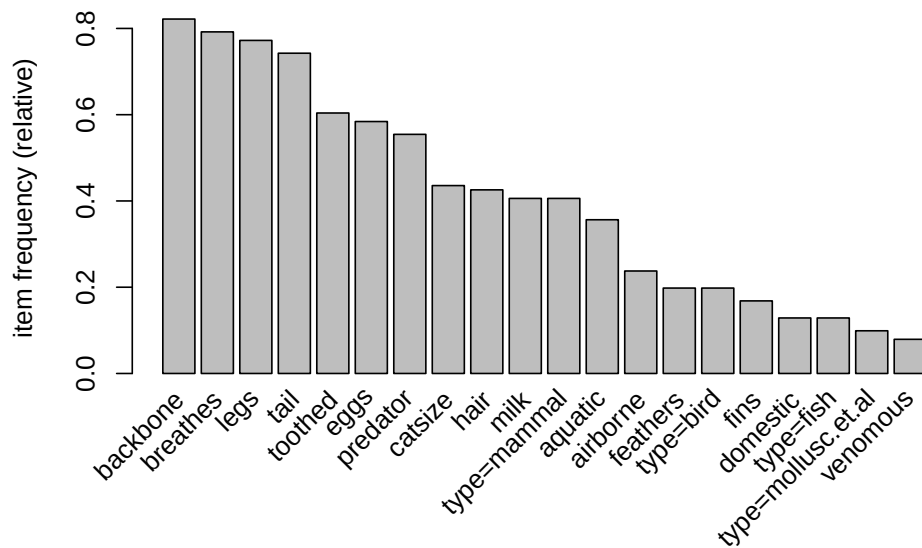
Since each transaction typically only contains a small number of items, it is often more convenient to inspect transactions as sets of items.

```
inspect(trans[1:3])
##      items      transactionID
## [1] {hair,
##      milk,
##      predator,
##      toothed,
##      backbone,
##      breathes,
##      legs,
##      catsize,
##      type=mammal}      aardvark
## [2] {hair,
##      milk,
##      toothed,
##      backbone,
##      breathes,
##      legs,
##      tail,
```

```
##      catsize,
##      type=mammal}      antelope
## [3] {eggs,
##      aquatic,
##      predator,
##      toothed,
##      backbone,
##      fins,
##      tail,
##      type=fish}      bass
```

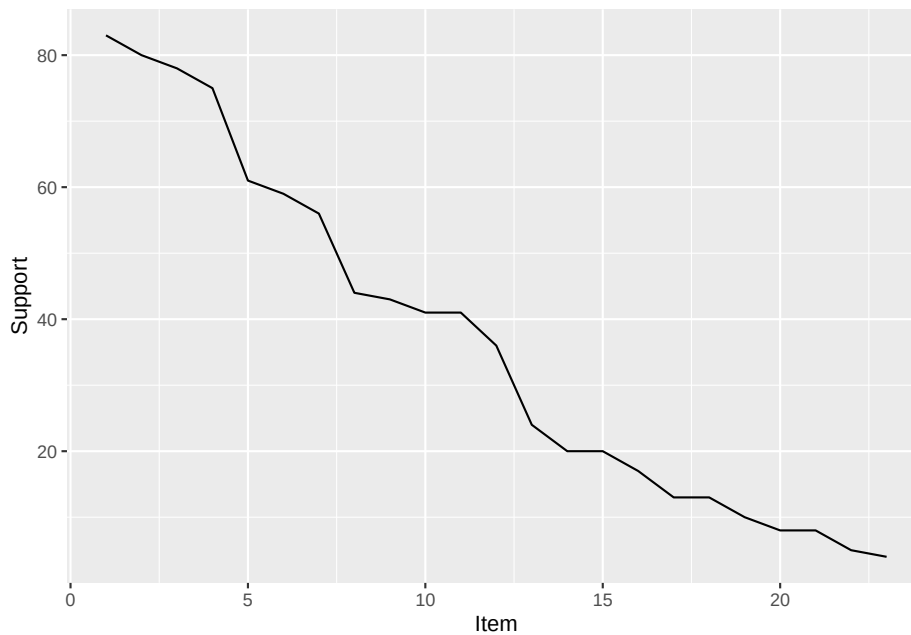
It is often also interesting to look at the relative frequency (=support) of items in the data set. Here we look at the 20 most frequent items.

```
itemFrequencyPlot(trans,topN = 20)
```



The bar chart only works well for a small number of items. The frequent distribution over all items can also be shown as a line graph.

```
ggplot(
  tibble(
    Support = sort(itemFrequency(trans, type = "absolute"),
                  decreasing = TRUE),
    Item = seq_len(ncol(trans))
  ), aes(x = Item, y = Support)) +
  geom_line()
```



5.1.2.4 Negative Items

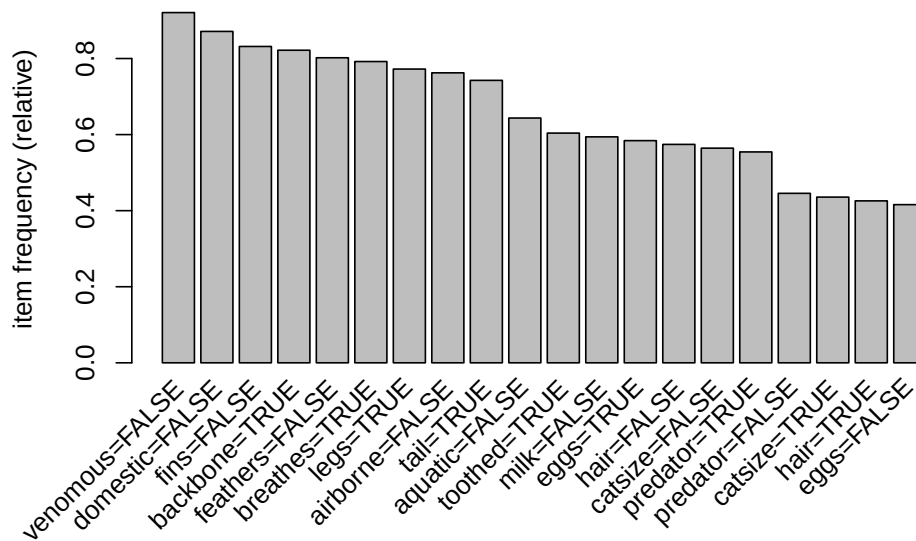
The typical transaction encoding focuses on items for features that the animals have. We can also create items for missing features (e.g., the animal does *not* lay eggs). This can be easily done by converting the logical values into factors. Now we have a nominal variable with two values which will be converted into two items.

```
sapply(Zoo_has_legs, class)
##      hair feathers      eggs      milk airborne aquatic
## "logical" "logical" "logical" "logical" "logical" "logical"
## predator toothed backbone breathes venomous      fins
## "logical" "logical" "logical" "logical" "logical" "logical"
##      legs      tail domestic catsize      type
## "logical" "logical" "logical" "logical" "factor"
Zoo_factors <- Zoo_has_legs |>
  mutate(across(where(is.logical), factor))
sapply(Zoo_factors, class)
##      hair feathers      eggs      milk airborne aquatic
## "factor" "factor" "factor" "factor" "factor" "factor"
## predator toothed backbone breathes venomous      fins
## "factor" "factor" "factor" "factor" "factor" "factor"
##      legs      tail domestic catsize      type
## "factor" "factor" "factor" "factor" "factor"
```

```

summary(Zoo_factors)
##      hair      feathers      eggs      milk      airborne
## FALSE:58  FALSE:81  FALSE:42  FALSE:60  FALSE:77
##  TRUE :43   TRUE :20   TRUE :59   TRUE :41   TRUE :24
##
##
##
##
##
## aquatic    predator    toothed    backbone    breathes
## FALSE:65   FALSE:45   FALSE:40   FALSE:18   FALSE:21
##  TRUE :36   TRUE :56   TRUE :61   TRUE :83   TRUE :80
##
##
##
##
##
## venomous    fins      legs      tail      domestic
## FALSE:93   FALSE:84   FALSE:23  FALSE:26  FALSE:88
##  TRUE : 8   TRUE :17   TRUE :78   TRUE :75   TRUE :13
##
##
##
##
##
## catsize      type
## FALSE:57    mammal      :41
##  TRUE :44    bird        :20
##              reptile     : 5
##              fish        :13
##              amphibian    : 4
##              insect       : 8
##              mollusc.et.al:10
trans_factors <- transactions(Zoo_factors)
trans_factors
## transactions in sparse format with
## 101 transactions (rows) and
## 39 items (columns)
itemFrequencyPlot(trans_factors, topN = 20)

```



```
## Select transactions that contain a certain item
trans_insects <- trans_factors[trans %in% "type=insect"]
trans_insects
## transactions in sparse format with
## 8 transactions (rows) and
## 39 items (columns)
inspect(trans_insects)
##      items      transactionID
## [1] {hair=FALSE,
##      feathers=FALSE,
##      eggs=TRUE,
##      milk=FALSE,
##      airborne=FALSE,
##      aquatic=FALSE,
##      predator=FALSE,
##      toothed=FALSE,
##      backbone=FALSE,
##      breathes=TRUE,
##      venomous=FALSE,
##      fins=FALSE,
##      legs=TRUE,
##      tail=FALSE,
##      domestic=FALSE,
##      catsize=FALSE,
##      type=insect}      flea
## [2] {hair=FALSE,
##      feathers=FALSE,
##      eggs=TRUE,
```

```

##      milk=FALSE,
##      airborne=TRUE,
##      aquatic=FALSE,
##      predator=FALSE,
##      toothed=FALSE,
##      backbone=FALSE,
##      breathes=TRUE,
##      venomous=FALSE,
##      fins=FALSE,
##      legs=TRUE,
##      tail=FALSE,
##      domestic=FALSE,
##      catsize=FALSE,
##      type=insect}          gnat
## [3] {hair=TRUE,
##      feathers=FALSE,
##      eggs=TRUE,
##      milk=FALSE,
##      airborne=TRUE,
##      aquatic=FALSE,
##      predator=FALSE,
##      toothed=FALSE,
##      backbone=FALSE,
##      breathes=TRUE,
##      venomous=TRUE,
##      fins=FALSE,
##      legs=TRUE,
##      tail=FALSE,
##      domestic=TRUE,
##      catsize=FALSE,
##      type=insect}          honeybee
## [4] {hair=TRUE,
##      feathers=FALSE,
##      eggs=TRUE,
##      milk=FALSE,
##      airborne=TRUE,
##      aquatic=FALSE,
##      predator=FALSE,
##      toothed=FALSE,
##      backbone=FALSE,
##      breathes=TRUE,
##      venomous=FALSE,
##      fins=FALSE,
##      legs=TRUE,
##      tail=FALSE,

```



```

##      domestic=FALSE,
##      catsize=FALSE,
##      type=insect}          housefly
## [5] {hair=FALSE,
##      feathers=FALSE,
##      eggs=TRUE,
##      milk=FALSE,
##      airborne=TRUE,
##      aquatic=FALSE,
##      predator=TRUE,
##      toothed=FALSE,
##      backbone=FALSE,
##      breathes=TRUE,
##      venomous=FALSE,
##      fins=FALSE,
##      legs=TRUE,
##      tail=FALSE,
##      domestic=FALSE,
##      catsize=FALSE,
##      type=insect}          ladybird
## [6] {hair=TRUE,
##      feathers=FALSE,
##      eggs=TRUE,
##      milk=FALSE,
##      airborne=TRUE,
##      aquatic=FALSE,
##      predator=FALSE,
##      toothed=FALSE,
##      backbone=FALSE,
##      breathes=TRUE,
##      venomous=FALSE,
##      fins=FALSE,
##      legs=TRUE,
##      tail=FALSE,
##      domestic=FALSE,
##      catsize=FALSE,
##      type=insect}          moth
## [7] {hair=FALSE,
##      feathers=FALSE,
##      eggs=TRUE,
##      milk=FALSE,
##      airborne=FALSE,
##      aquatic=FALSE,
##      predator=FALSE,
##      toothed=FALSE,

```

```
##      backbone=FALSE,
##      breathes=TRUE,
##      venomous=FALSE,
##      fins=FALSE,
##      legs=TRUE,
##      tail=FALSE,
##      domestic=FALSE,
##      catsize=FALSE,
##      type=insect}          termite
## [8] {hair=TRUE,
##      feathers=FALSE,
##      eggs=TRUE,
##      milk=FALSE,
##      airborne=TRUE,
##      aquatic=FALSE,
##      predator=FALSE,
##      toothed=FALSE,
##      backbone=FALSE,
##      breathes=TRUE,
##      venomous=TRUE,
##      fins=FALSE,
##      legs=TRUE,
##      tail=FALSE,
##      domestic=FALSE,
##      catsize=FALSE,
##      type=insect}          wasp
```

5.1.2.5 Vertical Layout (Transaction ID Lists)

The default layout for transactions is horizontal layout (i.e. each transaction is a row). The vertical layout represents transaction data as a list of transaction IDs for each item (= transaction ID lists).

```
vertical <- as(trans, "tidLists")
as(vertical, "matrix")[1:10, 1:5]
##      aardvark antelope  bass  bear  boar
## hair      TRUE      TRUE FALSE  TRUE  TRUE
## feathers  FALSE     FALSE FALSE FALSE FALSE
## eggs      FALSE     FALSE  TRUE FALSE FALSE
## milk      TRUE      TRUE FALSE  TRUE  TRUE
## airborne  FALSE     FALSE FALSE FALSE FALSE
## aquatic   FALSE     FALSE  TRUE FALSE FALSE
## predator  TRUE      FALSE  TRUE  TRUE  TRUE
## toothed   TRUE      TRUE  TRUE  TRUE  TRUE
```

## backbone	TRUE	TRUE	TRUE	TRUE	TRUE
## breathes	TRUE	TRUE	FALSE	TRUE	TRUE

5.2 Frequent Itemset Generation

Even for the small Zoo dataset, we have already a large number of possible itemsets.

```
2^ncol(trans)
## [1] 8388608
```

Finding frequent itemsets uses the parameter `target = "frequent"` (the default target is to directly mine rules).

```
its <- apriori(trans, parameter=list(target = "frequent"))
## Apriori
##
## Parameter specification:
## confidence minval smax arem aval originalSupport maxtime
##          NA    0.1    1 none FALSE          TRUE      5
## support minlen maxlen          target  ext
##    0.1      1      10 frequent itemsets TRUE
##
## Algorithmic control:
## filter tree heap memopt load sort verbose
##    0.1 TRUE TRUE  FALSE TRUE    2    TRUE
##
## Absolute minimum support count: 10
##
## set item appearances ...[0 item(s)] done [0.00s].
## set transactions ...[23 item(s), 101 transaction(s)] done [0.00s].
## sorting and recoding items ... [18 item(s)] done [0.00s].
## creating transaction tree ... done [0.00s].
## checking subsets of size 1 2 3 4 5 6 7 8 9 10
## Warning in apriori(trans, parameter = list(target =
## "frequent")): Mining stopped (maxlen reached). Only
## patterns up to a length of 10 returned!
## done [0.00s].
## sorting transactions ... done [0.00s].
## writing ... [1465 set(s)] done [0.00s].
## creating S4 object ... done [0.00s].
its
## set of 1465 itemsets
```

The default minimum support is .1 (10%). **Note:** The Zoo data set is very small with very few items. For larger datasets, the default minimum support might be too low and you may run out of memory. You probably want to start out with a higher minimum support like .5 (50%) and then work your way down.

We can calculate the support needed to find itemsets that effect at least 5 animals.

```
5/nrow(trans)
## [1] 0.0495
```

This shows that we need to go down to a minimum support of about 5%.

```
its <- apriori(trans, parameter=list(target = "frequent",
                                     support = 0.05))

## Apriori
##
## Parameter specification:
## confidence minval smax arem aval originalSupport maxtime
##          NA    0.1    1 none FALSE          TRUE      5
## support minlen maxlen          target ext
##    0.05      1     10 frequent itemsets TRUE
##
## Algorithmic control:
## filter tree heap memopt load sort verbose
##    0.1 TRUE TRUE  FALSE TRUE    2    TRUE
##
## Absolute minimum support count: 5
##
## set item appearances ...[0 item(s)] done [0.00s].
## set transactions ...[23 item(s), 101 transaction(s)] done [0.00s].
## sorting and recoding items ... [21 item(s)] done [0.00s].
## creating transaction tree ... done [0.00s].
## checking subsets of size 1 2 3 4 5 6 7 8 9 10
## Warning in apriori(trans, parameter = list(target =
## "frequent", support = 0.05)): Mining stopped (maxlen
## reached). Only patterns up to a length of 10 returned!
## done [0.00s].
## sorting transactions ... done [0.00s].
## writing ... [2537 set(s)] done [0.00s].
## creating S4 object ... done [0.00s].
its
## set of 2537 itemsets
```

We will get many itemsets. We can sort them by support, and show the top 10 itemsets.

```

its <- sort(its, by = "support")
its |> head(n = 10) |> inspect()
##      items                support count
## [1] {backbone}             0.8218  83
## [2] {breathes}             0.7921  80
## [3] {legs}                 0.7723  78
## [4] {tail}                 0.7426  75
## [5] {backbone, tail}        0.7327  74
## [6] {breathes, legs}        0.7228  73
## [7] {backbone, breathes}    0.6832  69
## [8] {backbone, legs}        0.6337  64
## [9] {backbone, breathes, legs} 0.6337  64
## [10] {toothed}              0.6040  61

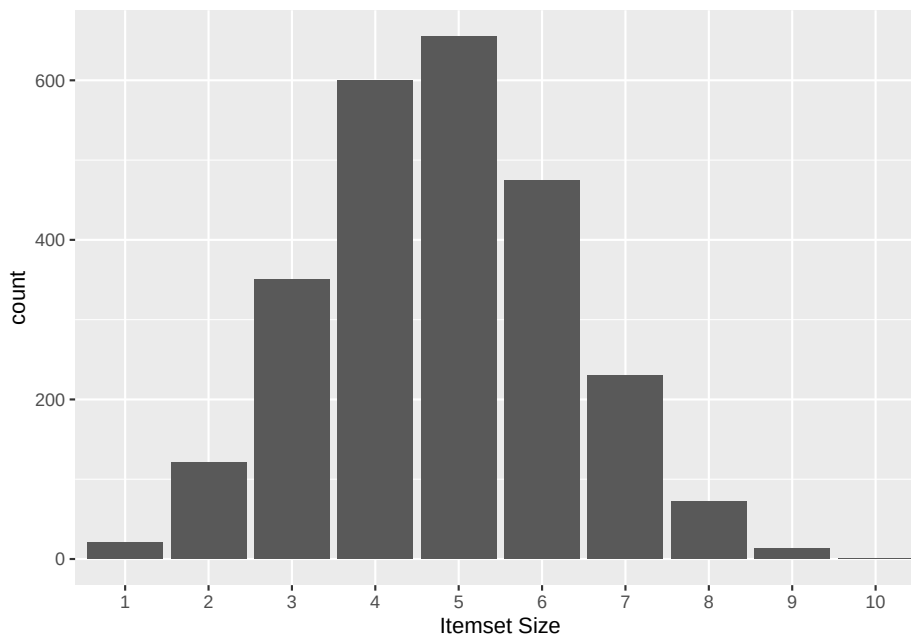
```

We can also look at the largest frequent itemsets. Large means here many items.

```

ggplot(tibble(`Itemset Size` = factor(size(its))),
  aes(`Itemset Size`)) +
  geom_bar()

```



```

its[size(its) > 8] |> inspect()
##      items                support count
## [1] {hair,
##      milk,

```

```

##      toothed,
##      backbone,
##      breathes,
##      legs,
##      tail,
##      catsize,
##      type=mammal} 0.23762      24
## [2] {hair,
##      milk,
##      predator,
##      toothed,
##      backbone,
##      breathes,
##      legs,
##      catsize,
##      type=mammal} 0.15842      16
## [3] {hair,
##      milk,
##      predator,
##      toothed,
##      backbone,
##      breathes,
##      legs,
##      tail,
##      type=mammal} 0.14851      15
## [4] {hair,
##      milk,
##      predator,
##      backbone,
##      breathes,
##      legs,
##      tail,
##      catsize,
##      type=mammal} 0.13861      14
## [5] {hair,
##      milk,
##      predator,
##      toothed,
##      breathes,
##      legs,
##      tail,
##      catsize,
##      type=mammal} 0.12871      13
## [6] {hair,
##      milk,

```

```

##      predator,
##      toothed,
##      backbone,
##      legs,
##      tail,
##      catsize,
##      type=mammal} 0.12871      13
## [7] {hair,
##      milk,
##      predator,
##      toothed,
##      backbone,
##      breathes,
##      tail,
##      catsize,
##      type=mammal} 0.12871      13
## [8] {milk,
##      predator,
##      toothed,
##      backbone,
##      breathes,
##      legs,
##      tail,
##      catsize,
##      type=mammal} 0.12871      13
## [9] {hair,
##      milk,
##      predator,
##      toothed,
##      backbone,
##      breathes,
##      legs,
##      tail,
##      catsize}      0.12871      13
## [10] {hair,
##      predator,
##      toothed,
##      backbone,
##      breathes,
##      legs,
##      tail,
##      catsize,
##      type=mammal} 0.12871      13
## [11] {hair,
##      milk,

```

```

##      predator,
##      toothed,
##      backbone,
##      breathes,
##      legs,
##      tail,
##      catsize,
##      type=mammal} 0.12871      13
## [12] {hair,
##      milk,
##      toothed,
##      backbone,
##      breathes,
##      legs,
##      domestic,
##      catsize,
##      type=mammal} 0.05941      6
## [13] {hair,
##      milk,
##      toothed,
##      backbone,
##      breathes,
##      legs,
##      tail,
##      domestic,
##      type=mammal} 0.05941      6
## [14] {feathers,
##      eggs,
##      airborne,
##      predator,
##      backbone,
##      breathes,
##      legs,
##      tail,
##      type=bird} 0.05941      6

```

5.3 Rule Generation

We use the APRIORI algorithm (see ? `apriori`⁹)

⁹<https://www.rdocumentation.org/packages/arules/topics/apriori>


```

rules <- apriori(trans,
                 parameter = list(support = 0.05,
                                   confidence = 0.9))

## Apriori
##
## Parameter specification:
## confidence minval smax arem aval originalSupport maxtime
##          0.9   0.1   1 none FALSE             TRUE     5
## support minlen maxlen target ext
##          0.05    1    10 rules TRUE
##
## Algorithmic control:
## filter tree heap memopt load sort verbose
##          0.1 TRUE TRUE  FALSE TRUE    2    TRUE
##
## Absolute minimum support count: 5
##
## set item appearances ...[0 item(s)] done [0.00s].
## set transactions ...[23 item(s), 101 transaction(s)] done [0.00s].
## sorting and recoding items ... [21 item(s)] done [0.00s].
## creating transaction tree ... done [0.00s].
## checking subsets of size 1 2 3 4 5 6 7 8 9 10
## Warning in apriori(trans, parameter = list(support = 0.05,
## confidence = 0.9)): Mining stopped (maxlen reached). Only
## patterns up to a length of 10 returned!
## done [0.00s].
## writing ... [7174 rule(s)] done [0.00s].
## creating S4 object ... done [0.00s].
length(rules)
## [1] 7174

```

```

rules |> head() |> inspect()
##      lhs                rhs      support confidence
## [1] {type=insect}      => {eggs}    0.07921 1.0
## [2] {type=insect}      => {legs}    0.07921 1.0
## [3] {type=insect}      => {breathes} 0.07921 1.0
## [4] {type=mollusc.et.al} => {eggs}    0.08911 0.9
## [5] {type=fish}         => {fins}    0.12871 1.0
## [6] {type=fish}         => {aquatic} 0.12871 1.0
##      coverage lift  count
## [1] 0.07921 1.712 8
## [2] 0.07921 1.295 8
## [3] 0.07921 1.262 8
## [4] 0.09901 1.541 9
## [5] 0.12871 5.941 13

```

```
## [6] 0.12871 2.806 13
rules |> head() |> quality()
##   support confidence coverage lift count
## 1 0.07921         1.0 0.07921 1.712    8
## 2 0.07921         1.0 0.07921 1.295    8
## 3 0.07921         1.0 0.07921 1.262    8
## 4 0.08911         0.9 0.09901 1.541    9
## 5 0.12871         1.0 0.12871 5.941   13
## 6 0.12871         1.0 0.12871 2.806   13
```

It is common to sort rules by the lift measure and inspect the rules with the largest lift.

```
rules <- sort(rules, by = "lift")
rules |> head(n = 10) |> inspect()
##      lhs                rhs      support confidence coverage lift count
## [1] {eggs,                => {type=fish} 0.12871         1 0.12871 7.769   13
##      fins}
## [2] {eggs,                => {type=fish} 0.12871         1 0.12871 7.769   13
##      aquatic,
##      fins}
## [3] {eggs,                => {type=fish} 0.08911         1 0.08911 7.769    9
##      predator,
##      fins}
## [4] {eggs,                => {type=fish} 0.12871         1 0.12871 7.769   13
##      toothed,
##      fins}
## [5] {eggs,                => {type=fish} 0.12871         1 0.12871 7.769   13
##      fins,
##      tail}
## [6] {eggs,                => {type=fish} 0.12871         1 0.12871 7.769   13
##      backbone,
##      fins}
## [7] {eggs,                => {type=fish} 0.08911         1 0.08911 7.769    9
##      aquatic,
##      predator,
##      fins}
## [8] {eggs,                => {type=fish} 0.12871         1 0.12871 7.769   13
##      aquatic,
##      toothed,
##      fins}
## [9] {eggs,                => {type=fish} 0.12871         1 0.12871 7.769   13
##      aquatic,
##      fins,
##      tail}
```

```
## [10] {eggs,
##      aquatic,
##      backbone,
##      fins}      => {type=fish} 0.12871      1 0.12871 7.769      13
```

We see that most high-lift rules are about fish. The reason may be since fish have very specific features that are very different from the other animal types in the data.

As a comparison, we show next what rules are generated if we include negative items in the transactions.

```
r <- apriori(trans_factors)
## Apriori
##
## Parameter specification:
## confidence minval smax arem aval originalSupport maxtime
##           0.8   0.1   1 none FALSE                TRUE    5
## support minlen maxlen target ext
##           0.1     1    10 rules TRUE
##
## Algorithmic control:
## filter tree heap memopt load sort verbose
##       0.1 TRUE TRUE  FALSE TRUE    2    TRUE
##
## Absolute minimum support count: 10
##
## set item appearances ...[0 item(s)] done [0.00s].
## set transactions ...[39 item(s), 101 transaction(s)] done [0.00s].
## sorting and recoding items ... [34 item(s)] done [0.00s].
## creating transaction tree ... done [0.00s].
## checking subsets of size 1 2 3 4 5 6 7 8 9 10
## Warning in apriori(trans_factors): Mining stopped (maxlen
## reached). Only patterns up to a length of 10 returned!
## done [0.07s].
## writing ... [1517191 rule(s)] done [0.32s].
## creating S4 object ... done [1.28s].
r
## set of 1517191 rules
```

This encoding creates very dense data (i.e., each transaction contains many items). This leads to finding many rules and may lead to the algorithm running out of memory. This is how much memory the rule set uses.

```
print(object.size(r), unit = "Mb")
## 110.2 Mb
```

Let's inspect the top 10 lift rules.

```
inspect(r[1:10])
```

##	lhs	rhs	support	confidence
## [1]	{}	=> {feathers=FALSE}	0.8020	0.8020
## [2]	{}	=> {backbone=TRUE}	0.8218	0.8218
## [3]	{}	=> {fins=FALSE}	0.8317	0.8317
## [4]	{}	=> {domestic=FALSE}	0.8713	0.8713
## [5]	{}	=> {venomous=FALSE}	0.9208	0.9208
## [6]	{domestic=TRUE}	=> {predator=FALSE}	0.1089	0.8462
## [7]	{domestic=TRUE}	=> {aquatic=FALSE}	0.1188	0.9231
## [8]	{domestic=TRUE}	=> {legs=TRUE}	0.1188	0.9231
## [9]	{domestic=TRUE}	=> {breathes=TRUE}	0.1188	0.9231
## [10]	{domestic=TRUE}	=> {backbone=TRUE}	0.1188	0.9231

```
##      coverage lift count
```

## [1]	1.0000	1.000	81
## [2]	1.0000	1.000	83
## [3]	1.0000	1.000	84
## [4]	1.0000	1.000	88
## [5]	1.0000	1.000	93
## [6]	0.1287	1.899	11
## [7]	0.1287	1.434	12
## [8]	0.1287	1.195	12
## [9]	0.1287	1.165	12
## [10]	0.1287	1.123	12

```
r |> head(n = 10, by = "lift") |> inspect()
```

##	lhs	rhs	support	confidence	coverage	lift	count
## [1]	{breathes=FALSE, fins=TRUE}	=> {type=fish}	0.1287	1	0.1287	7.769	13
## [2]	{eggs=TRUE, fins=TRUE}	=> {type=fish}	0.1287	1	0.1287	7.769	13
## [3]	{milk=FALSE, fins=TRUE}	=> {type=fish}	0.1287	1	0.1287	7.769	13
## [4]	{breathes=FALSE, fins=TRUE, legs=FALSE}	=> {type=fish}	0.1287	1	0.1287	7.769	13
## [5]	{aquatic=TRUE, breathes=FALSE, fins=TRUE}	=> {type=fish}	0.1287	1	0.1287	7.769	13
## [6]	{hair=FALSE, breathes=FALSE, fins=TRUE}	=> {type=fish}	0.1287	1	0.1287	7.769	13

```
## [7] {eggs=TRUE,
##      breathes=FALSE,
##      fins=TRUE}      => {type=fish}  0.1287      1  0.1287 7.769    13
## [8] {milk=FALSE,
##      breathes=FALSE,
##      fins=TRUE}      => {type=fish}  0.1287      1  0.1287 7.769    13
## [9] {toothed=TRUE,
##      breathes=FALSE,
##      fins=TRUE}      => {type=fish}  0.1287      1  0.1287 7.769    13
## [10] {breathes=FALSE,
##       fins=TRUE,
##       tail=TRUE}     => {type=fish}  0.1287      1  0.1287 7.769    13
```

The rules are similar, but now show that fish do not breathe (air) or give milk. It is often beneficial to introduce negative items for some features, but not for all.

5.3.1 Additional Interest Measures

Many different interest measures can be calculated for existing rules (or item-sets) using the function `interestMeasure()`.

```
interestMeasure(rules[1:10], measure = c("phi", "gini"),
  trans = trans)
##      phi  gini
## 1  1.0000 0.2243
## 2  1.0000 0.2243
## 3  0.8138 0.1485
## 4  1.0000 0.2243
## 5  1.0000 0.2243
## 6  1.0000 0.2243
## 7  0.8138 0.1485
## 8  1.0000 0.2243
## 9  1.0000 0.2243
## 10 1.0000 0.2243
```

To store the new interest measures with the rule set, we have to add them to the quality slot.

```
quality(rules) <- cbind(quality(rules),
  interestMeasure(rules, measure = c("phi", "gini"),
    trans = trans))
```

Find rules which score high for Phi correlation between the LHS and RHS of the rule.

```
rules |> head(by = "phi") |> inspect()
##      lhs                rhs      support confidence coverage  lift count phi    gini
## [1] {eggs,
##      fins}              => {type=fish} 0.1287          1    0.1287 7.769    13    1 0.2243
## [2] {eggs,
##      aquatic,
##      fins}              => {type=fish} 0.1287          1    0.1287 7.769    13    1 0.2243
## [3] {eggs,
##      toothed,
##      fins}              => {type=fish} 0.1287          1    0.1287 7.769    13    1 0.2243
## [4] {eggs,
##      fins,
##      tail}              => {type=fish} 0.1287          1    0.1287 7.769    13    1 0.2243
## [5] {eggs,
##      backbone,
##      fins}              => {type=fish} 0.1287          1    0.1287 7.769    13    1 0.2243
## [6] {eggs,
##      aquatic,
##      toothed,
##      fins}              => {type=fish} 0.1287          1    0.1287 7.769    13    1 0.2243
```

5.3.2 Mine Using Templates

Sometimes it is beneficial to specify what items should be where in the found rule. For `apriori()` we can use the parameter `appearance` to specify this (see ? `APappearance`¹⁰). In the following we restrict rules to an animal `type` in the RHS and any item in the LHS. We first create all type related items.

```
type <- grep("type=", itemLabels(trans), value = TRUE)
type
## [1] "type=mammal"      "type=bird"
## [3] "type=reptile"     "type=fish"
## [5] "type=amphibian"   "type=insect"
## [7] "type=mollusc.et.al"
```

Now, we can restrict the appearance in the RHS.

```
rules_type <- apriori(trans, appearance= list(rhs = type))
## Apriori
##
## Parameter specification:
## confidence minval  smax  arem  aval originalSupport maxtime
```

¹⁰<https://www.rdocumentation.org/packages/arules/topics/APappearance>

```
##      0.8    0.1    1 none FALSE          TRUE      5
## support minlen maxlen target ext
##      0.1      1     10 rules TRUE
##
## Algorithmic control:
## filter tree heap memopt load sort verbose
##      0.1 TRUE TRUE  FALSE TRUE    2    TRUE
##
## Absolute minimum support count: 10
##
## set item appearances ...[7 item(s)] done [0.00s].
## set transactions ...[23 item(s), 101 transaction(s)] done [0.00s].
## sorting and recoding items ... [18 item(s)] done [0.00s].
## creating transaction tree ... done [0.00s].
## checking subsets of size 1 2 3 4 5 6 7 8 9 10
## Warning in apriori(trans, appearance = list(rhs = type)):
## Mining stopped (maxlen reached). Only patterns up to a
## length of 10 returned!
## done [0.00s].
## writing ... [571 rule(s)] done [0.00s].
## creating S4 object ... done [0.00s].
```

```
rules_type |> sort(by = "lift") |> head() |> inspect()
##      lhs                rhs      support confidence coverage  lift count
## [1] {eggs,
##      fins}              => {type=fish} 0.1287          1 0.1287 7.769    13
## [2] {eggs,
##      aquatic,
##      fins}              => {type=fish} 0.1287          1 0.1287 7.769    13
## [3] {eggs,
##      toothed,
##      fins}              => {type=fish} 0.1287          1 0.1287 7.769    13
## [4] {eggs,
##      fins,
##      tail}              => {type=fish} 0.1287          1 0.1287 7.769    13
## [5] {eggs,
##      backbone,
##      fins}              => {type=fish} 0.1287          1 0.1287 7.769    13
## [6] {eggs,
##      aquatic,
##      toothed,
##      fins}              => {type=fish} 0.1287          1 0.1287 7.769    13
```

5.3.3 Redundant Rules

Association rule mining is prone to generate too many very specific rules. A rule can be defined as redundant if a more general rules with the same or a higher confidence exists.

```
rules_type
## set of 571 rules
table(size(rules_type))
##
##  2  3  4  5  6  7  8  9 10
##  3 29 95 155 150 92 37 9 1
```

```
rules_type_non_redundant <- rules_type[!is.redundant(rules_type)]
rules_type_non_redundant
## set of 29 rules
table(size(rules_type_non_redundant))
##
##  2  3  4  5
##  3 13 10 3
```

```
inspect(rules_type_non_redundant)
##      lhs      rhs      support confidence coverage lift count
## [1] {feathers} => {type=bird}  0.1980      1.0000  0.1980 5.050   20
## [2] {milk}    => {type=mammal} 0.4059      1.0000  0.4059 2.463   41
## [3] {hair}    => {type=mammal} 0.3861      0.9070  0.4257 2.234   39
## [4] {eggs,    => {type=fish}   0.1287      1.0000  0.1287 7.769   13
##      fins}
## [5] {fins,    => {type=fish}   0.1287      0.8125  0.1584 6.312   13
##      tail}
## [6] {airborne,=> {type=bird}   0.1584      0.8889  0.1782 4.489   16
##      tail}
## [7] {airborne,=> {type=bird}   0.1584      0.8889  0.1782 4.489   16
##      backbone}
## [8] {hair,    => {type=mammal} 0.2970      1.0000  0.2970 2.463   30
##      catsize}
## [9] {hair,    => {type=mammal} 0.1980      1.0000  0.1980 2.463   20
##      predator}
## [10] {hair,   => {type=mammal} 0.3762      1.0000  0.3762 2.463   38
##      toothed}
## [11] {hair,   => {type=mammal} 0.3267      1.0000  0.3267 2.463   33
##      tail}
## [12] {hair,   => {type=mammal} 0.3861      1.0000  0.3861 2.463   39
##      backbone}
## [13] {toothed,
```


##	catsize}	=> {type=mammal}	0.3069	0.8857	0.3465	2.182	31
##	[14] {breathes,						
##	catsize}	=> {type=mammal}	0.3168	0.8205	0.3861	2.021	32
##	[15] {toothed,						
##	legs}	=> {type=mammal}	0.3663	0.8810	0.4158	2.170	37
##	[16] {toothed,						
##	breathes}	=> {type=mammal}	0.3960	0.8511	0.4653	2.097	40
##	[17] {eggs,						
##	airborne,						
##	tail}	=> {type=bird}	0.1584	1.0000	0.1584	5.050	16
##	[18] {eggs,						
##	airborne,						
##	backbone}	=> {type=bird}	0.1584	1.0000	0.1584	5.050	16
##	[19] {eggs,						
##	legs,						
##	tail}	=> {type=bird}	0.1980	0.8333	0.2376	4.208	20
##	[20] {predator,						
##	legs,						
##	catsize}	=> {type=mammal}	0.1683	0.8095	0.2079	1.994	17
##	[21] {predator,						
##	breathes,						
##	catsize}	=> {type=mammal}	0.1980	0.8696	0.2277	2.142	20
##	[22] {toothed,						
##	legs,						
##	catsize}	=> {type=mammal}	0.2772	1.0000	0.2772	2.463	28
##	[23] {toothed,						
##	breathes,						
##	catsize}	=> {type=mammal}	0.3069	1.0000	0.3069	2.463	31
##	[24] {backbone,						
##	legs,						
##	catsize}	=> {type=mammal}	0.2871	0.8056	0.3564	1.984	29
##	[25] {toothed,						
##	legs,						
##	tail}	=> {type=mammal}	0.3168	0.9412	0.3366	2.319	32
##	[26] {toothed,						
##	breathes,						
##	tail}	=> {type=mammal}	0.3366	0.8947	0.3762	2.204	34
##	[27] {eggs,						
##	aquatic,						
##	toothed,						
##	tail}	=> {type=fish}	0.1287	0.9286	0.1386	7.214	13
##	[28] {predator,						
##	legs,						
##	tail,						
##	catsize}	=> {type=mammal}	0.1386	0.8235	0.1683	2.029	14

```
## [29] {predator,
##      backbone,
##      legs,
##      catsize} => {type=mammal} 0.1683    0.8500    0.1980 2.094    17
```

We see that this simple redundancy check results in a rule set of a manageable size which can be easily analyzed by a human.

5.3.4 Saving Rules for External Tools

Rules can be saved as a CSV-file to be opened with Excel or other tools.

```
write(rules, file = "rules.csv", quote = TRUE)
```

Another option is to export rules in PMML (Predictive Model Markup Language) format. See `pmml()`.

5.4 Compact Representation of Frequent Itemsets

An itemset is maximal in a set if no proper superset of the itemset is contained in the set. Since we are often interested in the most specific itemsets with the largest number of items this is often useful for reducing the size of the set for manual inspection.

```
its_max <- its[is.maximal(its)]
its_max
## set of 22 itemsets
its_max |> head(by = "support") |> inspect()
##      items      support count
## [1] {hair,
##      milk,
##      predator,
##      toothed,
##      backbone,
##      breathes,
##      legs,
##      tail,
##      catsize,
##      type=mammal} 0.12871    13
## [2] {eggs,
##      aquatic,
##      predator,
```

```
##      toothed,
##      backbone,
##      fins,
##      tail,
##      type=fish} 0.08911      9
## [3] {aquatic,
##      predator,
##      toothed,
##      backbone,
##      breathes} 0.07921      8
## [4] {aquatic,
##      predator,
##      toothed,
##      backbone,
##      fins,
##      tail,
##      catsize} 0.06931      7
## [5] {eggs,
##      venomous} 0.05941      6
## [6] {predator,
##      venomous} 0.05941      6
```

Another compact representation is the idea of closed itemsets. The closure of an itemset is its largest proper superset which has the same support (is contained in exactly the same transactions). An itemset is closed, if it is its own closure. This is similar to the idea of maximal itemsets but only removes itemsets if a superset with the same support exists.

```
its_closed <- its[is.closed(its)]
its_closed
## set of 230 itemsets
```

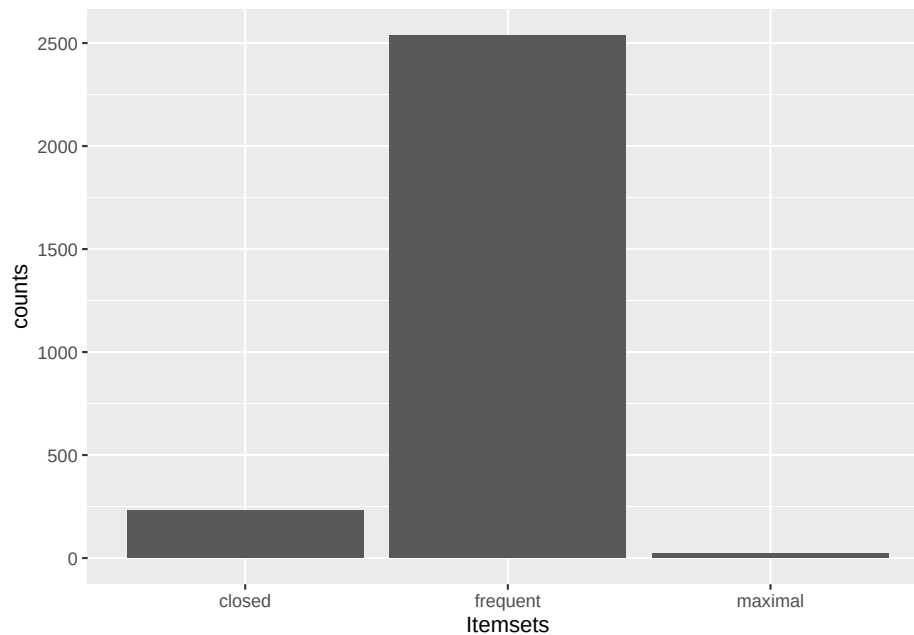
```
its_closed |> head(by = "support") |> inspect()
##      items      support count
## [1] {backbone} 0.8218 83
## [2] {breathes} 0.7921 80
## [3] {legs} 0.7723 78
## [4] {tail} 0.7426 75
## [5] {backbone, tail} 0.7327 74
## [6] {breathes, legs} 0.7228 73
```

```
counts <- c(
  frequent=length(its),
  closed=length(its_closed),
```

```
maximal=length(its_max)
)
```

Here is a comparison of the number of frequent itemsets with closed and maximal itemsets.

```
ggplot(as_tibble(counts, rownames = "Itemsets"),
  aes(Itemsets, counts)) + geom_bar(stat = "identity")
```



5.5 Association Rule Visualization*

Visualization is a very powerful approach to analyse large sets of mined association rules and frequent itemsets. We present here some options to create static visualizations and inspect rule sets interactively.

5.5.1 Static Visualizations

Load the `arulesViz` library.

```
library(arulesViz)
library(tidyverse)
```

Create rules for the Zoo dataset converting the legs variable into a binary indicator.

```
data(Zoo, package = "mlbench")
Zoo_has_legs <- Zoo |> mutate(legs = legs > 0)
trans <- transactions(Zoo_has_legs)
trans
## transactions in sparse format with
## 101 transactions (rows) and
## 23 items (columns)

rules <- apriori(trans,
                  parameter = list(support = 0.05,
                                   confidence = 0.9))

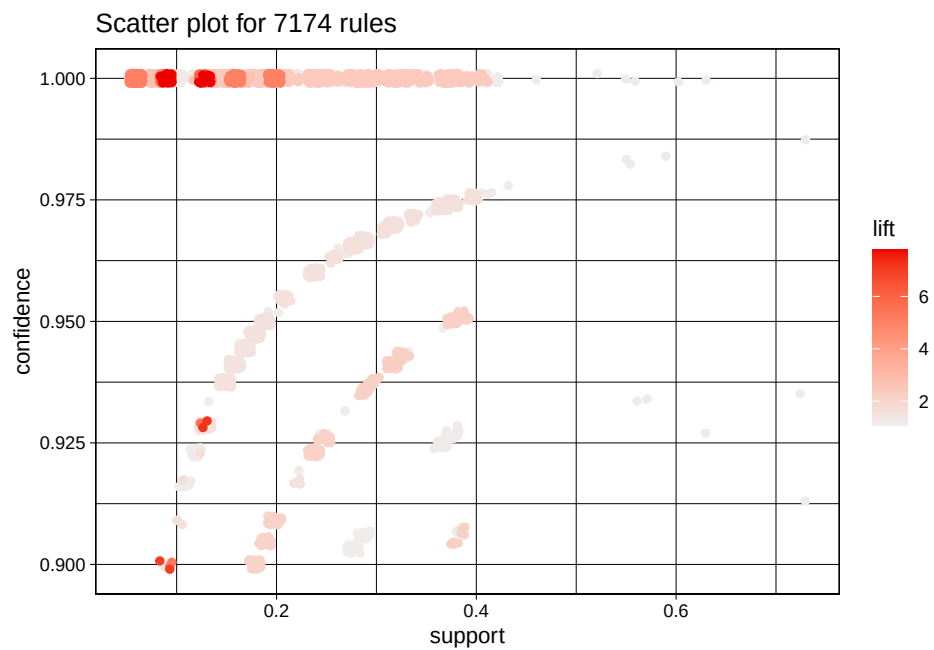
## Apriori
##
## Parameter specification:
## confidence minval smax arem aval originalSupport maxtime
##          0.9   0.1   1 none FALSE             TRUE     5
## support minlen maxlen target ext
##          0.05    1    10  rules TRUE
##
## Algorithmic control:
## filter tree heap memopt load sort verbose
##          0.1 TRUE TRUE  FALSE TRUE    2    TRUE
##
## Absolute minimum support count: 5
##
## set item appearances ...[0 item(s)] done [0.00s].
## set transactions ...[23 item(s), 101 transaction(s)] done [0.00s].
## sorting and recoding items ... [21 item(s)] done [0.00s].
## creating transaction tree ... done [0.00s].
## checking subsets of size 1 2 3 4 5 6 7 8 9 10
## Warning in apriori(trans, parameter = list(support = 0.05,
## confidence = 0.9)): Mining stopped (maxlen reached). Only
## patterns up to a length of 10 returned!
## done [0.00s].
## writing ... [7174 rule(s)] done [0.00s].
## creating S4 object ... done [0.00s].
quality(rules) <- cbind(quality(rules),
                        interestMeasure(rules, measure = c("phi", "gini"),
                                         trans = trans))

rules
## set of 7174 rules
```

5.5.1.1 Scatterplot

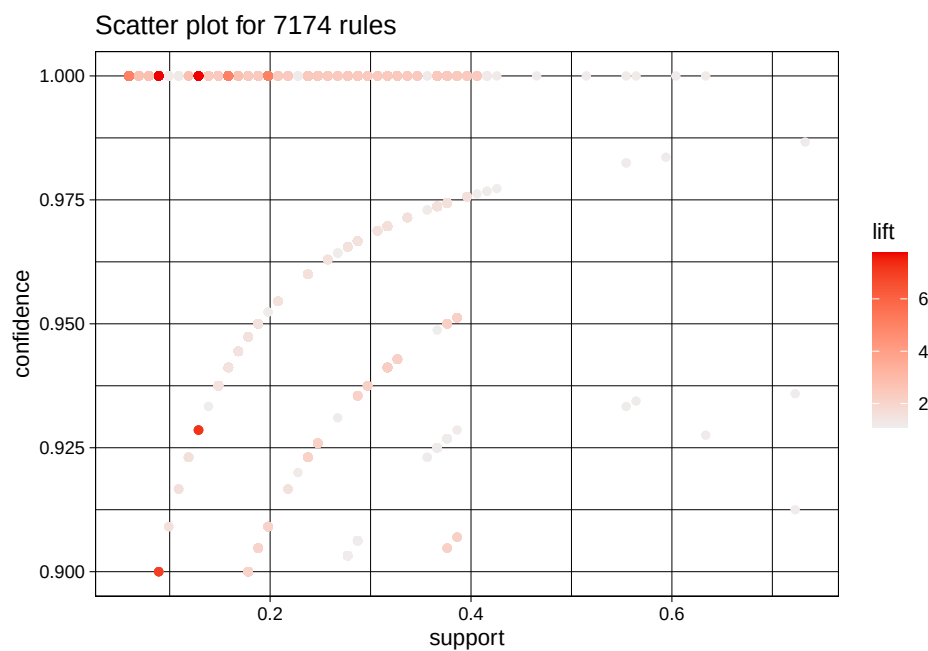
The default plot for association rules is a support/confidence scatterplot.

```
plot(rules)
## To reduce overplotting, jitter is added! Use jitter = 0 to prevent jitter.
```

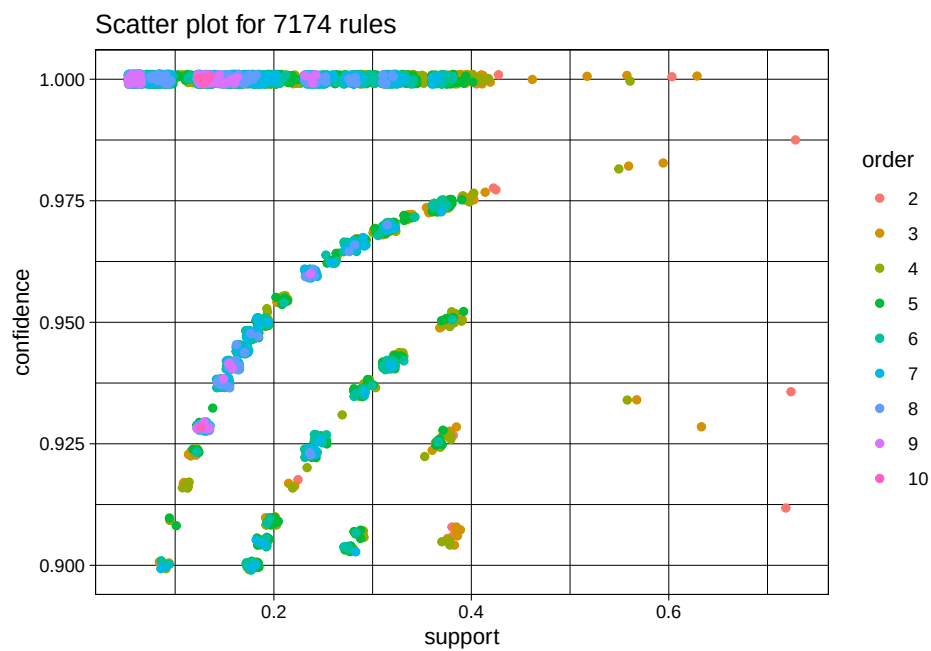


Each rules is represented by a point. Note that some jitter (randomly move points) was added to show how many rules have the same confidence and support value. Without jitter:

```
plot(rules, control = list(jitter = 0))
```



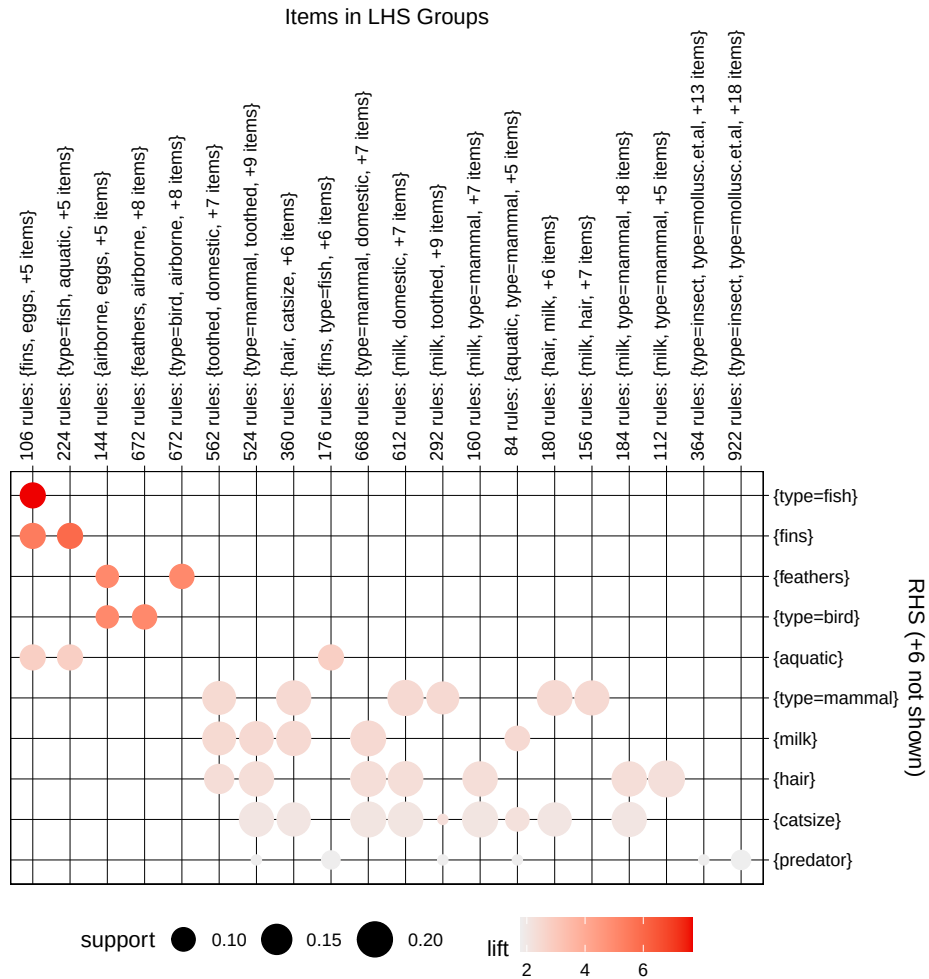
```
plot(rules, shading = "order")
## To reduce overplotting, jitter is added! Use jitter = 0 to prevent jitter.
```



5.5.1.2 Grouped Matrix Plot

The grouped matrix plot tries to group rules with a similar relationship between the rule's LHS and RHS and represents the groups support and lift using a balloon plot. Grouping is performed using clustering. Groups are organized such that the most interesting rules appear to the top left corner.

```
set.seed(1234)
plot(rules, method = "grouped matrix")
## Registered S3 methods overwritten by 'registry':
##   method          from
##   print.registry_field proxy
##   print.registry_entry proxy
```



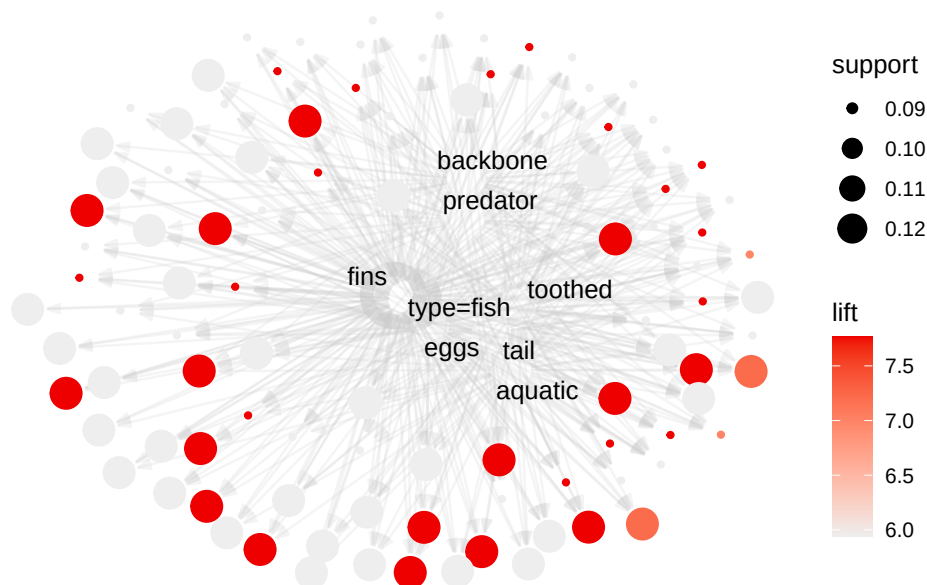
The most interesting rules are about the relationship between fins, eggs and 5 more items and the type fish.

This plot can also be used interactively using the parameter `engine = "interactive"`.

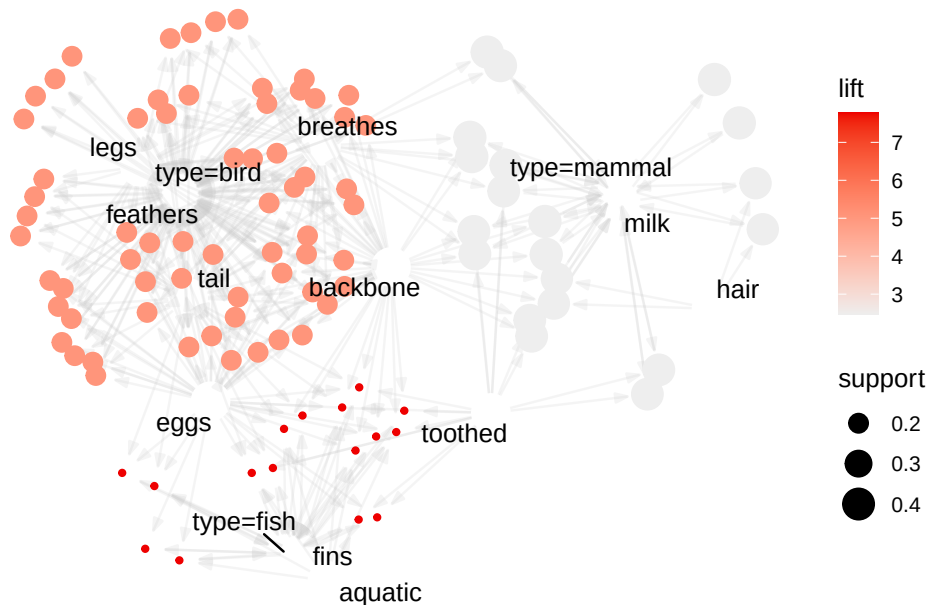
5.5.1.3 Graph-based Visualization

Graph-based techniques visualize association rules using vertices and edges, where vertices typically represent items or itemsets and edges indicate relationships in terms of rules. Interest measures are typically added to the plot as labels on the edges or by color or width of the arrows displaying the edges.

```
plot(rules, method = "graph")  
## Warning: Too many rules supplied. Only plotting the best  
## 100 using 'lift' (change control parameter max if needed).
```



```
plot(rules |> head(by = "phi", n = 100), method = "graph")
```



We see that the automatic graph layout organizes the rules roughly into three groups representing the three most frequent animal types: birds, mammals and fish. we can also see that birds and mammals share items like breathes while birds and fish share eggs.

5.5.2 Interactive Visualizations

Interactive visualizations let the user explore the large number of rules and item-sets and learn about the structure of the data. The overview article *arulesViz: Interactive Visualization of Association Rules with R*¹¹ provides an in-depth discussion of interactive association rule visualization.

We will use the association rules mined from the Iris dataset for the following examples.

```
data(iris)
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

¹¹<https://journal.r-project.org/archive/2017/RJ-2017-047/RJ-2017-047.pdf>

```
## 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
##
##
##
```

Convert the data to transactions.

```
iris_trans <- transactions(iris)
## Warning: Column(s) 1, 2, 3, 4 not logical or factor.
## Applying default discretization (see '? discretizeDF').
```

Note that this conversion gives a warning to indicate that some potentially unwanted conversion happens. Some features are numeric and need to be discretized. The conversion automatically applies frequency-based discretization with 3 classes to each numeric feature, however, the user may want to use a different discretization strategy.

```
iris_trans |> head() |> inspect()
## items transactionID
## [1] {Sepal.Length=[4.3,5.4),
## Sepal.Width=[3.2,4.4],
## Petal.Length=[1,2.63),
## Petal.Width=[0.1,0.867),
## Species=setosa} 1
## [2] {Sepal.Length=[4.3,5.4),
## Sepal.Width=[2.9,3.2),
## Petal.Length=[1,2.63),
## Petal.Width=[0.1,0.867),
## Species=setosa} 2
## [3] {Sepal.Length=[4.3,5.4),
## Sepal.Width=[3.2,4.4],
## Petal.Length=[1,2.63),
## Petal.Width=[0.1,0.867),
## Species=setosa} 3
## [4] {Sepal.Length=[4.3,5.4),
## Sepal.Width=[2.9,3.2),
## Petal.Length=[1,2.63),
## Petal.Width=[0.1,0.867),
## Species=setosa} 4
## [5] {Sepal.Length=[4.3,5.4),
```

```
##      Sepal.Width=[3.2,4.4],
##      Petal.Length=[1,2.63),
##      Petal.Width=[0.1,0.867),
##      Species=setosa}                                5
## [6] {Sepal.Length=[5.4,6.3),
##      Sepal.Width=[3.2,4.4],
##      Petal.Length=[1,2.63),
##      Petal.Width=[0.1,0.867),
##      Species=setosa}                                6
```

Next, we mine association rules.

```
rules <- apriori(iris_trans, parameter = list(support = 0.1,
                                              confidence = 0.8))

## Apriori
##
## Parameter specification:
## confidence minval smax arem aval originalSupport maxtime
##          0.8    0.1    1 none FALSE              TRUE     5
## support minlen maxlen target ext
##          0.1      1     10 rules TRUE
##
## Algorithmic control:
## filter tree heap memopt load sort verbose
##    0.1 TRUE TRUE  FALSE TRUE    2    TRUE
##
## Absolute minimum support count: 15
##
## set item appearances ...[0 item(s)] done [0.00s].
## set transactions ...[15 item(s), 150 transaction(s)] done [0.00s].
## sorting and recoding items ... [15 item(s)] done [0.00s].
## creating transaction tree ... done [0.00s].
## checking subsets of size 1 2 3 4 5 done [0.00s].
## writing ... [144 rule(s)] done [0.00s].
## creating S4 object ... done [0.00s].
rules
## set of 144 rules
```

5.5.2.1 Interactive Inspect With Sorting, Filtering and Paging

An interactive table that lets the user sort and filter the rules is a very effective exploration tool. We can quickly find the highest lift rules and filter by interesting items in the LHS and RHS of the rules.

```
inspectDT(rules,options = list(scrollX = TRUE))
```

The resulting interactive table can be seen in the online version of this book.¹²

5.5.2.2 Scatter Plot

Plot rules as a scatter plot using an interactive html widget. To avoid overplotting, jitter is added automatically. Set `jitter = 0` to disable jitter. Hovering over rules shows rule information. *Note:* plotly/javascript does not do well with too many points, so plot selects the top 1000 rules with a warning if more rules are supplied.

```
plot(rules, engine = "html")
```

The resulting interactive plot can be seen in the online version of this book.¹³

5.5.2.3 Matrix Visualization

The rules are organized in a matrix where the columns represent unique LHS itemsets and the rows are the RHS items. Hovering over rules shows rule information.

```
plot(rules, method = "matrix", engine = "html")
```

The resulting interactive plot can be seen in the online version of this book.¹⁴

5.5.2.4 Visualization as Graph

Plot rules as an interactive graph with items as rectangular vertices and rules as circular vertices. Hovering over a vertex shows additional information. On most devices, the mouse wheel lets the user zoom in and out of the graph. *Note:* the used javascript library does not do well with too many graph nodes, so plot selects the top 100 rules only (with a warning).

```
plot(rules, method = "graph", engine = "html")
```

The resulting interactive plot can be seen in the online version of this book.¹⁵

¹²https://mhahsler.github.io/Introduction_to_Data_Mining_R_Examples/book/association-analysis-basic-concepts-and-algorithms.html#interactive-visualizations

¹³https://mhahsler.github.io/Introduction_to_Data_Mining_R_Examples/book/association-analysis-basic-concepts-and-algorithms.html#interactive-visualizations

¹⁴https://mhahsler.github.io/Introduction_to_Data_Mining_R_Examples/book/association-analysis-basic-concepts-and-algorithms.html#interactive-visualizations

¹⁵https://mhahsler.github.io/Introduction_to_Data_Mining_R_Examples/book/association-analysis-basic-concepts-and-algorithms.html#interactive-visualizations

5.5.2.5 Interactive Rule Explorer

You can specify a rule set or a dataset. To explore rules that can be mined from iris, use: `ruleExplorer(iris)`

The rule explorer creates an interactive Shiny application that can be used locally or deployed on a server for sharing. A deployed version of the ruleExplorer is available here¹⁶ (using shinyapps.io¹⁷).

5.6 Exercises*

We will again use the Palmer penguin data for the exercises.

```
library(palmerpenguins)
head(penguins)
## # A tibble: 6 x 8
##   species island   bill_length_mm bill_depth_mm
##   <chr>   <chr>         <dbl>         <dbl>
## 1 Adelie  Torgersen         39.1           18.7
## 2 Adelie  Torgersen         39.5           17.4
## 3 Adelie  Torgersen         40.3            18
## 4 Adelie  Torgersen          NA            NA
## 5 Adelie  Torgersen         36.7           19.3
## 6 Adelie  Torgersen         39.3           20.6
## # i 4 more variables: flipper_length_mm <dbl>,
## #   body_mass_g <dbl>, sex <chr>, year <dbl>
```

1. Translate the penguin data into transaction data with:

```
trans <- transactions(penguins)
## Warning: Column(s) 1, 2, 3, 4, 5, 6, 7, 8 not logical or
## factor. Applying default discretization (see '?
## discretizeDF').
## Warning in discretize(x = c(2007, 2007, 2007, 2007, 2007, 2007, 2007, 2007, : The c
## Only unique breaks are used reducing the number of intervals. Look at ? discretiz
trans
## transactions in sparse format with
## 344 transactions (rows) and
## 22 items (columns)
```

Why does the conversion report warnings?

¹⁶https://mhahsler-apps.shinyapps.io/ruleExplorer_demo/

¹⁷<https://www.shinyapps.io/>

2. What do the following first three transactions mean?

```
inspect(trans[1:3])
##      items                                transactionID
## [1] {species=Adelie,
##      island=Torgersen,
##      bill_length_mm=[32.1,40.8),
##      bill_depth_mm=[18.3,21.5],
##      flipper_length_mm=[172,192),
##      body_mass_g=[3.7e+03,4.55e+03),
##      sex=male,
##      year=[2007,2008)}                                1
## [2] {species=Adelie,
##      island=Torgersen,
##      bill_length_mm=[32.1,40.8),
##      bill_depth_mm=[16.2,18.3),
##      flipper_length_mm=[172,192),
##      body_mass_g=[3.7e+03,4.55e+03),
##      sex=female,
##      year=[2007,2008)}                                2
## [3] {species=Adelie,
##      island=Torgersen,
##      bill_length_mm=[32.1,40.8),
##      bill_depth_mm=[16.2,18.3),
##      flipper_length_mm=[192,209),
##      body_mass_g=[2.7e+03,3.7e+03),
##      sex=female,
##      year=[2007,2008)}                                3
```

Next, use the `ruleExplorer()` function to analyze association rules created for the transaction data set.

1. Use the default settings for the parameters. Using the *Data Table*, what is the association rule with the highest lift. What does its LHS, RHS, support, confidence and lift mean?
2. Use the *Graph* visualization. Use select by id to highlight different species and different islands and then hover over some of the rules. What do you see?

Chapter 6

Association Analysis: Advanced Concepts

This chapter discusses a few advanced concepts of association analysis. First, we look at how categorical and continuous attributes are converted into items. Then we look at integrating item hierarchies into the analysis. Finally, sequence pattern mining is introduced.

Packages Used in this Chapter

```
pkgs <- c("arules", "arulesSequences", "tidyverse")  
  
pkgs_install <- pkgs[!(pkgs %in% installed.packages()[,"Package"])]  
if(length(pkgs_install)) install.packages(pkgs_install)
```

The packages used for this chapter are:

- *arules* (Hahsler et al. 2025)
- *arulesSequences* (Buchta and Hahsler 2024)
- *tidyverse* (Wickham 2023b)

6.1 Handling Categorical Attributes

Categorical attributes are nominal or ordinal variables. In R they are **factors** or **ordinal**. They are translated into a series of binary items (one for each level constructed as variable `name = level`). Items cannot represent order and this

ordered factors lose the order information. Note that nominal variables need to be encoded as factors (and not characters or numbers) before converting them into transactions.

For the special case of Boolean variables (`logical`), the `TRUE` value is converted into an item with the name of the variable and for the `FALSE` values no item is created.

We will give an example in the next section.

6.2 Handling Continuous Attributes

Continuous variables cannot directly be represented as items and need to be discretized first (see Discretization in Chapter 2). An item resulting from discretization might be `age>18` and the column contains only `TRUE` or `FALSE`. Alternatively, it can be a factor with levels `age<=18`, `50=>age>18` and `age>50`. These will be automatically converted into 3 items, one for each level. Discretization is described in functions `discretize()` and `discretizeDF()` to discretize all columns in a `data.frame`.

We give a short example using the iris dataset. We add an extra `logical` column to show how Boolean attributes are converted in to items.

```
data(iris)

## add a Boolean attribute
iris$Versicolor <- iris$Species == "versicolor"
head(iris)
##   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
##   Versicolor
## 1      FALSE
## 2      FALSE
## 3      FALSE
## 4      FALSE
## 5      FALSE
## 6      FALSE
```

The first step is to discretize continuous attributes (marked as `<dbl>` in the table above). We discretize the two Petal features.

```

library(tidyverse)
library(arules)

iris_disc <- iris %>%
  mutate(Petal.Length = discretize(Petal.Length,
                                   method = "frequency",
                                   breaks = 3,
                                   labels = c("short", "medium", "long")),
         Petal.Width = discretize(Petal.Width,
                                   method = "frequency",
                                   breaks = 2,
                                   labels = c("narrow", "wide"))
  )

head(iris_disc)
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5         short     narrow  setosa
## 2          4.9         3.0         short     narrow  setosa
## 3          4.7         3.2         short     narrow  setosa
## 4          4.6         3.1         short     narrow  setosa
## 5          5.0         3.6         short     narrow  setosa
## 6          5.4         3.9         short     narrow  setosa
##   Versicolor
## 1      FALSE
## 2      FALSE
## 3      FALSE
## 4      FALSE
## 5      FALSE
## 6      FALSE

```

Next, we convert the dataset into transactions.

```

trans <- transactions(iris_disc)
## Warning: Column(s) 1, 2 not logical or factor. Applying
## default discretization (see '? discretizeDF').
trans
## transactions in sparse format with
## 150 transactions (rows) and
## 15 items (columns)

```

The conversion creates a warning because there are still two undiscretized columns in the data. The warning indicates that the default discretization is used automatically.

```

itemLabels(trans)
## [1] "Sepal.Length=[4.3,5.4)" "Sepal.Length=[5.4,6.3)"
## [3] "Sepal.Length=[6.3,7.9]" "Sepal.Width=[2,2.9)"
## [5] "Sepal.Width=[2.9,3.2)" "Sepal.Width=[3.2,4.4)"
## [7] "Petal.Length=short"      "Petal.Length=medium"
## [9] "Petal.Length=long"       "Petal.Width=narrow"
## [11] "Petal.Width=wide"        "Species=setosa"
## [13] "Species=versicolor"     "Species=virginica"
## [15] "Versicolor"

```

We see that all continuous variables are discretized and the different ranges create an item. For example `Petal.Width` has the two items `Petal.Width=narrow` and `Petal.Width=wide`. The automatically discretized variables show intervals. `Sepal.Length=[4.3,5.4)` means that this item used for flowers with a sepal length between 4.3 and 5.4 cm.

The species is converted into three items, one for each class. The logical variable `Versicolor` created only a single item that is used when the variable is `TRUE`.

6.3 Handling Concept Hierarchies

Often an item hierarchy is available for transactions used for association rule mining. For example in a supermarket dataset items like “bread” and “beagle” might belong to the item group (category) “baked goods.” Transactions can store item hierarchies as additional columns in the `itemInfo` data.frame.

6.3.1 Aggregation

To perform analysis at a group level of the item hierarchy, `aggregate()` produces a new object with items aggregated to a given group level. A group-level item is present if one or more of the items in the group are present in the original object. If rules are aggregated, and the aggregation would lead to the same aggregated group item in the lhs and in the rhs, then that group item is removed from the lhs. Rules or itemsets, which are not unique after the aggregation, are also removed. Note also that the quality measures are not applicable to the new rules and thus are removed. If these measures are required, then aggregate the transactions before mining rules.

We use the Groceries data set in this example. It contains 1 month (30 days) of real-world point-of-sale transaction data from a typical local grocery outlet. The items are 169 products categories.

```
data("Groceries")
Groceries
## transactions in sparse format with
## 9835 transactions (rows) and
## 169 items (columns)
```

The dataset also contains two aggregation levels.

```
head(itemInfo(Groceries))
##           labels level2           level1
## 1 frankfurter sausage meat and sausage
## 2          sausage sausage meat and sausage
## 3      liver loaf sausage meat and sausage
## 4             ham sausage meat and sausage
## 5             meat sausage meat and sausage
## 6 finished products sausage meat and sausage
```

We aggregate to level1 stored in Groceries. All items with the same level2 label will become a single item with that name. This reduces the number of items to the 55 level2 categories

```
Groceries_level2 <- aggregate(Groceries, by = "level2")
Groceries_level2
## transactions in sparse format with
## 9835 transactions (rows) and
## 55 items (columns)
head(itemInfo(Groceries_level2)) ## labels are alphabetically sorted!
##           labels           level2           level1
## 1      baby food      baby food      canned food
## 2             bags             bags      non-food
## 3 bakery improver bakery improver processed food
## 4 bathroom cleaner bathroom cleaner      detergent
## 5             beef             beef meat and sausage
## 6             beer             beer             drinks
```

We can now compare an original transaction with the aggregated transaction.

```
inspect(head(Groceries, 3))
##      items
## [1] {citrus fruit,
##      semi-finished bread,
##      margarine,
##      ready soups}
## [2] {tropical fruit,
```

```
##      yogurt,
##      coffee}
## [3] {whole milk}
inspect(head(Groceries_level2, 3))
##      items
## [1] {bread and backed goods,
##      fruit,
##      soups/sauces,
##      vinegar/oils}
## [2] {coffee,
##      dairy produce,
##      fruit}
## [3] {dairy produce}
```

For example, citrus fruit in the first transaction was translated to the category fruit. Note that the order of items in a transaction is not important, so it might change during aggregation.

It is now easy to mine rules on the aggregated data.

```
rules <- apriori(Groceries_level2, support = 0.005)
## Apriori
##
## Parameter specification:
## confidence minval  smax  arem  aval originalSupport  maxtime
##           0.8     0.1   1 none  FALSE              TRUE     5
## support  minlen maxlen target  ext
##    0.005      1    10  rules TRUE
##
## Algorithmic control:
## filter tree heap memopt load sort verbose
##    0.1 TRUE TRUE  FALSE TRUE   2    TRUE
##
## Absolute minimum support count: 49
##
## set item appearances ...[0 item(s)] done [0.00s].
## set transactions ...[55 item(s), 9835 transaction(s)] done [0.00s].
## sorting and recoding items ... [47 item(s)] done [0.00s].
## creating transaction tree ... done [0.00s].
## checking subsets of size 1 2 3 4 5 6 done [0.00s].
## writing ... [243 rule(s)] done [0.00s].
## creating S4 object ... done [0.00s].
rules |> head(3, by = "support") |> inspect()
##      lhs                                rhs      support confidence coverage lift
## [1] {bread and backed goods,
```

```
##      cheese,
##      fruit}          => {dairy produce} 0.02481    0.8385  0.02959 1.893   244
## [2] {bread and backed goods,
##      cheese,
##      vegetables}      => {dairy produce} 0.02379    0.8239  0.02888 1.860   234
## [3] {cheese,
##      fruit,
##      vegetables}      => {dairy produce} 0.02267    0.8479  0.02674 1.914   223
```

You can add your own aggregation to an existing dataset by constructing the and iteminfo dataframe and adding it to the transactions. See ? `hierarchy` for details.

6.3.2 Multi-level Analysis

To analyze relationships between individual items and item groups at the same time, `addAggregate()` can be used to create a new transactions object which contains both, the original items and group-level items.

```
Groceries_multilevel <- addAggregate(Groceries, "level2")
Groceries_multilevel |> head(n=3) |> inspect()
##      items
## [1] {citrus fruit,
##      semi-finished bread,
##      margarine,
##      ready soups,
##      bread and backed goods*,
##      fruit*,
##      soups/sauces*,
##      vinegar/oils*}
## [2] {tropical fruit,
##      yogurt,
##      coffee,
##      coffee*,
##      dairy produce*,
##      fruit*}
## [3] {whole milk,
##      dairy produce*}
```

The added group-level items are marked with an * after the name. Now we can mine rules including items from multiple levels.

```

rules <- apriori(Groceries_multilevel,
  parameter = list(support = 0.005))
## Apriori
##
## Parameter specification:
## confidence minval smax arem aval originalSupport maxtime
##          0.8    0.1    1 none FALSE          TRUE      5
## support minlen maxlen target  ext
##    0.005      1     10  rules TRUE
##
## Algorithmic control:
## filter tree heap memopt load sort verbose
##    0.1 TRUE TRUE  FALSE TRUE    2    TRUE
##
## Absolute minimum support count: 49
##
## set item appearances ...[0 item(s)] done [0.00s].
## set transactions ...[224 item(s), 9835 transaction(s)] done [0.01s].
## sorting and recoding items ... [167 item(s)] done [0.00s].
## creating transaction tree ... done [0.00s].
## checking subsets of size 1 2 3 4 5 6 7 8 done [0.04s].
## writing ... [21200 rule(s)] done [0.00s].
## creating S4 object ... done [0.01s].
rules
## set of 21200 rules

```

Mining rules with group-level items added will create many spurious rules of the type

```
item A => group of item A
```

with a confidence of 1. This will also happen if you mine itemsets. `filterAggregate()` can be used to filter these spurious rules or itemsets.

```

rules <- filterAggregate(rules)
rules
## set of 838 rules
rules |> head(n = 3, by = "lift") |> inspect()
##      lhs                                rhs      support confidence cov
## [1] {whole milk,
##      whipped/sour cream,
##      bread and backed goods*,
##      cheese*}                               => {vegetables*} 0.005186    0.8095 0.0
## [2] {sausage,
##      poultry*}                               => {vegetables*} 0.005084    0.8065 0.0

```



```
## [3] {other vegetables,
##      soda,
##      fruit*,
##      sausage*}          => {bread and backed goods*} 0.005287    0.8525 0.006202 2.467
```

Using multi-level mining can reduce the number of rules and help to analyze if customers differentiate between products in a group.

6.4 Sequential Patterns

The frequent sequential pattern mining algorithm cSPADE (Zaki 2000) is implemented in the `arules` extension package `arulesSequences`.

Sequential pattern mining starts with sequences of events. Each sequence is identified by a sequence ID and each event is a set of items that happen together. The order of events is specified using event IDs. The goal is to find subsequences of items in events that follow each other frequently. These are called frequent sequential pattern.

We will look at a small example dataset that comes with the package `arulesSequences`.

```
library(arulesSequences)
##
## Attaching package: 'arulesSequences'
## The following object is masked from 'package:arules':
##
##      itemsets
data(zaki)

inspect(zaki)
##      items      sequenceID eventID SIZE
## [1] {C, D}          1         10     2
## [2] {A, B, C}       1         15     3
## [3] {A, B, F}       1         20     3
## [4] {A, C, D, F}    1         25     4
## [5] {A, B, F}       2         15     3
## [6] {E}            2         20     1
## [7] {A, B, F}       3         10     3
## [8] {D, G, H}       4         10     3
## [9] {B, F}          4         20     2
## [10] {A, G, H}      4         25     3
```

The dataset contains four sequences (see `sequenceID`) and the event IDs are integer numbers to provide the order events in a sequence. In `arulesSequences`,

this set of sequences is implemented as a regular transaction set, where each transaction is an event. The temporal information is added as extra columns to the transaction's `transactionInfo()` data.frame.

Mine frequent sequence patterns using `cspade` is very similar to using `apriori`. Here we set support so we will find patterns that occur in 50% of the sequences.

```
fsp <- cspade(zaki, parameter = list(support = .5))
fsp |> inspect()
##      items support
##  1 <{A}>      1.00
##  2 <{B}>      1.00
##  3 <{D}>      0.50
##  4 <{F}>      1.00
##  5 <{A,
##     F}>      0.75
##  6 <{B,
##     F}>      1.00
##  7 <{D},
##     {F}>      0.50
##  8 <{D},
##     {B,
##        F}>      0.50
##  9 <{A,
##     B,
##     F}>      0.75
## 10 <{A,
##     B}>      0.75
## 11 <{D},
##     {B}>      0.50
## 12 <{B},
##     {A}>      0.50
## 13 <{D},
##     {A}>      0.50
## 14 <{F},
##     {A}>      0.50
## 15 <{D},
##     {F},
##     {A}>      0.50
## 16 <{B,
##     F},
##     {A}>      0.50
## 17 <{D},
##     {B,
##        F},
##     {A}>      0.50
```

```
## 18 <{D},
##      {B},
##      {A}> 0.50
##
```

For example, pattern 17 shows that D in an event, it is often followed by an event by containing B and F which in turn is followed by an event containing A.

The cspade algorithm supports many additional parameters to control gaps and windows. Details can be found in the manual page for `cspade`.

Rules, similar to regular association rules can be generated from frequent sequence patterns using `ruleInduction()`.

```
rules <- ruleInduction(fsp, confidence = .8)
rules |> inspect()
##   lhs      rhs      support confidence lift
## 1 <{D}> => <{F}>      0.5          1    1
## 2 <{D}> => <{B,      0.5          1    1
##      F}>
## 3 <{D}> => <{B}>      0.5          1    1
## 4 <{D}> => <{A}>      0.5          1    1
## 5 <{D},
##   {F}> => <{A}>      0.5          1    1
## 6 <{D},
##   {B,
##   F}> => <{A}>      0.5          1    1
## 7 <{D},
##   {B}> => <{A}>      0.5          1    1
##
```

The usual measures of confidence and lift are used.

Chapter 7

Cluster Analysis

This chapter introduces cluster analysis using K-means, hierarchical clustering and DBSCAN. We will discuss how to choose the number of clusters and how to evaluate the quality clusterings. In addition, we will introduce more clustering algorithms and how clustering is influenced by outliers.

The corresponding chapter of the data mining textbook is available online: Chapter 7: Cluster Analysis: Basic Concepts and Algorithms.¹

Packages Used in this Chapter

```
pkgs <- c("cluster", "dbscan", "e1071", "factoextra", "fpc",  
         "GGally", "kernlab", "mclust", "mlbench", "scatterpie",  
         "seriation", "tidyverse")  
  
pkgs_install <- pkgs[!(pkgs %in% installed.packages()[,"Package"])]  
if(length(pkgs_install)) install.packages(pkgs_install)
```

The packages used for this chapter are:

- *cluster* (Maechler et al. 2025)
- *dbscan* (Hahsler and Piekenbrock 2025)
- *e1071* (Meyer et al. 2024)
- *factoextra* (Kassambara and Mundt 2020)
- *fpc* (Hennig 2024)
- *GGally* (Schloerke et al. 2025)

¹https://mhahsler.github.io/Introduction_to_Data_Mining_R_Examples/DM_chapters/ch7_clustering.pdf

- *kernlab* (Karatzoglou, Smola, and Hornik 2024)
- *mclust* (Fraley, Raftery, and Scrucca 2024)
- *mlbench* (Leisch and Dimitriadou 2024)
- *scatterpie* (Yu 2025)
- *seriation* (Hahsler, Buchta, and Hornik 2025)
- *tidyverse* (Wickham 2023b)

7.1 Overview

Cluster analysis or clustering² is the task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense) to each other than to those in other groups (clusters).

Clustering is also called unsupervised learning, because it tries to directly learn the structure of the data and does not rely on the availability of a correct answer or class label as supervised learning does. Clustering is often used for exploratory analysis or to preprocess data by grouping.

You can read the free sample chapter from the textbook (Tan, Steinbach, and Kumar 2005): Chapter 7. Cluster Analysis: Basic Concepts and Algorithms³

7.1.1 Data Preparation

```
library(tidyverse)
```

We will use here a small and very clean toy dataset called Ruspini which is included in the R package **cluster**.

```
data(ruspini, package = "cluster")
```

The Ruspini data set, consisting of 75 points in four groups that is popular for illustrating clustering techniques. It is a very simple data set with well separated clusters. The original dataset has the points ordered by group. We can shuffle the data (rows) using `sample_frac` which samples by default 100%.

```
ruspini <- as_tibble(ruspini) |>
  sample_frac()
ruspini
## # A tibble: 75 x 2
```

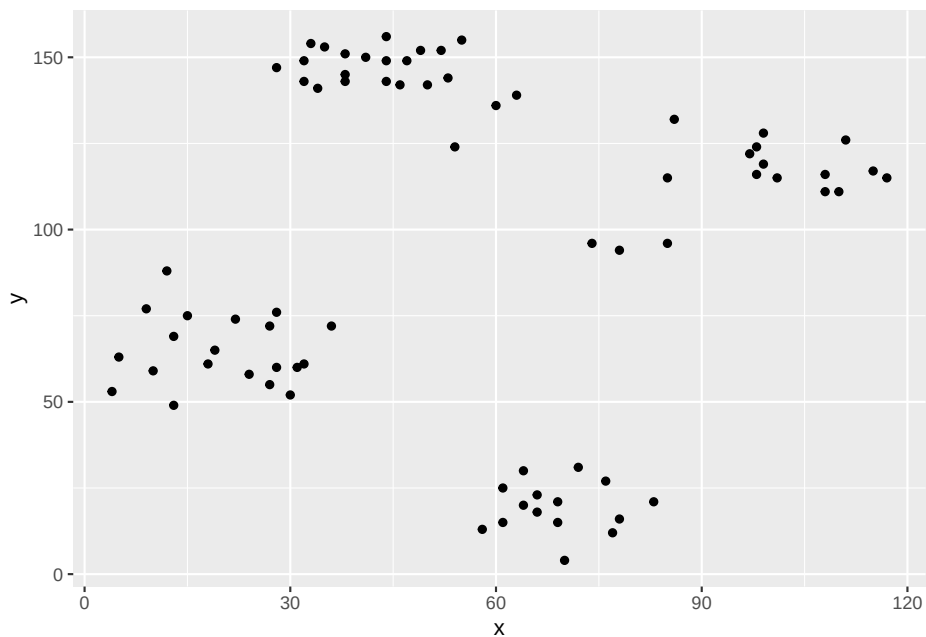
²https://en.wikipedia.org/wiki/Cluster_analysis

³https://mhahsler.github.io/Introduction_to_Data_Mining_R_Examples/DM_chapters/ch7_clustering.pdf

```
##      x      y
##    <int> <int>
##  1    78    94
##  2     9    77
##  3   117   115
##  4    24    58
##  5    85    96
##  6    30    52
##  7    60   136
##  8    74    96
##  9   108   111
## 10    69    21
## # i 65 more rows
```

7.1.2 Data cleaning

```
ggplot(ruspini, aes(x = x, y = y)) + geom_point()
```



```
summary(ruspini)
##      x      y
##  Min.   : 4.0   Min.   : 4.0
##  1st Qu.: 31.5  1st Qu.: 56.5
```

```
## Median : 52.0 Median : 96.0
## Mean   : 54.9 Mean   : 92.0
## 3rd Qu.: 76.5 3rd Qu.:141.5
## Max.    :117.0 Max.    :156.0
```

For most clustering algorithms it is necessary to handle missing values and outliers (e.g., remove the observations). For details see Section “Outlier removal” below. This data set has not missing values or strong outlier and looks like it has some very clear groups.ⁱ

7.1.3 Scale data

Clustering algorithms use distances and the variables with the largest number range will dominate distance calculation. The summary above shows that this is not an issue for the Ruspini dataset with both, x and y, being roughly between 0 and 150. Most data analysts will still scale each column in the data to zero mean and unit standard deviation (z-scores⁴).

Note: The standard `scale()` function scales a whole data matrix so we implement a function for a single vector and apply it to all numeric columns.

```
## I use this till tidyverse implements a scale function
scale_numeric <- function(x) {
  x |> mutate(across(where(is.numeric),
                        function(y) (y - mean(y, na.rm = TRUE)) / sd(y, na.rm = TRUE)))
}
```

```
ruspini_scaled <- ruspini |>
  scale_numeric()
summary(ruspini_scaled)
##           x           y
## Min.      :-1.6681  Min.      :-1.8074
## 1st Qu.   :-0.7665  1st Qu.   :-0.7295
## Median    :-0.0944  Median    : 0.0816
## Mean      : 0.0000  Mean      : 0.0000
## 3rd Qu.   : 0.7088  3rd Qu.   : 1.0158
## Max.      : 2.0366  Max.      : 1.3136
```

After scaling, most z-scores will fall in the range $[-3, 3]$ (z-scores are measured in standard deviations from the mean), where 0 means average.

⁴https://en.wikipedia.org/wiki/Standard_score

7.2 K-means

k-means⁵ implicitly assumes Euclidean distances. We use $k = 4$ clusters and run the algorithm 10 times with random initialized centroids. The best result is returned.

```
km <- kmeans(ruspini_scaled, centers = 4, nstart = 10)
km
## K-means clustering with 4 clusters of sizes 20, 15, 17, 23
##
## Cluster means:
##      x      y
## 1 -1.1386 -0.5560
## 2  0.4607 -1.4912
## 3  1.4194  0.4693
## 4 -0.3595  1.1091
##
## Clustering vector:
## [1] 3 1 3 1 3 1 4 3 3 2 2 4 4 3 1 4 3 4 4 3 2 1 1 3 1 2 1 1
## [29] 4 4 2 4 4 2 2 1 4 2 4 4 4 1 2 2 3 4 3 3 3 4 4 4 2 2 2 4
## [57] 1 1 3 2 2 3 1 1 4 4 1 1 3 4 1 4 3 1 1
##
## Within cluster sum of squares by cluster:
## [1] 2.705 1.082 3.641 2.659
## (between_SS / total_SS =  93.2 %)
##
## Available components:
##
## [1] "cluster"      "centers"      "totss"
## [4] "withinss"     "tot.withinss" "betweenss"
## [7] "size"         "iter"         "ifault"
```

km is an R object implemented as a list.

```
str(km)
## List of 9
## $ cluster      : int [1:75] 3 1 3 1 3 1 4 3 3 2 ...
## $ centers      : num [1:4, 1:2] -1.139 0.461 1.419 -0.36 -0.556 ...
## ..- attr(*, "dimnames")=List of 2
## .. ..$ : chr [1:4] "1" "2" "3" "4"
## .. ..$ : chr [1:2] "x" "y"
## $ totss       : num 148
## $ withinss    : num [1:4] 2.71 1.08 3.64 2.66
```

⁵https://en.wikipedia.org/wiki/K-means_clustering

```
## $ tot.withinss: num 10.1
## $ betweenss   : num 138
## $ size        : int [1:4] 20 15 17 23
## $ iter        : int 2
## $ ifault      : int 0
## - attr(*, "class")= chr "kmeans"
```

The clustering vector is just a list element containing the cluster assignment for each data row and can be accessed using `km$cluster`. I add the cluster assignment as a column to the original dataset (I make it a factor since it represents a nominal label).

```
ruspini_clustered <- ruspini |>
  add_column(cluster = factor(km$cluster))
ruspini_clustered
## # A tibble: 75 x 3
##       x     y cluster
##   <int> <int> <fct>
## 1    78    94 3
## 2     9    77 1
## 3   117   115 3
## 4    24    58 1
## 5    85    96 3
## 6    30    52 1
## 7    60   136 4
## 8    74    96 3
## 9   108   111 3
## 10   69    21 2
## # i 65 more rows
ggplot(ruspini_clustered, aes(x = x, y = y)) +
  geom_point(aes(color = cluster))
```



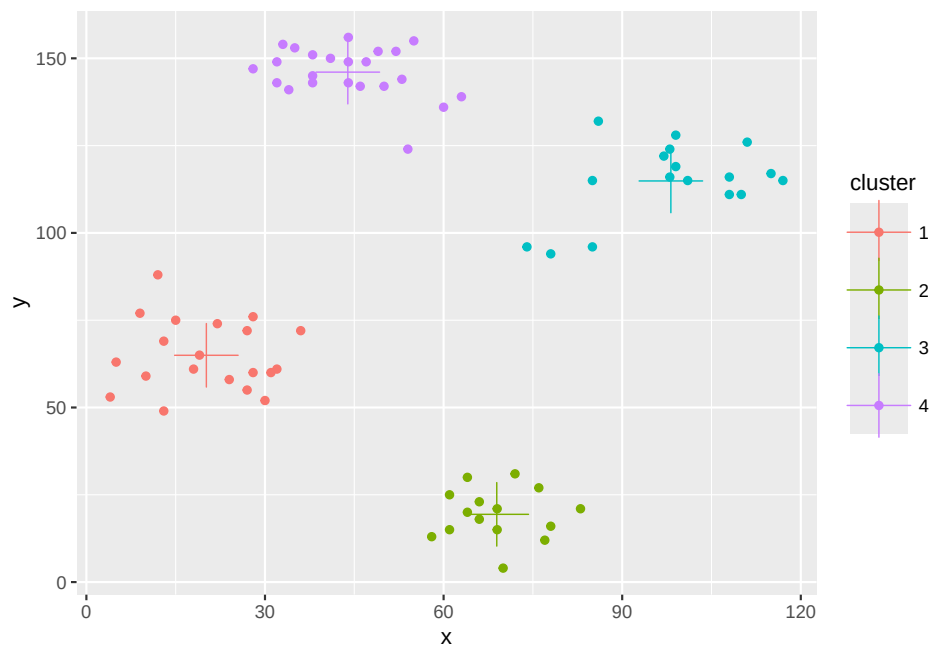
Add the centroids to the plot. The centroids are scaled, so we need to unscale them to plot them over the original data. The second `geom_points` uses not the original data but specifies the centroids as its dataset.

```
unscale <- function(x, original_data) {
  if (ncol(x) != ncol(original_data))
    stop("Function needs matching columns!")
  x * matrix(apply(original_data, MARGIN = 2, sd, na.rm = TRUE),
             byrow = TRUE, nrow = nrow(x), ncol = ncol(x)) +
    matrix(apply(original_data, MARGIN = 2, mean, na.rm = TRUE),
           byrow = TRUE, nrow = nrow(x), ncol = ncol(x))
}

centroids <- km$centers %>%
  unscale(original_data = ruspini) %>%
  as_tibble(rownames = "cluster")

centroids
## # A tibble: 4 x 3
##   cluster    x    y
##   <chr>   <dbl> <dbl>
## 1 1      20.2  64.9
## 2 2      68.9  19.4
## 3 3      98.2 115.
## 4 4      43.9 146.
ggplot(ruspini_clustered, aes(x = x, y = y)) +
```

```
geom_point(aes(color = cluster)) +  
geom_point(data = centroids, aes(x = x, y = y, color = cluster),  
           shape = 3, size = 10)
```



The **factoextra** package provides also a good visualization with object labels and ellipses for clusters.

```
library(factoextra)  
fviz_cluster(km, data = ruspini_scaled, centroids = TRUE,  
             repel = TRUE, ellipse.type = "norm")
```

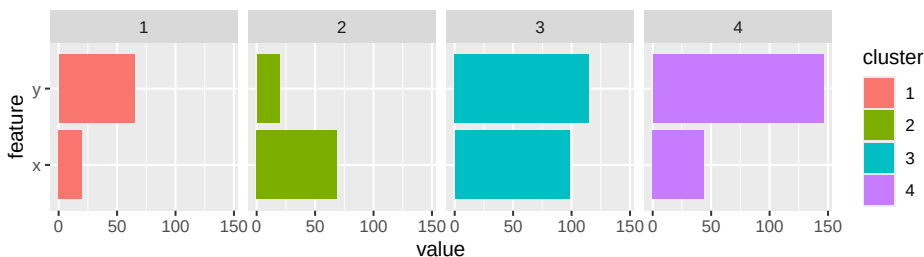


7.2.1 Inspect Clusters

We inspect the clusters created by the 4-cluster k-means solution. The following code can be adapted to be used for other clustering methods.

Inspect the centroids with horizontal bar charts organized by cluster. To group the plots by cluster, we have to change the data format to the “long”-format using a pivot operation. I use colors to match the clusters in the scatter plots.

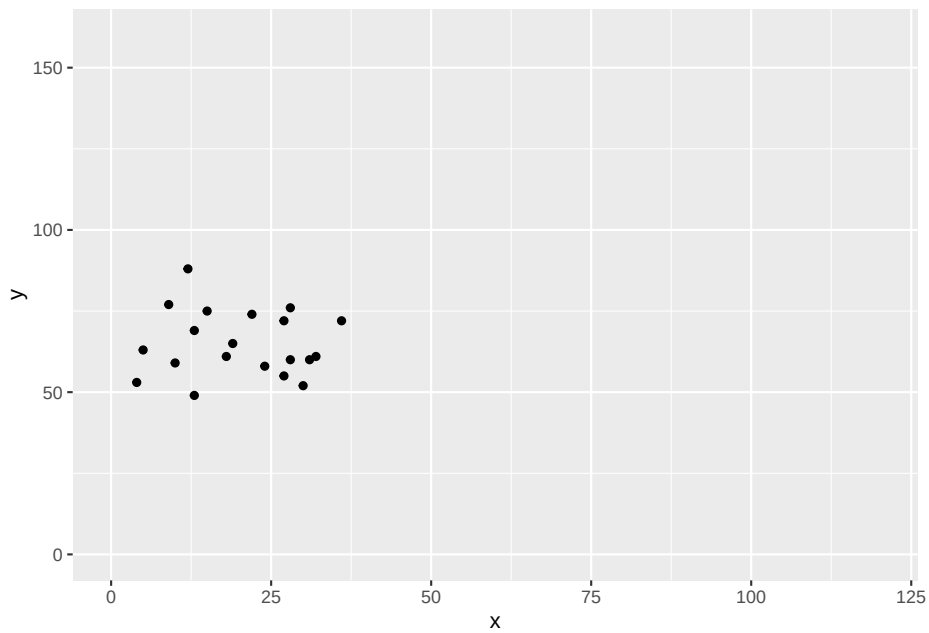
```
ggplot(pivot_longer(centroids,
  cols = c(x, y),
  names_to = "feature"),
  #aes(x = feature, y = value, fill = cluster)) +
  aes(x = value, y = feature, fill = cluster)) +
  geom_bar(stat = "identity") +
  facet_grid(cols = vars(cluster))
```



7.2.2 Extract a Single Cluster

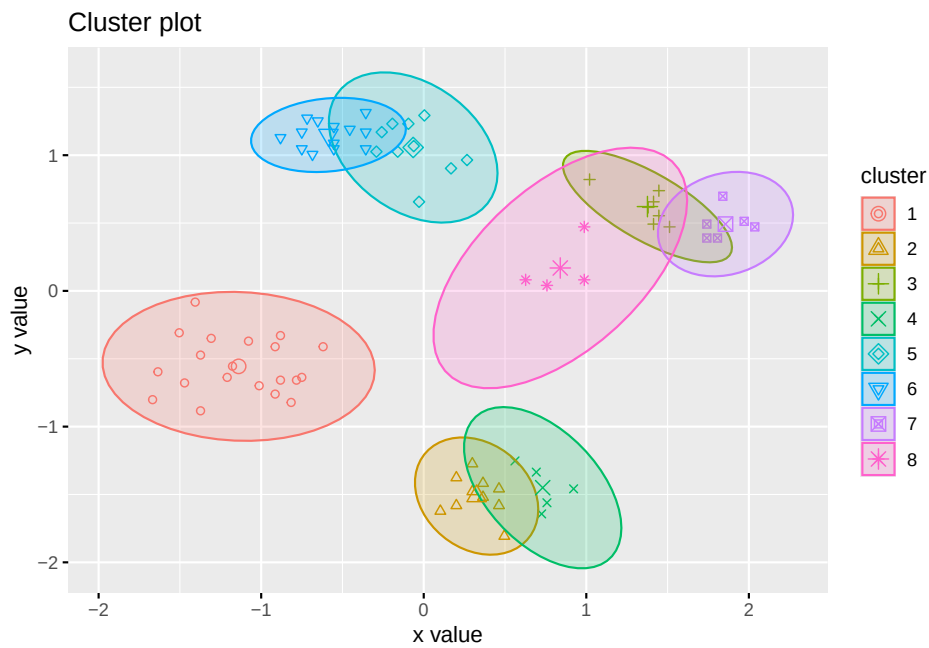
You need is to filter the rows corresponding to the cluster index. The next example calculates summary statistics and then plots all data points of cluster 1.

```
cluster1 <- ruspini_clustered |>
  filter(cluster == 1)
cluster1
## # A tibble: 20 x 3
##       x     y cluster
##   <int> <int> <fct>
## 1     9    77 1
## 2    24    58 1
## 3    30    52 1
## 4    27    55 1
## 5    28    76 1
## 6    22    74 1
## 7    15    75 1
## 8    28    60 1
## 9    27    72 1
## 10   32    61 1
## 11   19    65 1
## 12   36    72 1
## 13   31    60 1
## 14   10    59 1
## 15   13    69 1
## 16    5    63 1
## 17   18    61 1
## 18   13    49 1
## 19    4    53 1
## 20   12    88 1
summary(cluster1)
##           x             y      cluster
## Min.      : 4.0   Min.   :49.0   1:20
## 1st Qu.:12.8   1st Qu.:58.8   2: 0
## Median :20.5   Median :62.0   3: 0
## Mean     :20.1   Mean    :65.0   4: 0
## 3rd Qu.:28.0   3rd Qu.:72.5
## Max.     :36.0   Max.    :88.0
ggplot(cluster1, aes(x = x, y = y)) + geom_point() +
  coord_cartesian(xlim = c(0, 120), ylim = c(0, 160))
```



What happens if we try to cluster with 8 centers?

```
fviz_cluster(kmeans(ruspini_scaled, centers = 8), data = ruspini_scaled,  
             centroids = TRUE, geom = "point", ellipse.type = "norm")
```



7.3 Agglomerative Hierarchical Clustering

Hierarchical clustering starts with a distance matrix. `dist()` defaults to `method = "euclidean"`.

Notes:

- Distance matrices become very large quickly (the space complexity is $O(n^2)$ where n is the number of data points). It is only possible to calculate and store the matrix for small to medium data sets (maybe a few hundred thousand data points) in main memory. If your data is too large then you can use sampling to reduce the number of points to cluster.
- The data needs to be scaled since we compute distances.

7.3.1 Creating a Dendrogram

```
d <- dist(ruspini_scaled)
```

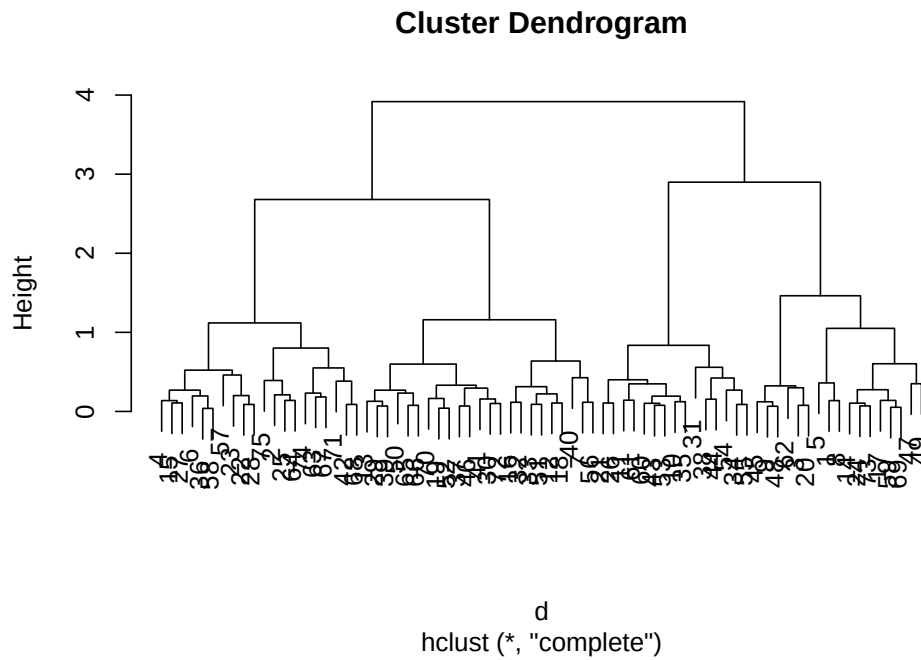
`hclust()` implements agglomerative hierarchical clustering⁶. The linkage function defines how the distance between two clusters (sets of points) is calculated. We specify the complete-link method where the distance between two groups is measured by the distance between the two farthest apart points in the two groups.

```
hc <- hclust(d, method = "complete")
```

Hierarchical clustering does not return cluster assignments but a dendrogram object which shows how the objects on the x-axis are successively joined together when going up along the y-axis. The y-axis represents the distance at which groups of points are joined together. The standard plot function displays the dendrogram.

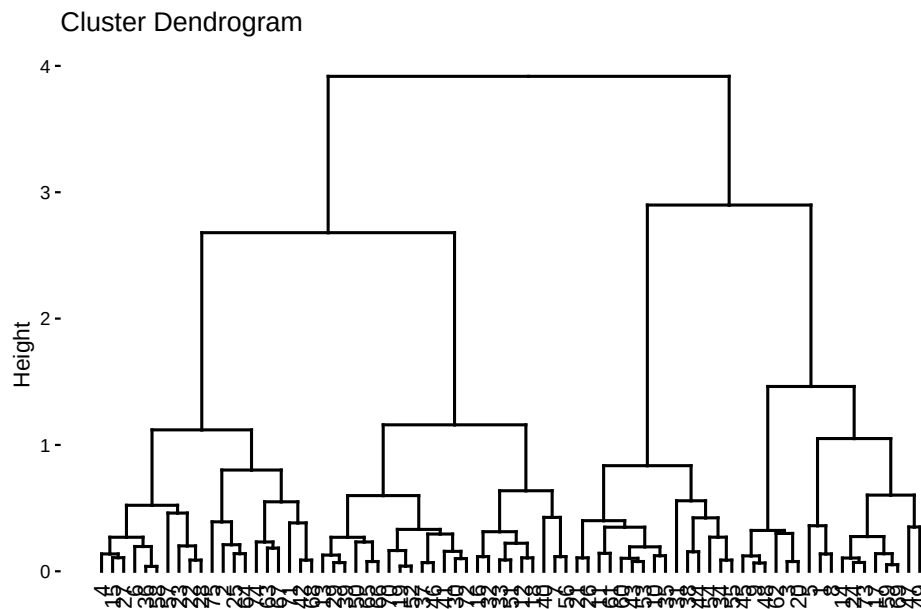
```
plot(hc)
```

⁶https://en.wikipedia.org/wiki/Hierarchical_clustering



The package `factoextra` provides a `ggplot` function to plot dendrograms.

```
fviz_dend(hc)
```



More plotting options for dendrograms, including plotting parts of large den-

dendrograms can be found here.⁷

7.3.2 Extracting a Partitional Clustering

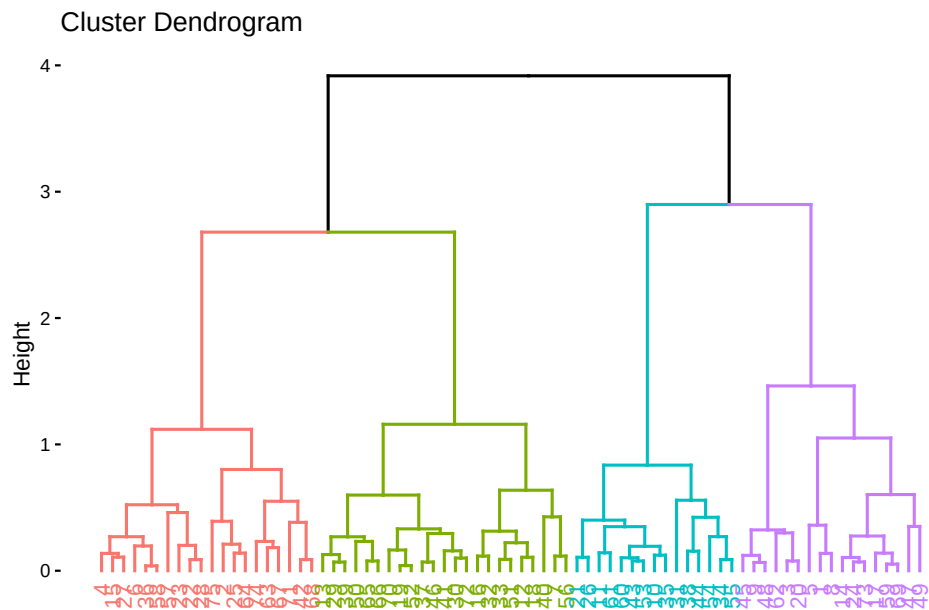
Partitional clusterings with cluster assignments can be extracted from a dendrogram by cutting the dendrogram horizontally using `cutree()`. Here we cut it into four parts and add the cluster id to the data.

```
clusters <- cutree(hc, k = 4)
cluster_complete <- ruspini_scaled |>
  add_column(cluster = factor(clusters))
cluster_complete
## # A tibble: 75 x 3
##       x      y cluster
##   <dbl> <dbl> <fct>
## 1  0.758  0.0405 1
## 2 -1.50  -0.309 2
## 3  2.04   0.472 1
## 4 -1.01  -0.699 2
## 5  0.987  0.0816 1
## 6 -0.816 -0.822 2
## 7  0.168  0.903 3
## 8  0.627  0.0816 1
## 9  1.74   0.390 1
## 10 0.463 -1.46 4
## # i 65 more rows
```

The 4 different clusters can be shown in the dendrogram using different colors.

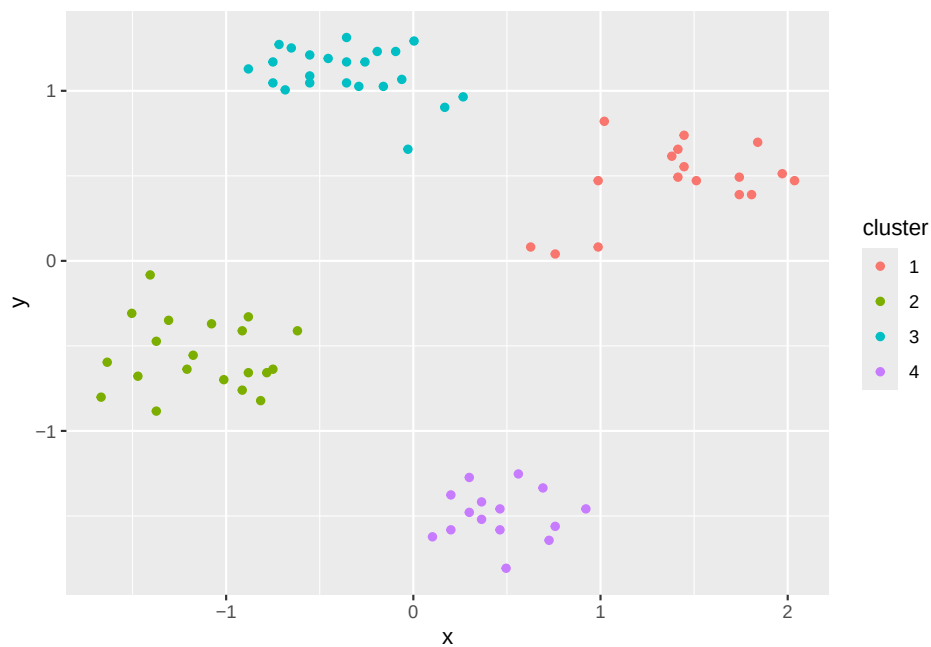
```
fviz_dend(hc, k = 4)
```

⁷<https://rpubs.com/gaston/dendrograms>



We can also color in the scatterplot.

```
ggplot(cluster_complete, aes(x, y, color = cluster)) +  
  geom_point()
```



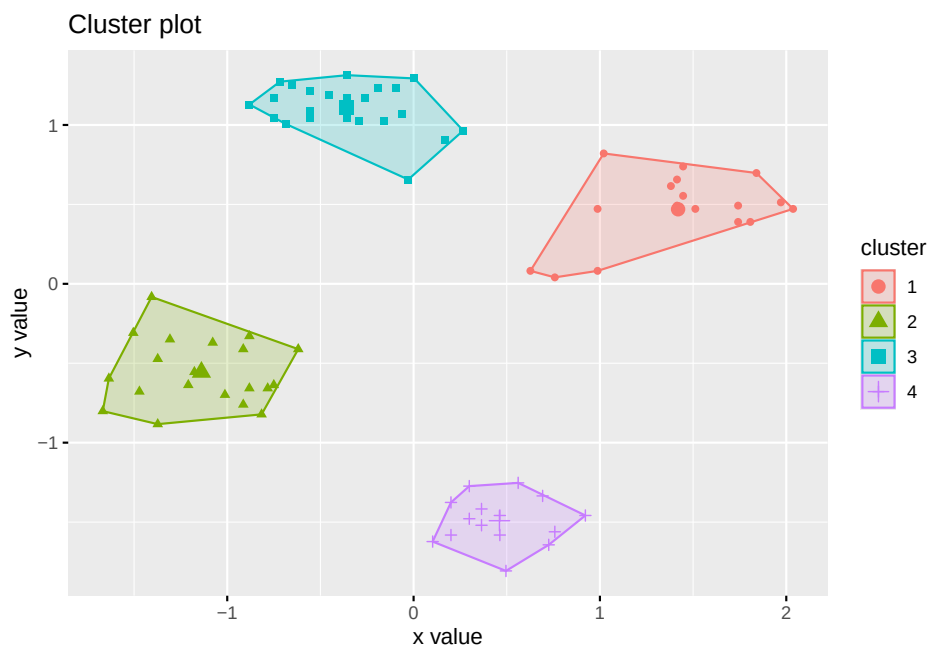
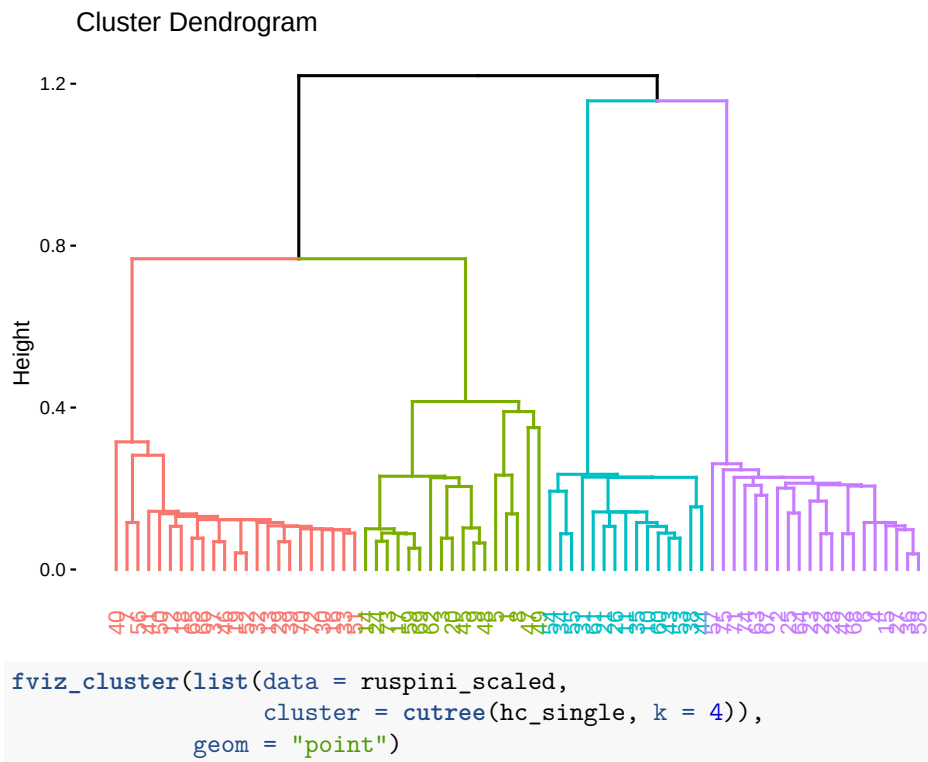
The package **factoextra** provides an alternative visualization. Since it needs a partition object as the first argument, we create one as a simple list.

```
fviz_cluster(list(data = ruspini_scaled,
                  cluster = cutree(hc, k = 4)),
             geom = "point")
```



Here are the results if we use the single-link method to join clusters. Single-link measures the distance between two groups by the distance between the two closest points from the two groups.

```
hc_single <- hclust(d, method = "single")
fviz_dend(hc_single, k = 4)
```



The most important difference between complete-link and single-link is that

complete-link prefers globular clusters and single-link shows chaining (see the staircase pattern in the dendrogram).

7.4 DBSCAN

DBSCAN⁸ stands for “Density-Based Spatial Clustering of Applications with Noise.” It groups together points that are closely packed together and treats points in low-density regions as outliers. DBSCAN is implemented in package `dbscan`.

```
library(dbscan)
##
## Attaching package: 'dbscan'
## The following object is masked from 'package:stats':
##
##      as.dendrogram
```

7.4.1 DBSCAN Parameters

DBSCAN has two parameters that interact with each other. Changing one typically means that the other one also has to be adjusted.

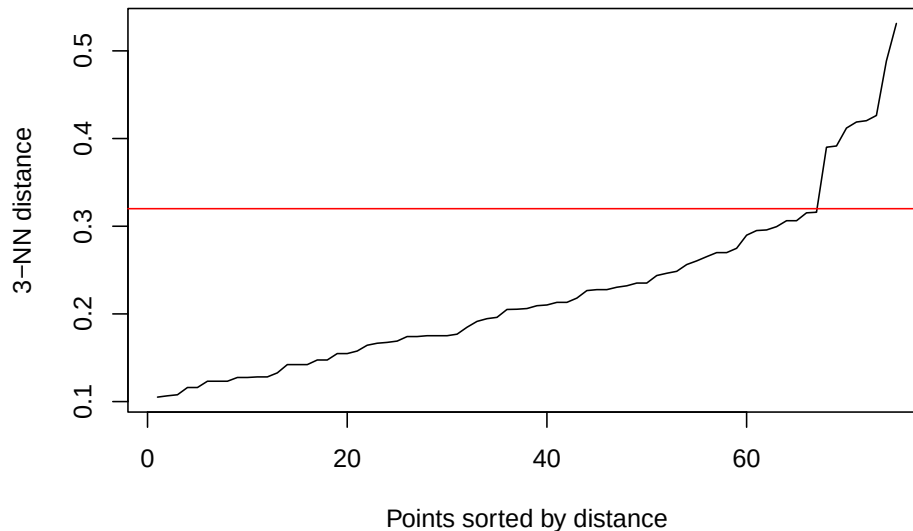
`minPts` defines how many points are needed in the epsilon neighborhood to make a point a core point for a cluster. It is often chosen as a smoothing parameter, where larger values smooth the density estimation more but also ignore smaller structures with less than `minPts`.

`eps` is the radius of the epsilon neighborhood around a point in which the number of points is counted.

Users typically first select `minPts`. We use here `minPts = 4`. To decide on epsilon, the knee in the kNN distance plot is often used. All points are sorted by their kNN distance. The idea is that points with a low kNN distance are located in a dense area which should become a cluster. Points with a large kNN distance are in a low density area and most likely represent outliers and noise.

```
kNNdistplot(ruspini_scaled, minPts = 4)
abline(h = .32, col = "red")
```

⁸<https://en.wikipedia.org/wiki/DBSCAN>



Note that `minPts` contains the point itself, while the `k`-nearest neighbor distance calculation does not. The plot uses therefor `k = minPts - 1`.

The knee is visible around `eps = .32` (shown by the manually added red line). The points to the left of the intersection of the `k`-nearest neighbor distance line and the red line are the points that will be core points during clustering at the specified parameters.

7.4.2 Cluster using DBSCAN

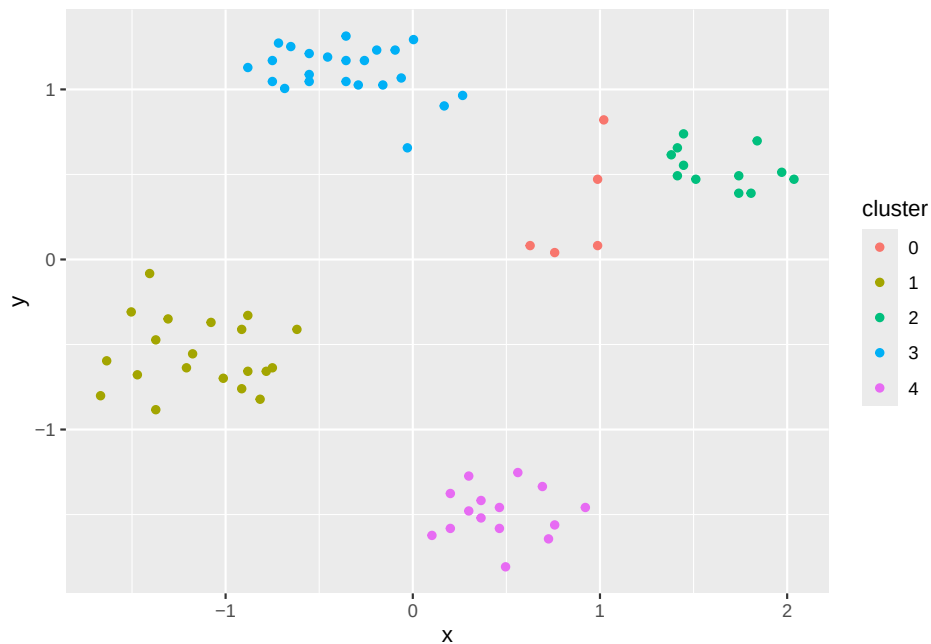
Clustering with `dbscan()` returns a `dbscan` object.

```
db <- dbscan(ruspini_scaled, eps = .32, minPts = 4)
db
## DBSCAN clustering for 75 objects.
## Parameters: eps = 0.32, minPts = 4
## Using euclidean distances and borderpoints = TRUE
## The clustering contains 4 cluster(s) and 5 noise points.
##
##  0  1  2  3  4
##  5 20 12 23 15
##
## Available fields: cluster, eps, minPts, metric,
##                  borderPoints
str(db)
## List of 5
## $ cluster      : int [1:75] 0 1 2 1 0 1 3 0 2 4 ...
## $ eps          : num 0.32
```

```
## $ minPts      : num 4
## $ metric      : chr "euclidean"
## $ borderPoints: logi TRUE
## - attr(*, "class")= chr [1:2] "dbscan_fast" "dbscan"
```

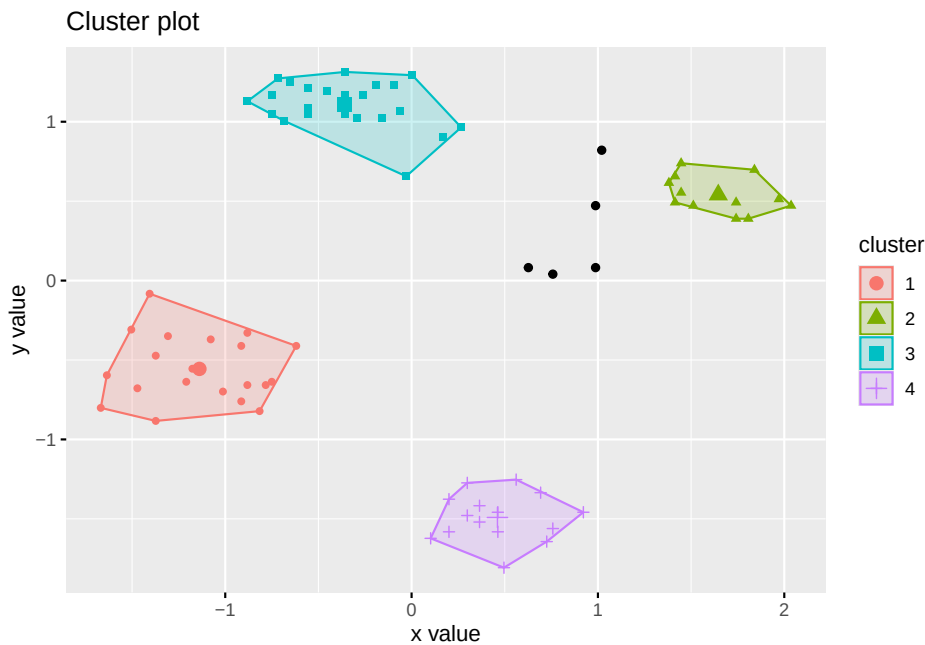
The `cluster` element is the cluster assignment with cluster labels. The special cluster label 0 means that the point is noise and not assigned to a cluster.

```
ggplot(ruspini_scaled |> add_column(cluster = factor(db$cluster)),
  aes(x, y, color = cluster)) + geom_point()
```



DBSCAN found 5 noise points. `fviz_cluster()` can be used to create a ggplot visualization.

```
fviz_cluster(db, ruspini_scaled, geom = "point")
```

Different values for `minPts` and `eps` can lead to vastly different clusterings. Often, a misspecification leads to all points being noise points or a single cluster. Also, if the clusters are not well separated, then DBSCAN has a hard time splitting them.

7.5 Cluster Evaluation

7.5.1 Unsupervised Cluster Evaluation

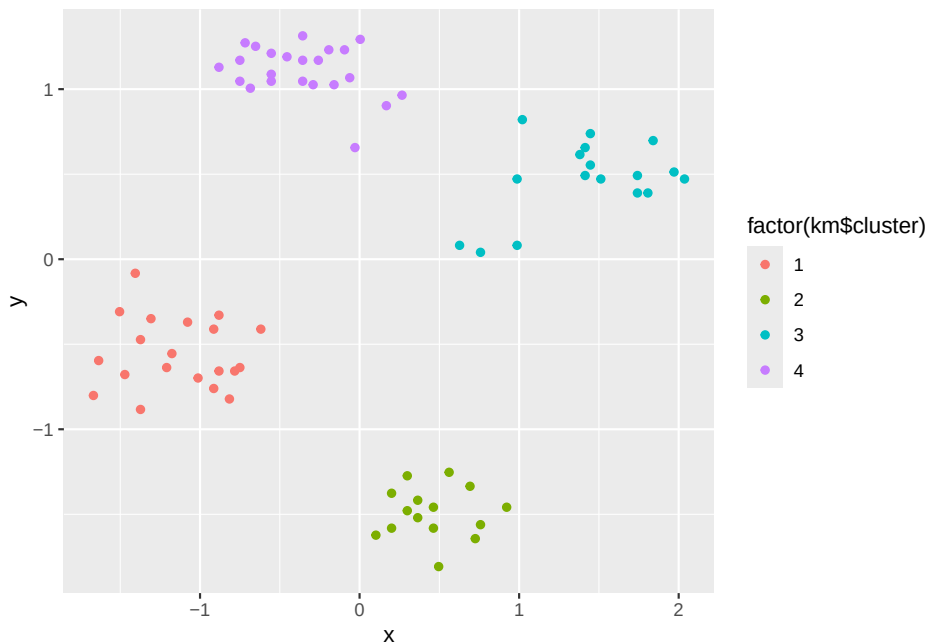
This is also often called internal cluster evaluation since it does not use extra external labels.

We will evaluate the k-means clustering from above.

7.5.1.1 Visual Methods

For data in 2 dimensions, a scatter plot will show if the clusters look right.

```
ggplot(ruspini_scaled,
  aes(x, y, color = factor(km$cluster))) +
  geom_point()
```



For data with more dimensions or with columns containing nominal/ordinal data, we cannot look at the scatter plot. However, we can visualize a reordered similarity/dissimilarity matrix.

First, we calculate a distance matrix and inspect the distance matrix between the first 5 objects.

```
d <- dist(ruspini_scaled)

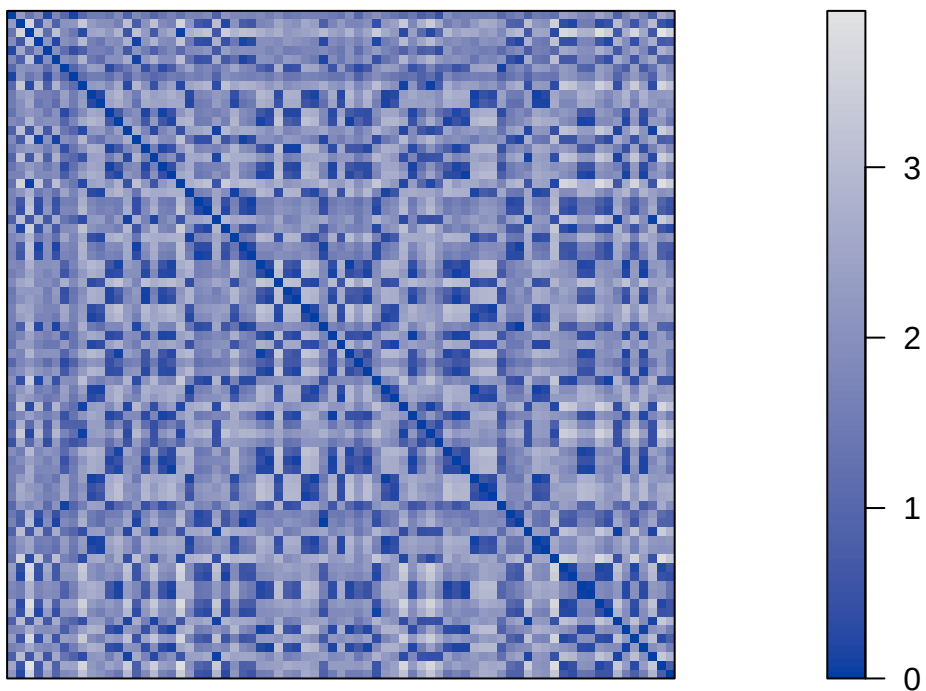
as.matrix(d)[1:5, 1:5]
##           1      2      3      4      5
## 1 0.0000 2.2889 1.349 1.9185 0.2331
## 2 2.2889 0.0000 3.626 0.6277 2.5220
## 3 1.3493 3.6256 0.000 3.2658 1.1193
## 4 1.9185 0.6277 3.266 0.0000 2.1467
## 5 0.2331 2.5220 1.119 2.1467 0.0000
```

Matrix visualizations with reordering are provided in package **seriation**. Matrix visualization creates a false-color image where each value in the matrix as a pixel with the color representing the value.

```
library(seriation)
##
## Attaching package: 'seriation'
## The following object is masked from 'package:lattice':
##
```

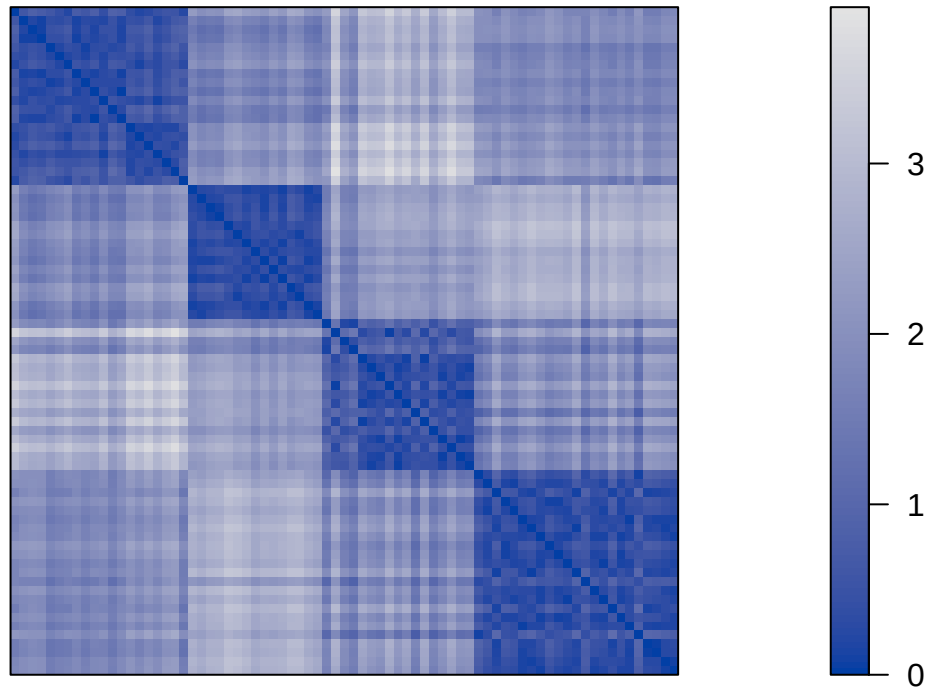
```
##      panel.lines  
pimage(d, main = "Unordered")
```

Unordered



```
pimage(d, order = order(km$cluster), main = "Reordered by cluster")
```

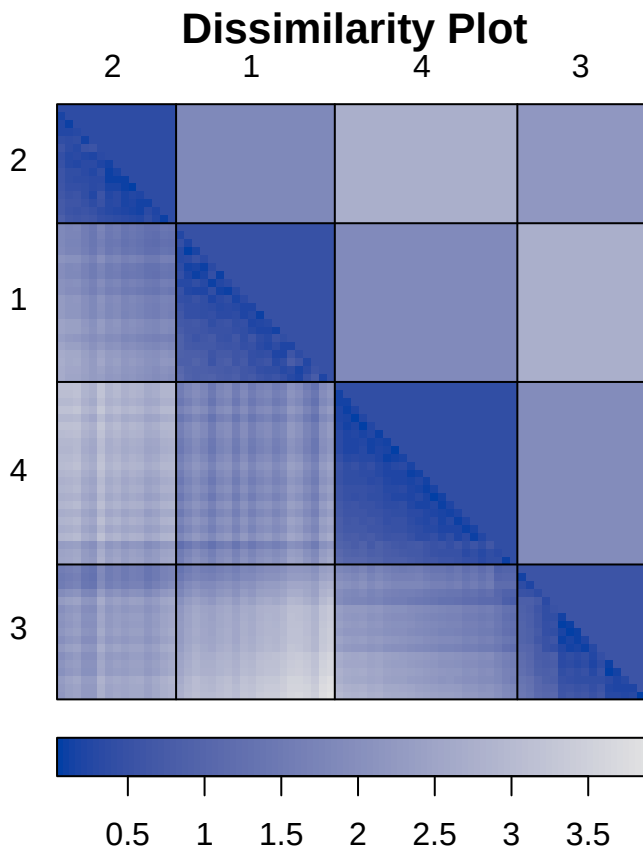
Reordered by cluster



Remember, the rows and columns in the matrix represent the objects (rows of the original data). The reordered distance matrix has the objects ordered by cluster (all points in cluster 1 followed by all points in cluster 2, and so on). The matrix visualization shows a clear block structure indicating that the points within each cluster are close together (low distance), while the other clusters are farther away.

An advanced version of this plot is called a dissimilarity plot. It reorders rows and columns based on cluster labels and adds lines for cluster borders. It also presents average distance values above the diagonal to make the structure between clusters easier to evaluate.

```
dissplot(d, km$cluster)
```



7.5.1.2 Evaluation Metrics

The two most popular quality metrics are the within-cluster sum of squares (WCSS) used as the optimization objective by *k*-means⁹ and the average silhouette width¹⁰. Look at `within.cluster.ss` and `avg.silwidth` below.

A comprehensive set of evaluation metric is calculated by `cluster.stats()` in package `fpc`.

```
# library(fpc)
fpc::cluster.stats(d, as.integer(km$cluster))
## $n
## [1] 75
##
## $cluster.number
## [1] 4
```

⁹https://en.wikipedia.org/wiki/K-means_clustering

¹⁰[https://en.wikipedia.org/wiki/Silhouette_\(clustering\)](https://en.wikipedia.org/wiki/Silhouette_(clustering))

```

##
## $cluster.size
## [1] 20 15 17 23
##
## $min.cluster.size
## [1] 15
##
## $noisen
## [1] 0
##
## $diameter
## [1] 1.1193 0.8359 1.4627 1.1591
##
## $average.distance
## [1] 0.4824 0.3564 0.5806 0.4286
##
## $median.distance
## [1] 0.4492 0.3380 0.5024 0.3934
##
## $separation
## [1] 1.1577 1.1577 0.7676 0.7676
##
## $average.toother
## [1] 2.157 2.308 2.293 2.149
##
## $separation.matrix
##      [,1] [,2] [,3] [,4]
## [1,] 0.000 1.158 1.3397 1.2199
## [2,] 1.158 0.000 1.3084 1.9577
## [3,] 1.340 1.308 0.0000 0.7676
## [4,] 1.220 1.958 0.7676 0.0000
##
## $ave.between.matrix
##      [,1] [,2] [,3] [,4]
## [1,] 0.000 1.874 2.772 1.887
## [2,] 1.874 0.000 2.220 2.750
## [3,] 2.772 2.220 0.000 1.925
## [4,] 1.887 2.750 1.925 0.000
##
## $average.between
## [1] 2.219
##
## $average.within
## [1] 0.463
##

```

```
## $n.between
## [1] 2091
##
## $n.within
## [1] 684
##
## $max.diameter
## [1] 1.463
##
## $min.separation
## [1] 0.7676
##
## $within.cluster.ss
## [1] 10.09
##
## $clus.avg.silwidths
##      1      2      3      4
## 0.7211 0.8074 0.6813 0.7455
##
## $avg.silwidth
## [1] 0.7368
##
## $g2
## NULL
##
## $g3
## NULL
##
## $pearsongamma
## [1] 0.8416
##
## $dunn
## [1] 0.5248
##
## $dunn2
## [1] 3.228
##
## $entropy
## [1] 1.373
##
## $wb.ratio
## [1] 0.2086
##
## $ch
## [1] 323.6
```

```
##
## $cwidegap
## [1] 0.2612 0.2352 0.4150 0.3153
##
## $widestgap
## [1] 0.415
##
## $sindex
## [1] 0.8583
##
## $corrected.rand
## NULL
##
## $vi
## NULL
```

Some notes about the code:

- I do not load the package `fpc` using `library()` since it would mask the `dbscan()` function in package `dbscan`. Instead I use the namespace operator `::`.
- The clustering (second argument below) has to be supplied as a vector of integers (cluster IDs) and cannot be a factor (to make sure, we can use `as.integer()`).

Most metrics are easy to identify by their name. Some important metrics are:

- `cluster.size`: vector of cluster sizes
- `within.cluster.ss`: within clusters sum of squares (k-means objective function).
- `avg.silwidth`: average silhouette width
- `pearsongamma`: correlation between the distances and a 0-1-vector cluster incidence matrix.

Some numbers are `NULL`. These are measures only available for supervised evaluation. Read the man page for `cluster.stats()` for an explanation of all the available indices.

We can compare different clusterings.

```
sapply(
  list(
    km_4 = km$cluster,
    hc_compl_4 = cutree(hc, k = 4),
```



```

    hc_compl_5 = cutree(hc, k = 5),
    hc_single_5 = cutree(hc_single, k = 5)
),
FUN = function(x)
  fpc::cluster.stats(d, x)[c("within.cluster.ss",
                             "avg.silwidth",
                             "pearsongamma",
                             "dunn"), ]
##           km_4   hc_compl_4 hc_compl_5 hc_single_5
## within.cluster.ss 10.09   10.09      8.314      7.791
## avg.silwidth      0.7368 0.7368      0.6642      0.6886
## pearsongamma      0.8416 0.8416      0.8042      0.816
## dunn              0.5248 0.5248      0.1988      0.358

```

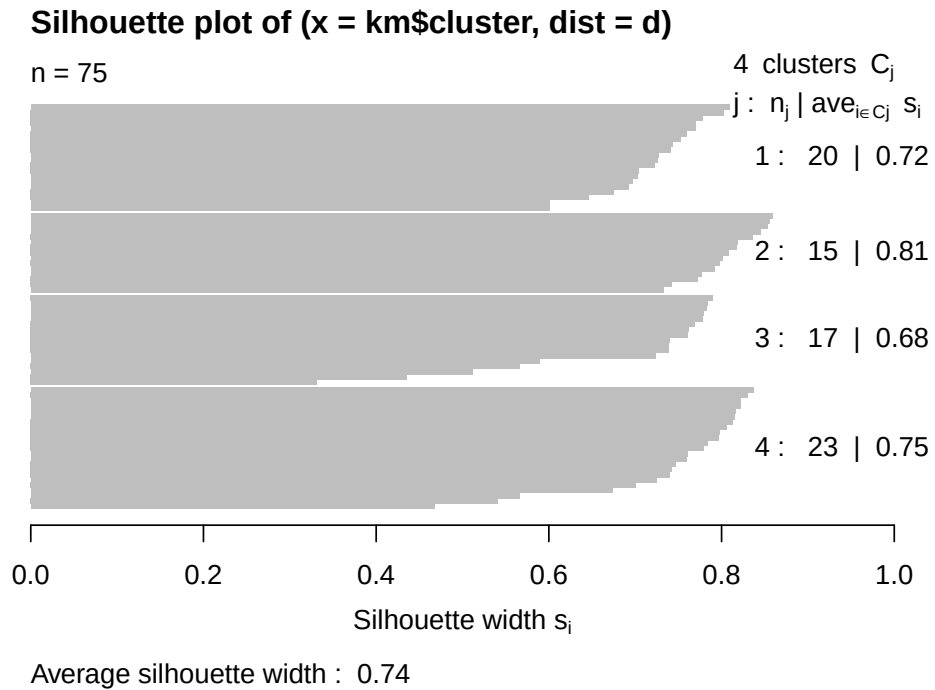
With 4 clusters, k-means and hierarchical clustering produce exactly the same clustering. While the two different hierarchical methods with 5 clusters produce a smaller WCSS, they are actually worse given all three other measures.

Next, we look at the silhouette using a silhouette plot.

```

library(cluster)
##
## Attaching package: 'cluster'
## The following object is masked _by_ '.GlobalEnv':
##
##      ruspini
plot(silhouette(km$cluster, d))

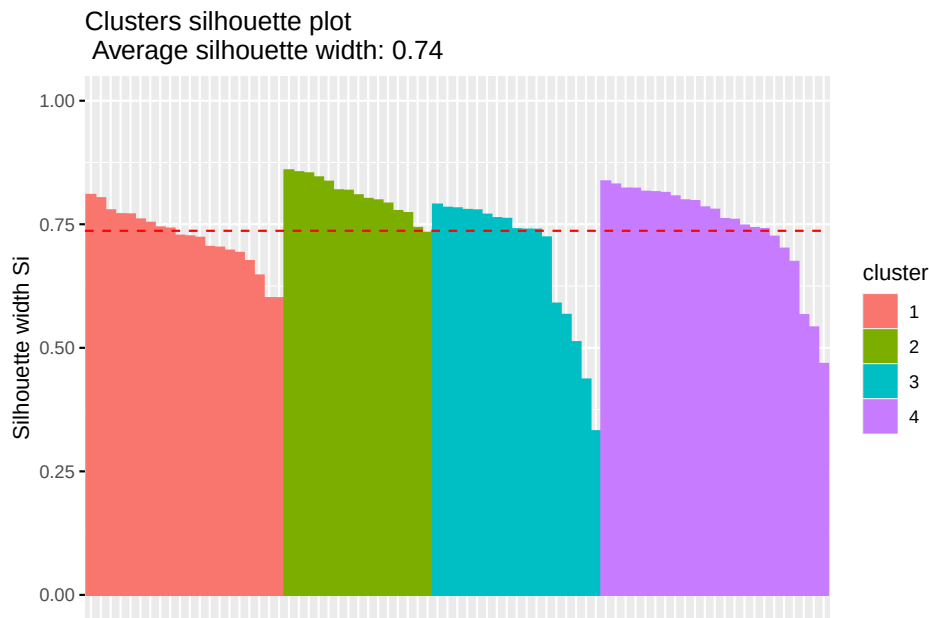
```



Note: The silhouette plot does not show correctly in R Studio if you have too many objects (bars are missing). I will work when you open a new plotting device with `windows()`, `x11()` or `quartz()`.

ggplot visualization using `factoextra`

```
fviz_silhouette(silhouette(km$cluster, d))
## cluster size ave.sil.width
## 1      1  20      0.72
## 2      2  15      0.81
## 3      3  17      0.68
## 4      4  23      0.75
```

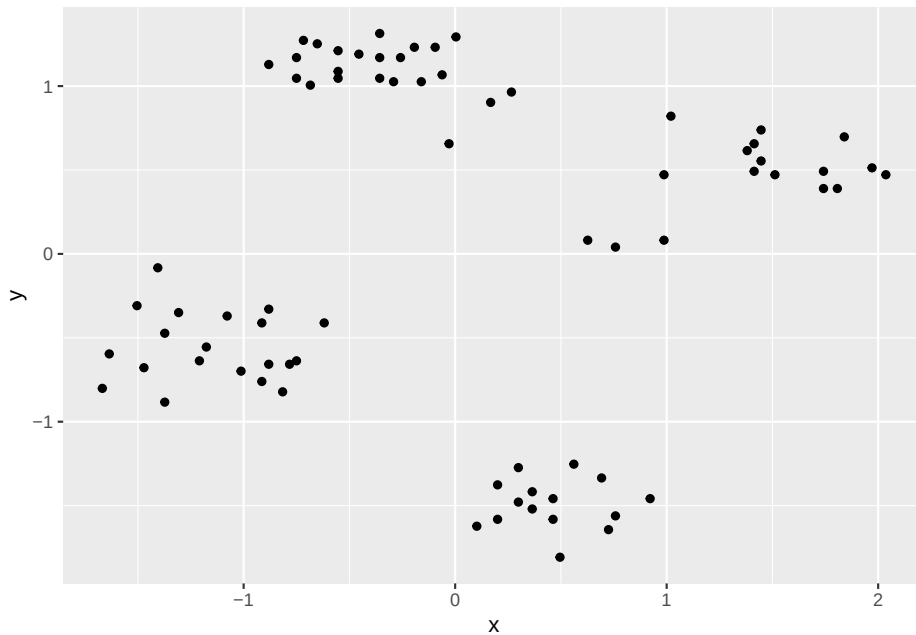


7.5.2 Determining the Correct Number of Clusters

The user needs to specify the number of clusters for most clustering algorithms. Determining the number of clusters in a data set¹¹ is therefore an important task before we can cluster data. We will apply different methods to the scaled Ruspini data set.

```
ggplot(ruspini_scaled, aes(x, y)) + geom_point()
```

¹¹https://en.wikipedia.org/wiki/Determining_the_number_of_clusters_in_a_data_set



```
## We will use different methods and try 1-10 clusters.
set.seed(1234)
ks <- 2:10
```

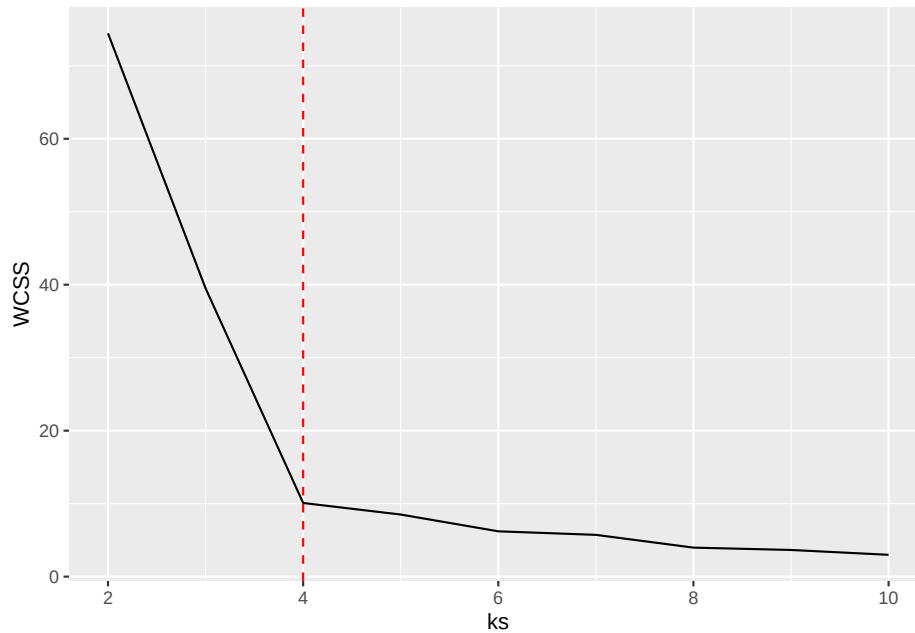
7.5.2.1 Elbow Method: Within-Cluster Sum of Squares

A method often used for k-means is to calculate the within-cluster sum of squares for different numbers of clusters and look for the knee or elbow¹² in the plot. We use `nstart = 5` to restart k-means 5 times and return the best solution.

```
WCSS <- sapply(ks, FUN = function(k) {
  kmeans(ruspini_scaled, centers = k, nstart = 5)$tot.withinss
})

ggplot(tibble(ks, WCSS), aes(ks, WCSS)) +
  geom_line() +
  geom_vline(xintercept = 4, color = "red", linetype = 2)
```

¹²[https://en.wikipedia.org/wiki/Elbow_method_\(clustering\)](https://en.wikipedia.org/wiki/Elbow_method_(clustering))

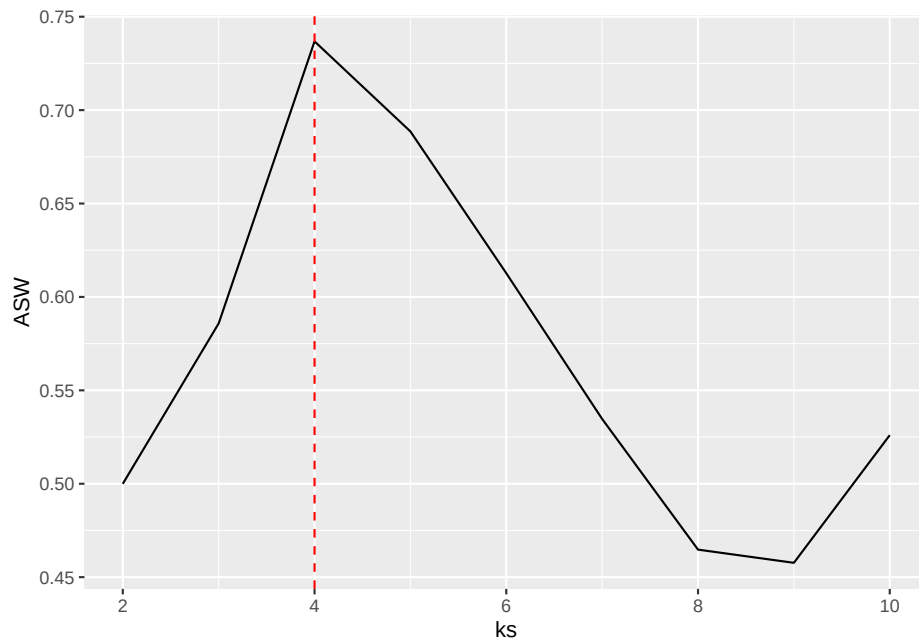


7.5.2.2 Average Silhouette Width

Another popular method (often preferred for clustering methods starting with a distance matrix) is to plot the average silhouette width for different numbers of clusters and look for the maximum in the plot.

```
ASW <- sapply(ks, FUN=function(k) {
  fpc::cluster.stats(d,
    kmeans(ruspini_scaled,
      centers = k,
      nstart = 5)$cluster)$avg.silwidth
})

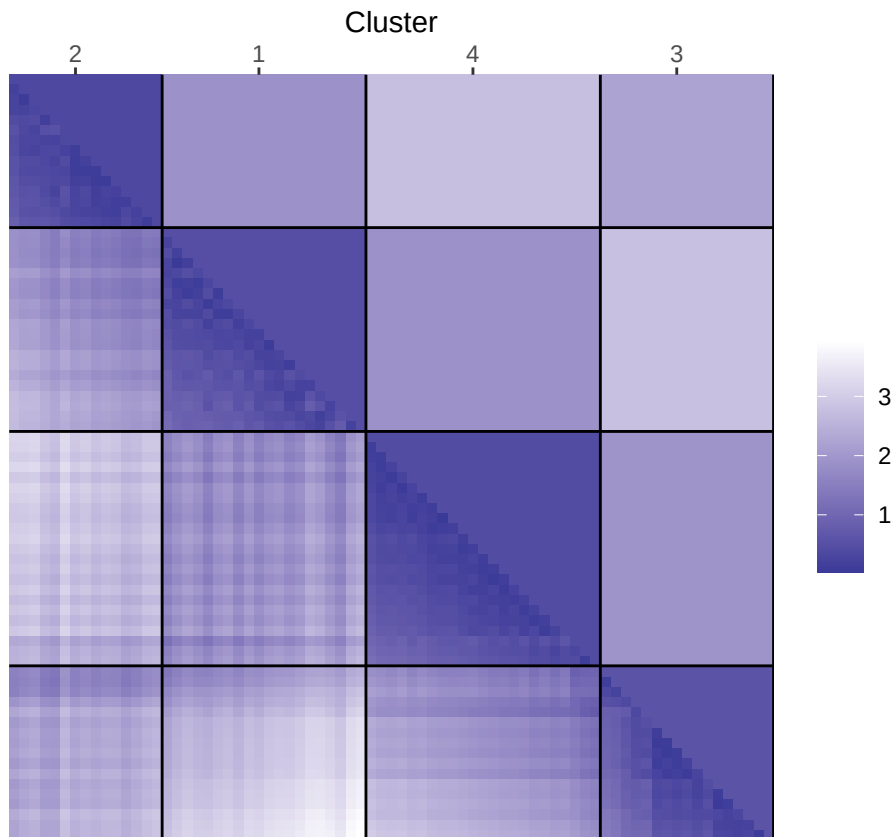
best_k <- ks[which.max(ASW)]
best_k
## [1] 4
ggplot(tibble(ks, ASW), aes(ks, ASW)) +
  geom_line() +
  geom_vline(xintercept = best_k, color = "red", linetype = 2)
```



7.5.2.3 Similarity Matrix Visualization

We can visualize a similarity matrix reordered by cluster labels to visually determine if the number of clusters is a good fit for the data. Package **seriation** provides a dissimilarity plot to make this easy.

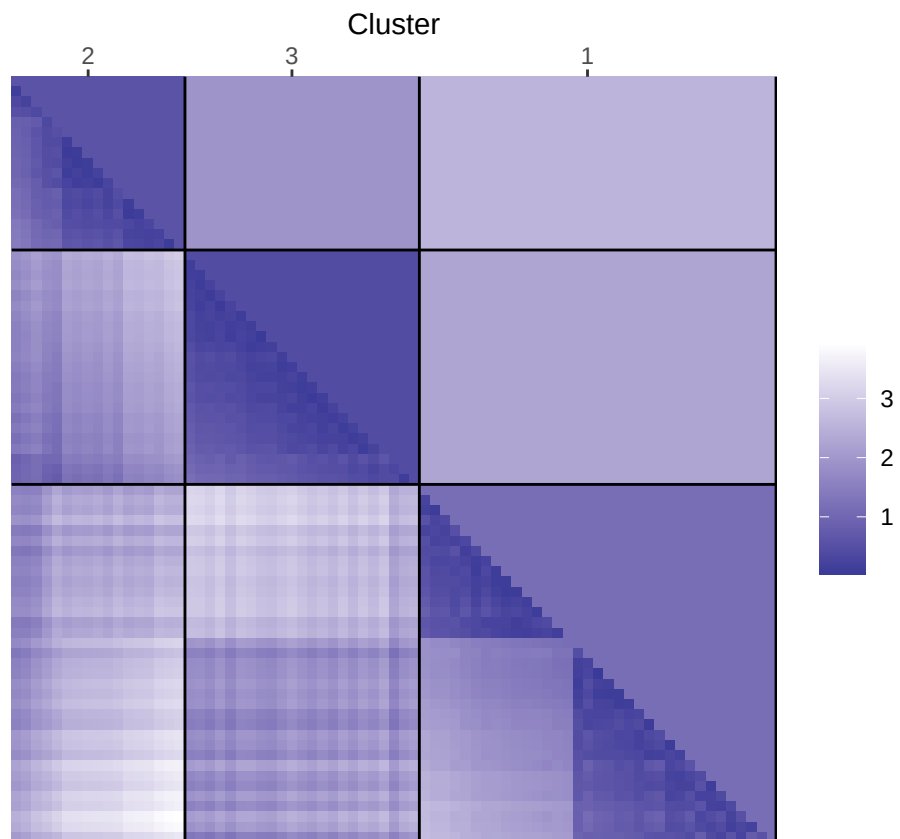
```
ggdissplot(d, labels = km$cluster,  
           options = list(main = "k-means with k=4"))
```



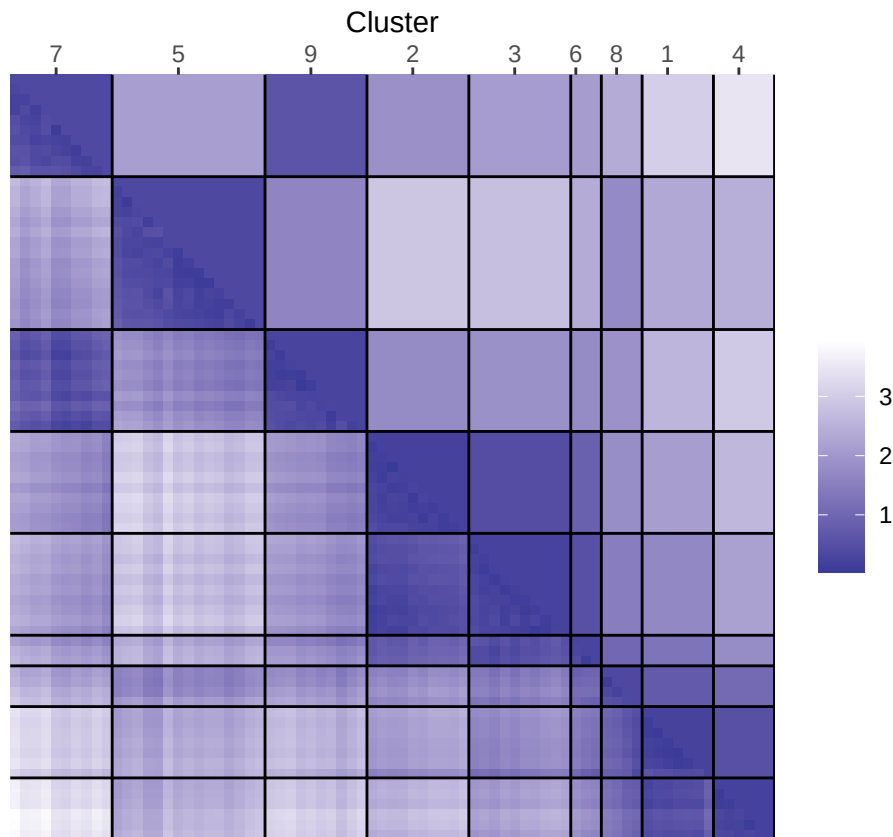
The plot reorders rows and columns based on cluster labels and adds lines for cluster borders. It also presents average distance values above the diagonal to make the structure between clusters easier to evaluate. Above, we see a clear structure with the four clusters.

Next, we look at two examples where the number of clusters was misspecified.

```
ggdissplot(d,
  labels = kmeans(ruspini_scaled, centers = 3, nstart = 5)$cluster)
```



```
ggdisplot(d,  
  labels = kmeans(ruspini_scaled, centers = 9, nstart = 5)$cluster)
```

In the example with 3 clusters, we see that the plot shows two separated triangles for the largest cluster indicating that it contains two separate structures which should be their own clusters. For the example with 9 clusters, the plot tries to rearrange the clusters into four larger blocks indicating that the number of clusters should be 4.

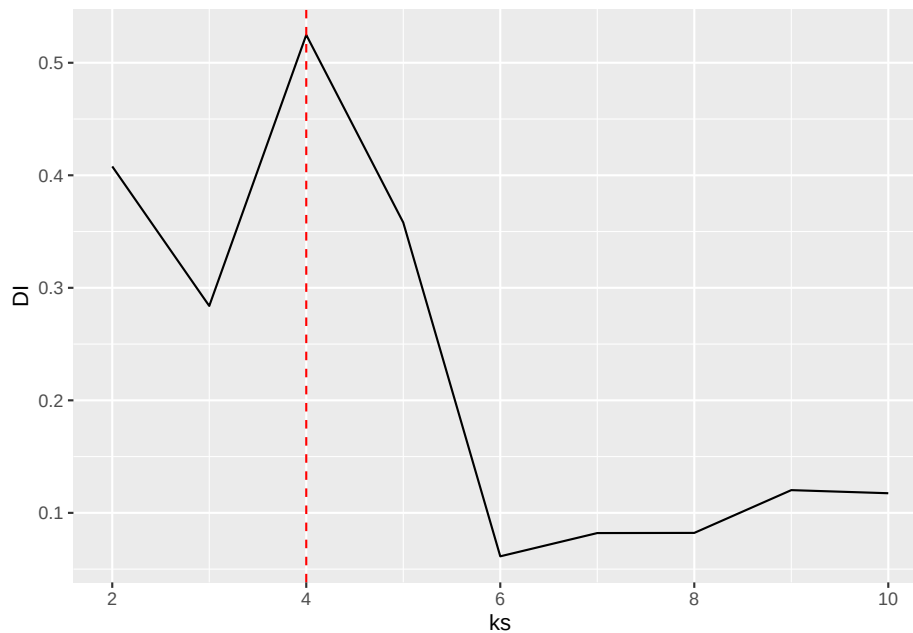
7.5.2.4 Dunn Index

The Dunn index¹³ is another internal measure given by the smallest separation between clusters scaled by the largest cluster diameter.

```
DI <- sapply(ks, FUN = function(k) {
  fpc::cluster.stats(d,
    kmeans(ruspini_scaled, centers = k,
      nstart = 5)$cluster)$dunn
})
```

¹³https://en.wikipedia.org/wiki/Dunn_index

```
best_k <- ks[which.max(DI)]
ggplot(tibble(ks, DI), aes(ks, DI)) +
  geom_line() +
  geom_vline(xintercept = best_k, color = "red", linetype = 2)
```



7.5.2.5 Gap Statistic

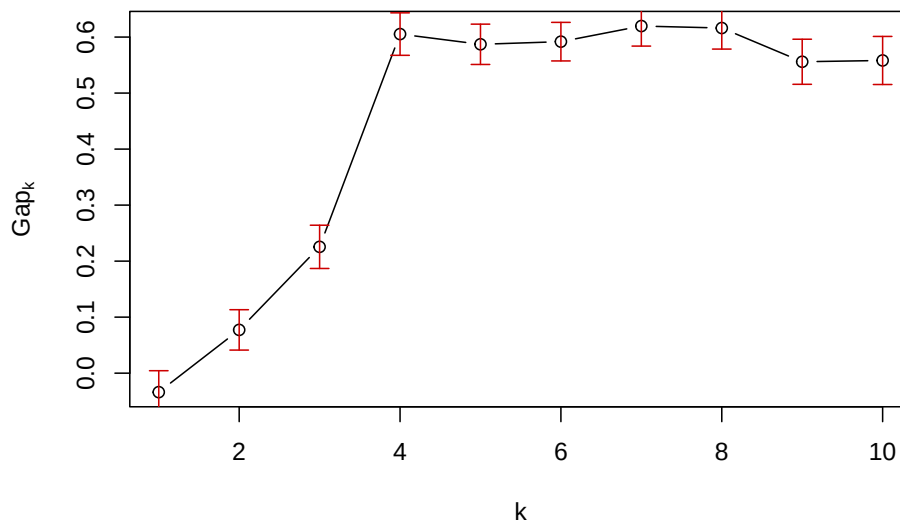
The Gap statistic¹⁴ compares the change in within-cluster dispersion with that expected from a null model (see `clusGap()`). The default method is to choose the smallest k such that its value $\text{Gap}(k)$ is not more than 1 standard error away from the first local maximum.

```
library(cluster)
k <- clusGap(ruspini_scaled,
             FUN = kmeans,
             nstart = 10,
             K.max = 10)
k
## Clustering Gap statistic ["clusGap"] from call:
## clusGap(x = ruspini_scaled, FUNcluster = kmeans, K.max = 10, nstart = 10)
## B=100 simulated reference sets, k = 1..10; spaceH0="scaledPCA"
```

¹⁴https://en.wikipedia.org/wiki/Determining_the_number_of_clusters_in_a_data_set#The_gap_statistics

```
## --> Number of clusters (method 'firstSEmax', SE.factor=1): 4
##      logW E.logW      gap SE.sim
## [1,] 3.498 3.464 -0.03387 0.03824
## [2,] 3.074 3.151 0.07724 0.03605
## [3,] 2.678 2.903 0.22549 0.03861
## [4,] 2.106 2.712 0.60533 0.03793
## [5,] 1.987 2.574 0.58708 0.03599
## [6,] 1.864 2.456 0.59181 0.03442
## [7,] 1.732 2.352 0.61966 0.03577
## [8,] 1.640 2.257 0.61613 0.03758
## [9,] 1.612 2.168 0.55592 0.04024
## [10,] 1.531 2.090 0.55823 0.04291
plot(k)
```

clusGap(x = ruspini_scaled, FUNcluster = kmeans, K.max = 10, nstart = 10)



There have been many other methods and indices proposed to determine the number of clusters. See, e.g., package NbClust¹⁵.

7.5.3 Clustering Tendency

Most clustering algorithms will always produce a clustering, even if the data does not contain a cluster structure. It is typically good to check cluster tendency before attempting to cluster the data.

We use again the smiley data.

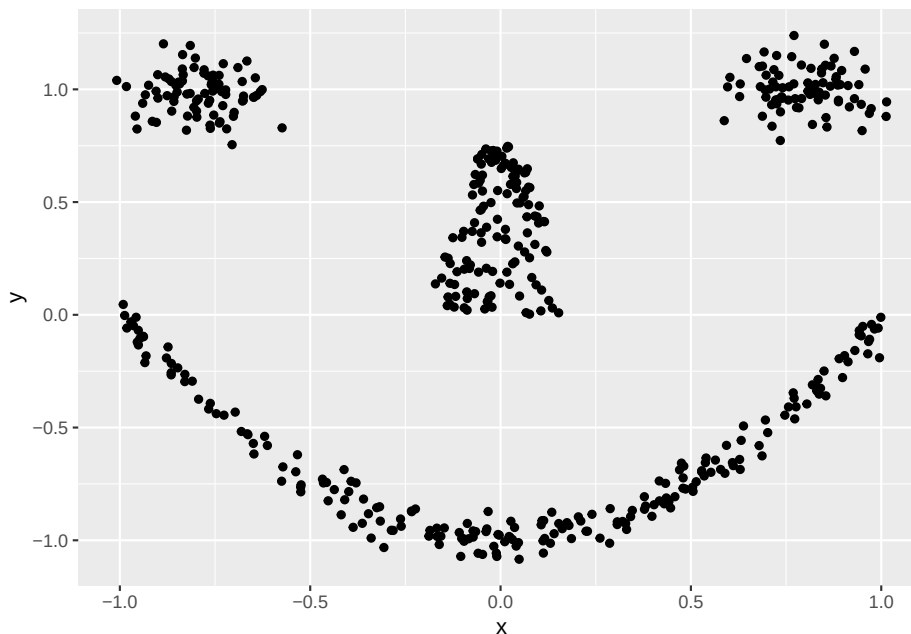
¹⁵<https://cran.r-project.org/package=NbClust>

```
library(mlbench)
shapes <- mlbench.smiley(n = 500, sd1 = 0.1, sd2 = 0.05)$x
colnames(shapes) <- c("x", "y")
shapes <- as_tibble(shapes)
```

7.5.3.1 Scatter plots

The first step is visual inspection using scatter plots.

```
ggplot(shapes, aes(x = x, y = y)) + geom_point()
```



Cluster tendency is typically indicated by several separated point clouds. Often an appropriate number of clusters can also be visually obtained by counting the number of point clouds. We see four clusters, but the mouth is not convex/spherical and thus will pose a problems to algorithms like k-means.

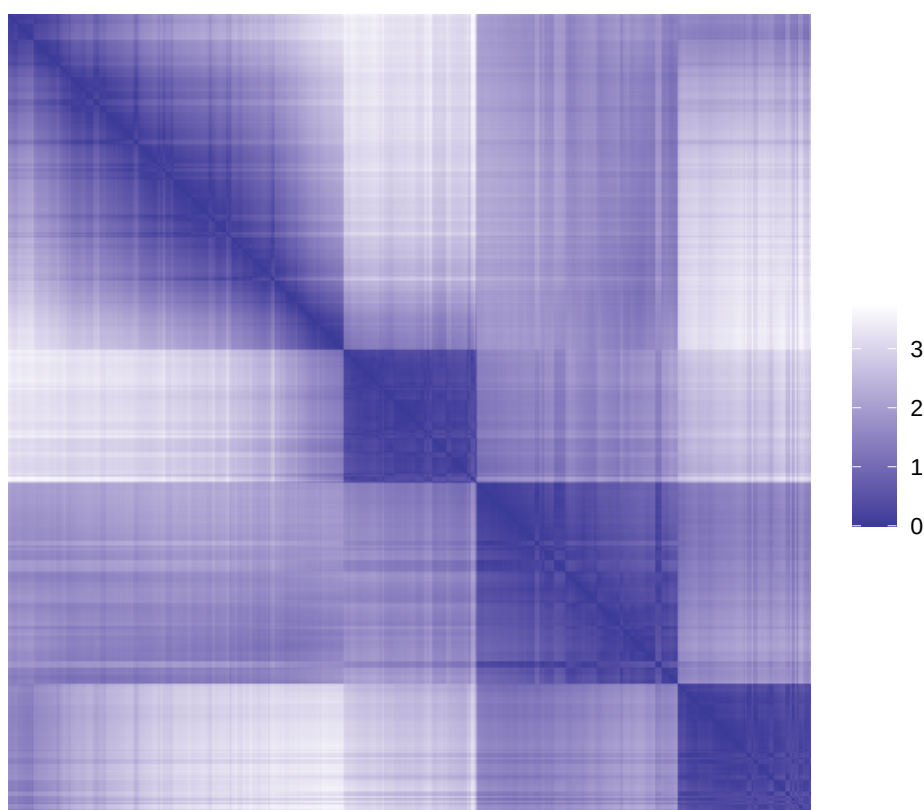
If the data has more than two features then you can use a pairs plot (scatterplot matrix) or look at a scatterplot of the first two principal components using PCA.

7.5.3.2 Visual Analysis for Cluster Tendency Assessment (VAT)

VAT reorders the objects to show potential clustering tendency as a block structure (dark blocks along the main diagonal). We scale the data before using Euclidean distance.

```
library(seriation)

d_shapes <- dist(scale(shapes))
ggVAT(d_shapes)
```



iVAT uses the largest distances for all possible paths between two objects instead of the direct distances to make the block structure better visible.

```
ggiVAT(d_shapes)
```



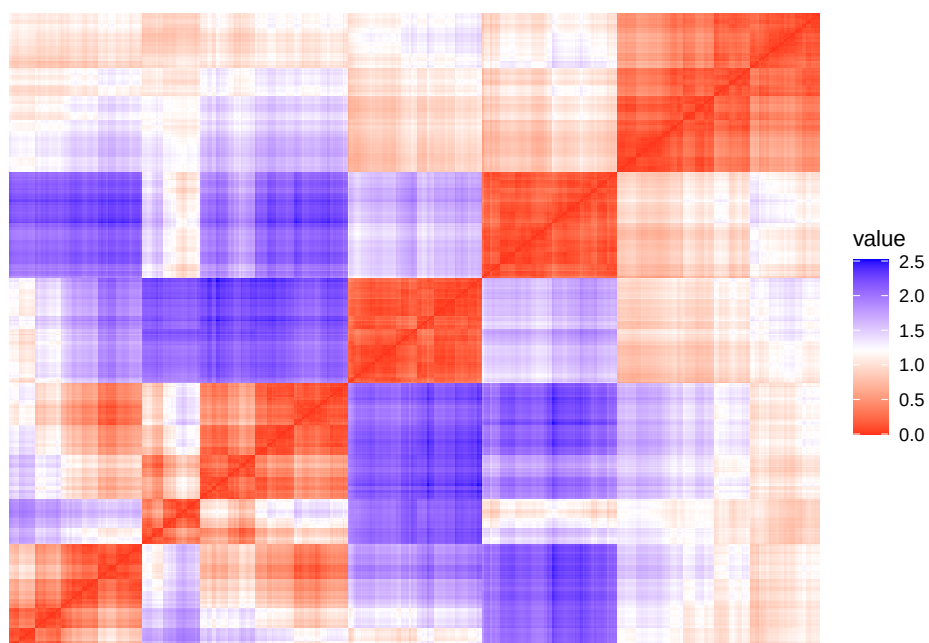
7.5.3.3 Hopkins statistic

`factoextra` can also create a VAT plot and calculate the Hopkins statistic¹⁶ to assess clustering tendency. For the Hopkins statistic, a sample of size n is drawn from the data and then compares the nearest neighbor distribution with a simulated dataset drawn from a random uniform distribution (see detailed explanation¹⁷). A values $>.5$ indicates usually a clustering tendency.

```
get_clust_tendency(shapes, n = 10)
## $hopkins_stat
## [1] 0.9498
##
## $plot
```

¹⁶https://en.wikipedia.org/wiki/Hopkins_statistic

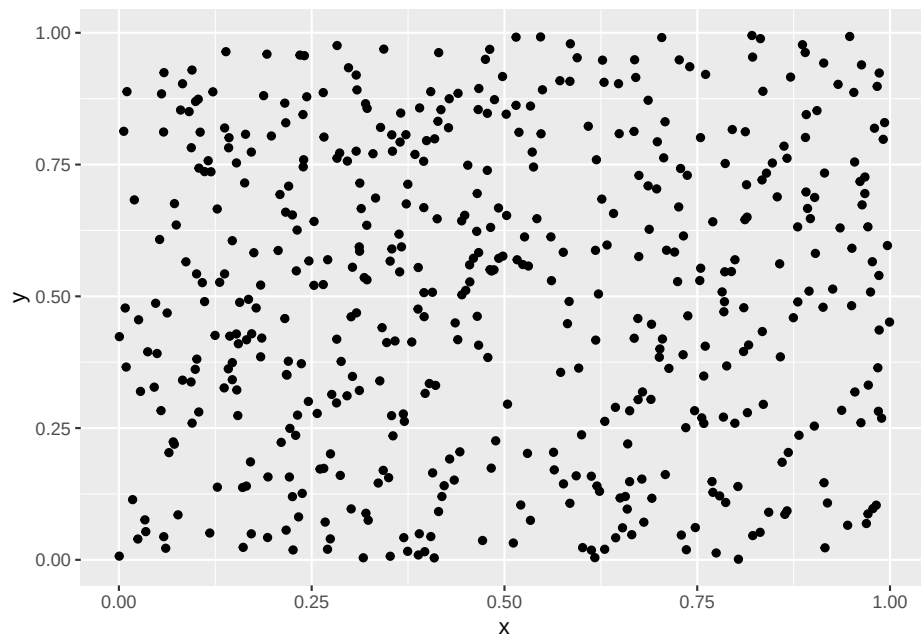
¹⁷<https://www.datanovia.com/en/lessons/assessing-clustering-tendency/#statistical-methods>



Both plots show a strong cluster structure with 4 clusters.

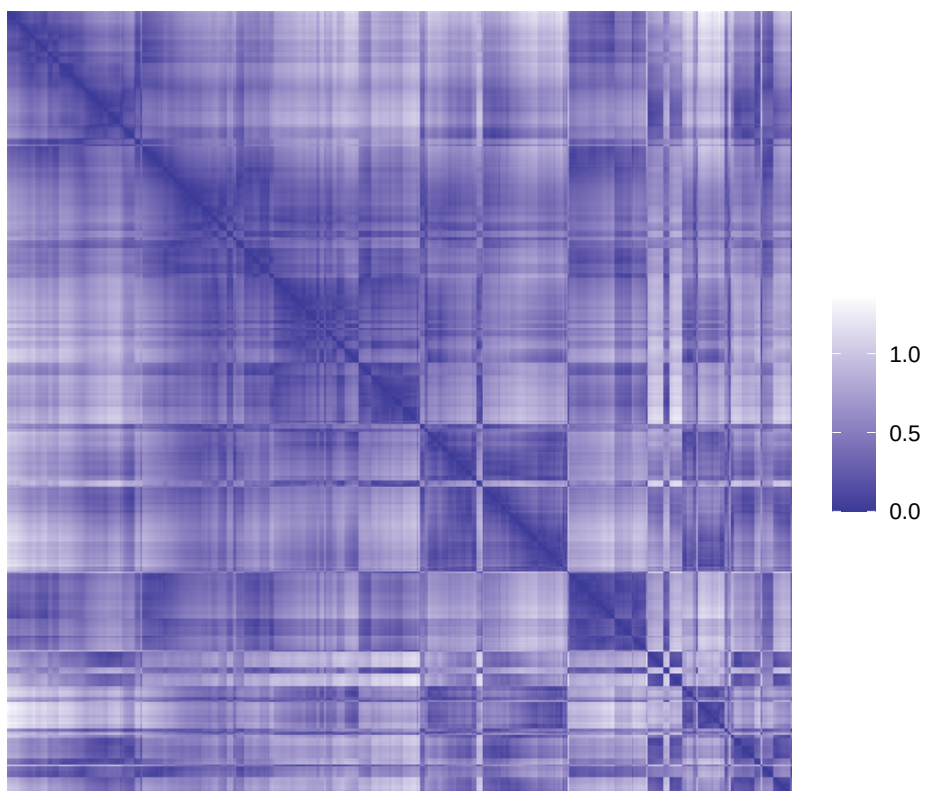
7.5.3.4 Data Without Clustering Tendency

```
data_random <- tibble(x = runif(500), y = runif(500))  
ggplot(data_random, aes(x, y)) + geom_point()
```

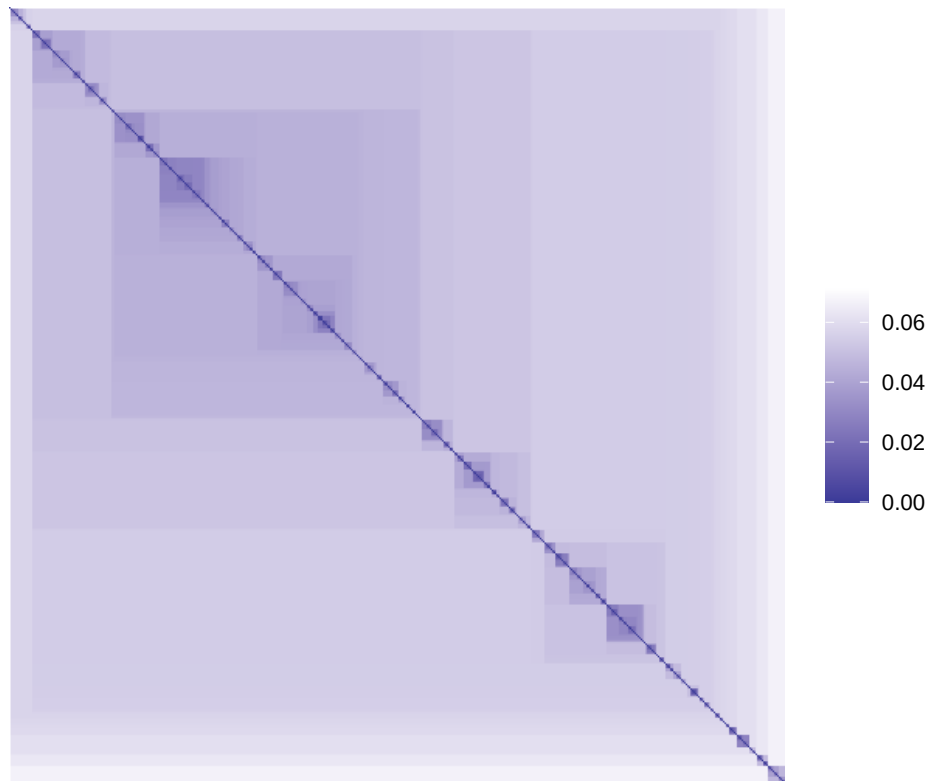


No point clouds are visible, just noise.

```
d_random <- dist(data_random)
ggVAT(d_random)
```

```
ggiVAT(d_random)
```



```
get_clust_tendency(data_random, n = 10, graph = FALSE)
## $hopkins_stat
## [1] 0.4642
##
## $plot
## NULL
```

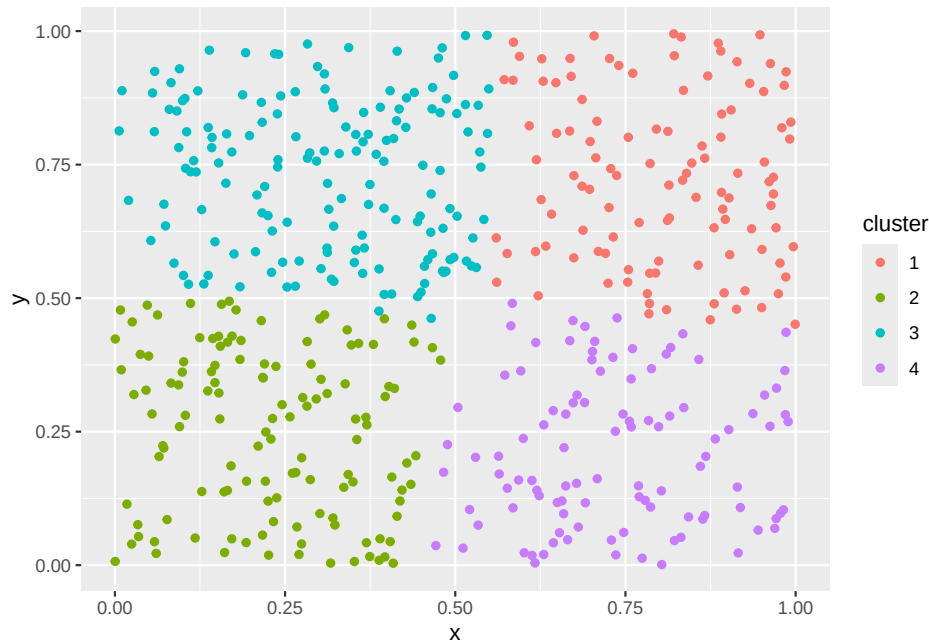
There is very little clustering structure visible indicating low clustering tendency and clustering should not be performed on this data. However, k-means can be used to partition the data into k regions of roughly equivalent size. This can be used as a data-driven discretization of the space.

7.5.3.5 k-means on Data Without Clustering Tendency

What happens if we perform k-means on data that has no inherent clustering structure?

```
km <- kmeans(data_random, centers = 4)
```

```
random_clustered <- data_random |>
  add_column(cluster = factor(km$cluster))
ggplot(random_clustered, aes(x = x, y = y, color = cluster)) +
  geom_point()
```



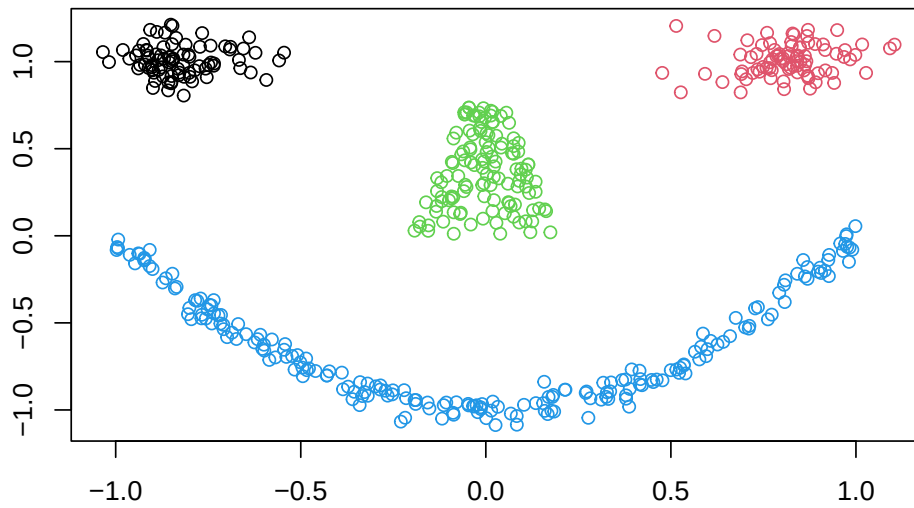
k-means discretizes the space into similarly sized regions.

7.5.4 Supervised Cluster Evaluation

Also called external cluster validation since it uses ground truth information. That is, the user has an idea how the data should be grouped. This could be a known class label not provided to the clustering algorithm.

We use an artificial data set with known groups.

```
library(mlbench)
set.seed(1234)
shapes <- mlbench.smiley(n = 500, sd1 = 0.1, sd2 = 0.05)
plot(shapes)
```

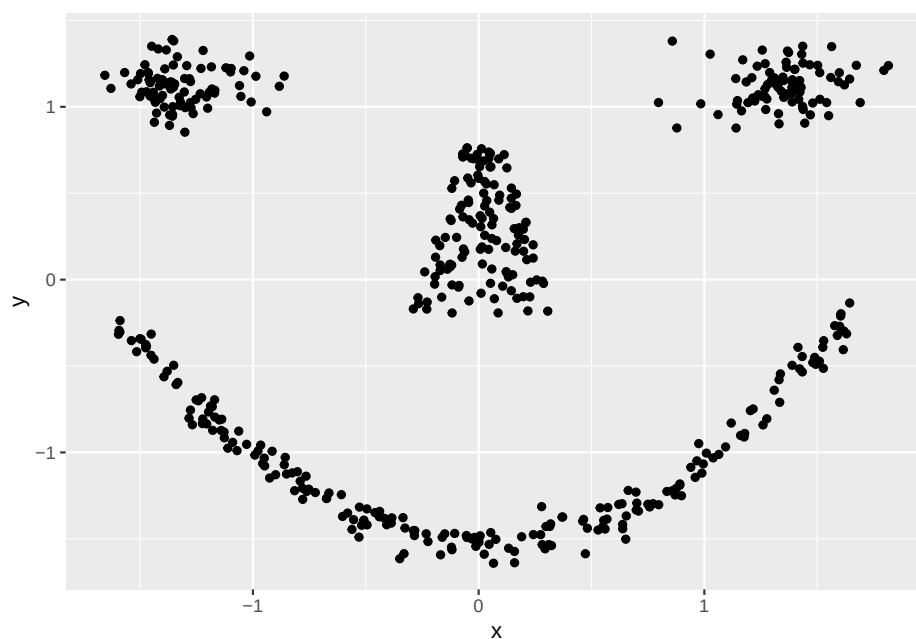


First, we prepare the data and hide the known class label which we will use for evaluation after clustering.

```
truth <- as.integer(shapes$class)
shapes <- shapes$x
colnames(shapes) <- c("x", "y")

shapes <- shapes |> scale() |> as_tibble()

ggplot(shapes, aes(x, y)) + geom_point()
```

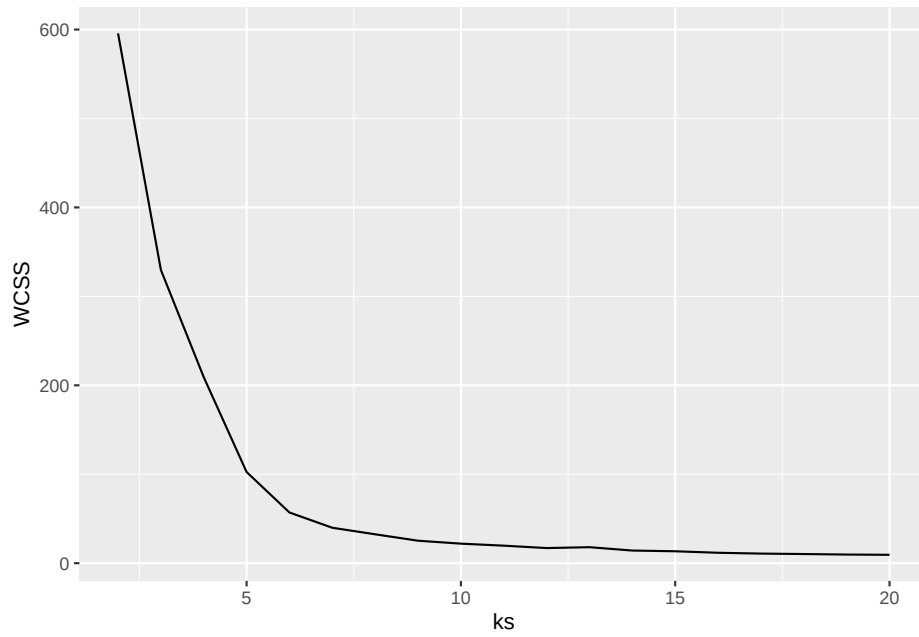


Find optimal number of Clusters for k-means

```
ks <- 2:20
```

Use within sum of squares (look for the knee)

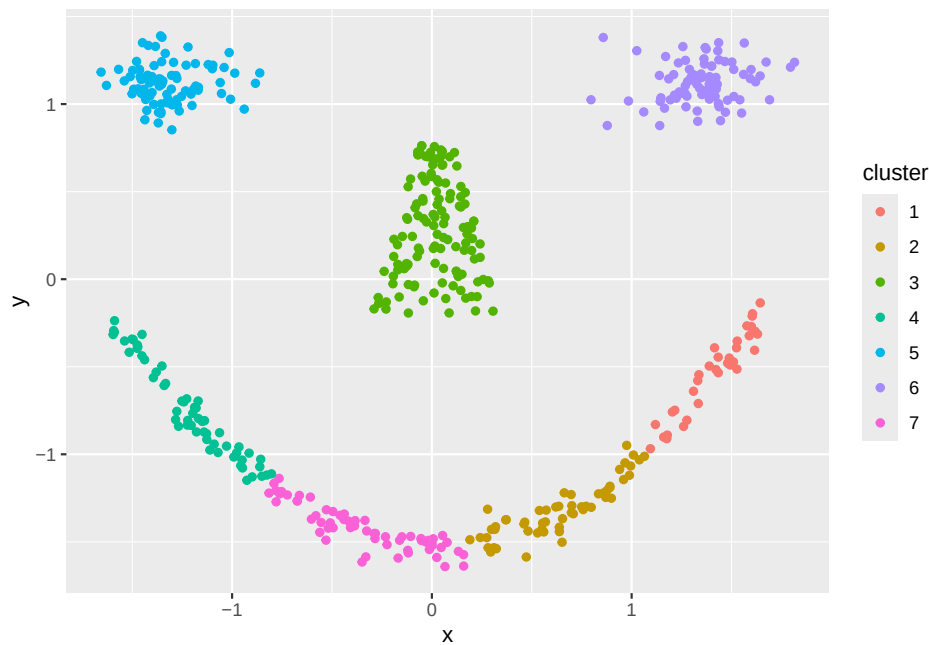
```
WCSS <- sapply(ks, FUN = function(k) {  
  kmeans(shapes, centers = k, nstart = 10)$tot.withinss  
})  
  
ggplot(tibble(ks, WCSS), aes(ks, WCSS)) + geom_line()
```



Looks like it could be 7 clusters?

```
km <- kmeans(shapes, centers = 7, nstart = 10)

ggplot(shapes |> add_column(cluster = factor(km$cluster)),
       aes(x, y, color = cluster)) +
  geom_point()
```



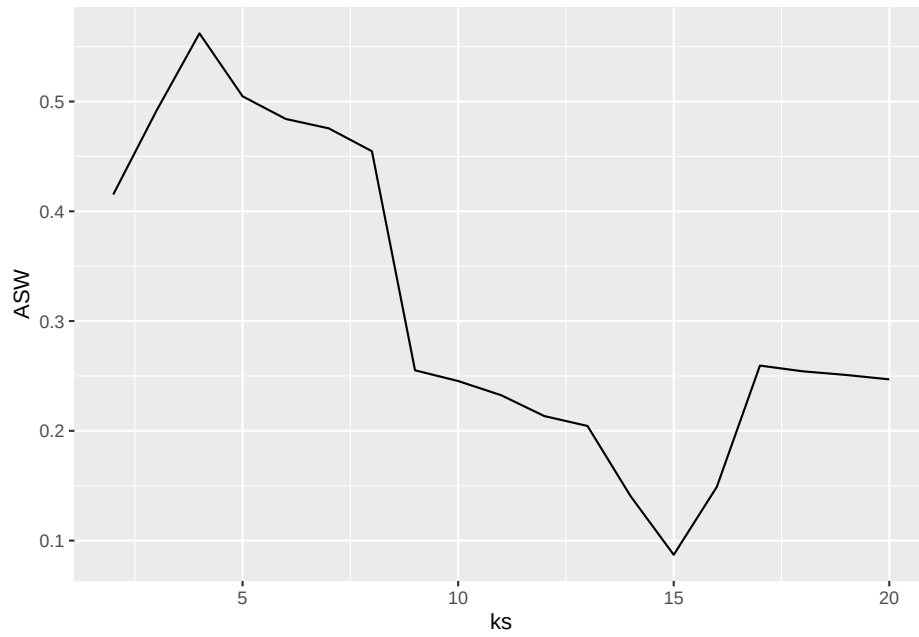
The mouth is an issue for k-means. We could use hierarchical clustering with single-linkage because of the mouth is non-convex and chaining may help.

```
d <- dist(shapes)
hc <- hclust(d, method = "single")
```

Find optimal number of clusters

```
ASW <- sapply(ks, FUN = function(k) {
  fpc::cluster.stats(d, cutree(hc, k))$avg.silwidth
})

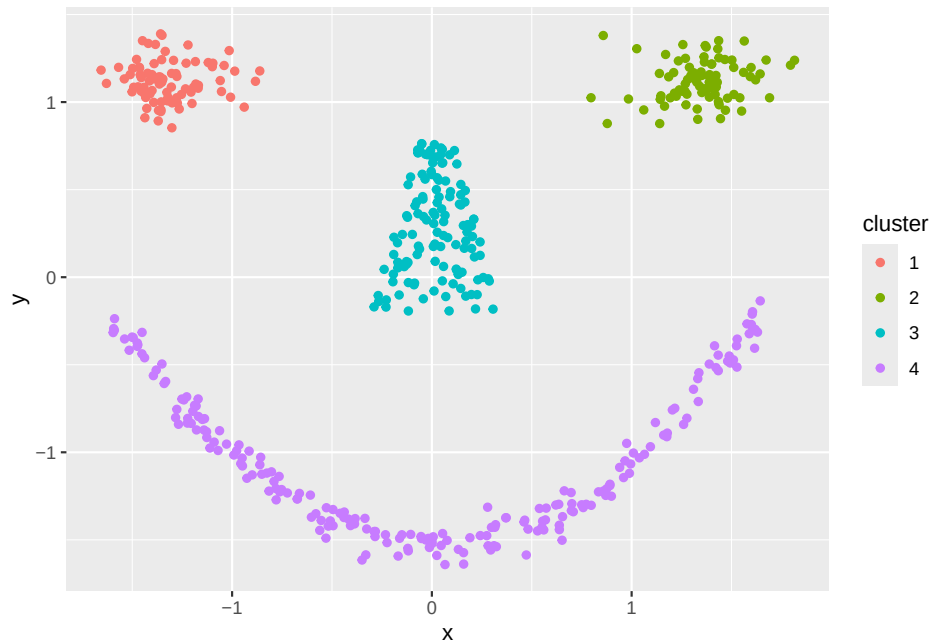
ggplot(tibble(ks, ASW), aes(ks, ASW)) + geom_line()
```



The maximum is clearly at 4 clusters.

```
hc_4 <- cutree(hc, 4)

ggplot(shapes |> add_column(cluster = factor(hc_4)),
       aes(x, y, color = cluster)) +
  geom_point()
```

Compare with ground truth with the adjusted Rand index (ARI, also corrected Rand index)¹⁸, the variation of information (VI) index¹⁹, mutual information (MI)²⁰, entropy²¹ and purity²².

`cluster_stats()` computes ARI and VI as comparative measures. I define functions for the purity and the entropy of a clustering given the ground truth here:

```
purity <- function(cluster, truth, show_table = FALSE) {
  if (length(cluster) != length(truth))
    stop("Cluster vector and ground truth vectors are not of the same length!")

  # tabulate
  tbl <- table(cluster, truth)
  if(show_table)
    print(tbl)

  # find majority class
  majority <- apply(tbl, 1, max)
  sum(majority) / length(cluster)
}
```

¹⁸https://en.wikipedia.org/wiki/Rand_index#Adjusted_Rand_index

¹⁹https://en.wikipedia.org/wiki/Variation_of_information

²⁰https://en.wikipedia.org/wiki/Mutual_information

²¹[https://en.wikipedia.org/wiki/Entropy_\(information_theory\)](https://en.wikipedia.org/wiki/Entropy_(information_theory))

²²https://en.wikipedia.org/wiki/Cluster_analysis#External_evaluation

```

entropy <- function(cluster, truth, show_table = FALSE) {
  if (length(cluster) != length(truth))
    stop("Cluster vector and ground truth vectors are not of the same length!")

  # calculate membership probability of cluster to class
  tbl <- table(cluster, truth)
  p <- sweep(tbl, 2, colSums(tbl), "/")

  if(show_table)
    print(p)

  # calculate cluster entropy
  e <- -p * log(p, 2)
  e <- rowSums(e, na.rm = TRUE)

  # weighted sum over clusters
  w <- table(cluster) / length(cluster)
  sum(w * e)
}

```

We calculate the measures and add for comparison two random “clusterings” with 4 and 6 clusters.

```

random_4 <- sample(1:4, nrow(shapes), replace = TRUE)
random_6 <- sample(1:6, nrow(shapes), replace = TRUE)

r <- rbind(
  truth = c(
    unlist(fpc::cluster.stats(d, truth,
                             truth, compareonly = TRUE)),
    purity = purity(truth, truth),
    entropy = entropy(truth, truth)
  ),

  kmeans_7 = c(
    unlist(fpc::cluster.stats(d, km$cluster,
                             truth, compareonly = TRUE)),
    purity = purity(km$cluster, truth),
    entropy = entropy(km$cluster, truth)
  ),

  hc_4 = c(
    unlist(fpc::cluster.stats(d, hc_4,
                             truth, compareonly = TRUE)),
    purity = purity(hc_4, truth),
    entropy = entropy(hc_4, truth)
  )
)

```

```

),
random_4 = c(
  unlist(fpc::cluster.stats(d, random_4,
                           truth, compareonly = TRUE)),
  purity = purity(random_4, truth),
  entropy = entropy(random_4, truth)
),
random_6 = c(
  unlist(fpc::cluster.stats(d, random_6,
                           truth, compareonly = TRUE)),
  purity = purity(random_6, truth),
  entropy = entropy(random_6, truth)
)
)
r
##           corrected.rand      vi purity entropy
## truth           1.000000 0.0000   1.000   0.0000
## kmeans_7         0.638229 0.5709   1.000   0.2088
## hc_4             1.000000 0.0000   1.000   0.0000
## random_4        -0.003235 2.6832   0.418   1.9895
## random_6        -0.002125 3.0763   0.418   1.7129

```

Notes:

- Comparing the ground truth to itself produces perfect scores.
- Hierarchical clustering found the perfect clustering.
- Entropy and purity are heavily impacted by the number of clusters (more clusters improve the metric) and the fact that the mouth has 41.8% of all the data points becoming automatically the majority class for purity.
- The adjusted rand index shows clearly that the random clusterings have no relationship with the ground truth (very close to 0). This is a very helpful property and explains why the ARI is the most popular measure.

Read the manual page for `fpc::cluster.stats()` for an explanation of all the available indices.

7.6 More Clustering Algorithms*

Note: Some of these methods are covered in Chapter 8 of the textbook.

7.6.1 Partitioning Around Medoids (PAM)

PAM²³ tries to solve the k -medoids problem. The problem is similar to k -means, but uses medoids instead of centroids to represent clusters. Like hierarchical clustering, it typically works with a precomputed distance matrix. An advantage is that you can use any distance metric not just Euclidean distances. **Note:** The medoid is the most central data point in the middle of the cluster. PAM is a lot more computationally expensive compared to k -means.

```
library(cluster)

d <- dist(ruspini_scaled)
str(d)
## 'dist' num [1:2775] 2.289 1.349 1.918 0.233 1.794 ...
## - attr(*, "Size")= int 75
## - attr(*, "Diag")= logi FALSE
## - attr(*, "Upper")= logi FALSE
## - attr(*, "method")= chr "Euclidean"
## - attr(*, "call")= language dist(x = ruspini_scaled)
p <- pam(d, k = 4)
p
## Medoids:
##      ID
## [1,] 24 24
## [2,] 42 42
## [3,] 30 30
## [4,] 10 10
## Clustering vector:
## [1] 1 2 1 2 1 2 3 1 1 4 4 3 3 1 2 3 1 3 3 1 4 2 2 1 2 4 2 2
## [29] 3 3 4 3 3 4 4 2 3 4 3 3 3 2 4 4 1 3 1 1 1 3 3 3 4 4 4 3
## [57] 2 2 1 4 4 1 2 2 3 3 2 2 1 3 2 3 1 2 2
## Objective function:
## build swap
## 0.4423 0.3187
##
## Available components:
## [1] "medoids"      "id.med"      "clustering" "objective"
## [5] "isolation"    "clusinfo"    "silinfo"    "diss"
## [9] "call"
```

Extract the clustering and medoids for visualization.

²³<https://en.wikipedia.org/wiki/K-medoids>

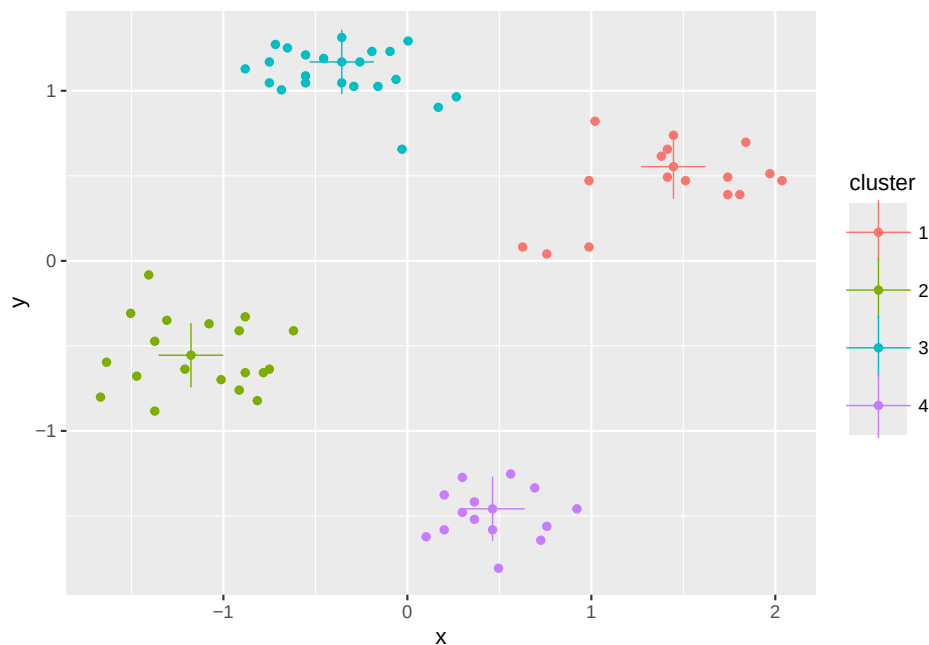
```

ruspini_clustered <- ruspini_scaled |>
  add_column(cluster = factor(p$cluster))

medoids <- as_tibble(ruspini_scaled[p$medoids, ],
  rownames = "cluster")

medoids
## # A tibble: 4 x 3
##   cluster      x      y
##   <chr>    <dbl> <dbl>
## 1 1        1.45  0.554
## 2 2       -1.18 -0.555
## 3 3       -0.357 1.17
## 4 4        0.463 -1.46
ggplot(ruspini_clustered, aes(x = x, y = y, color = cluster)) +
  geom_point() +
  geom_point(data = medoids, aes(x = x, y = y, color = cluster),
    shape = 3, size = 10)

```

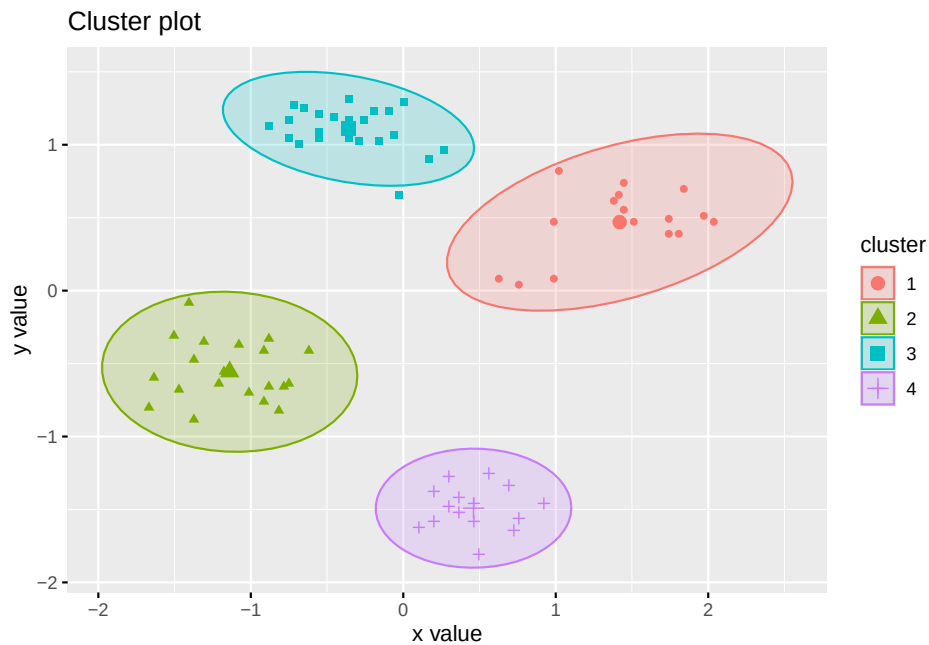


Alternative visualization using `fviz_cluster()`.

```

## __Note:__ `fviz_cluster` needs the original data.
fviz_cluster(c(p, list(data = ruspini_scaled)),
  geom = "point",
  ellipse.type = "norm")

```



7.6.2 Gaussian Mixture Models

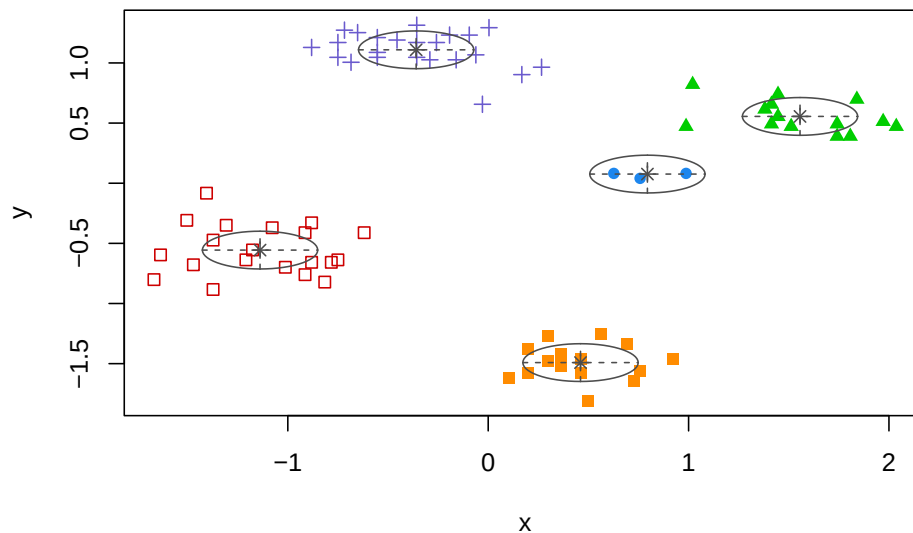
```
library(mclust)
## Package 'mclust' version 6.1.1
## Type 'citation("mclust")' for citing this R package in publications.
##
## Attaching package: 'mclust'
## The following object is masked from 'package:purrr':
##
##      map
```

Gaussian mixture models²⁴ assume that the data set is the result of drawing data from a set of Gaussian distributions where each distribution represents a cluster. Estimation algorithms try to identify the location parameters of the distributions and thus can be used to find clusters. `Mclust()` uses Bayesian Information Criterion (BIC) to find the number of clusters (model selection). BIC uses the likelihood and a penalty term to guard against overfitting.

```
m <- Mclust(ruspini_scaled)
summary(m)
## -----
```

²⁴https://en.wikipedia.org/wiki/Mixture_model#Multivariate_Gaussian_mixture_model

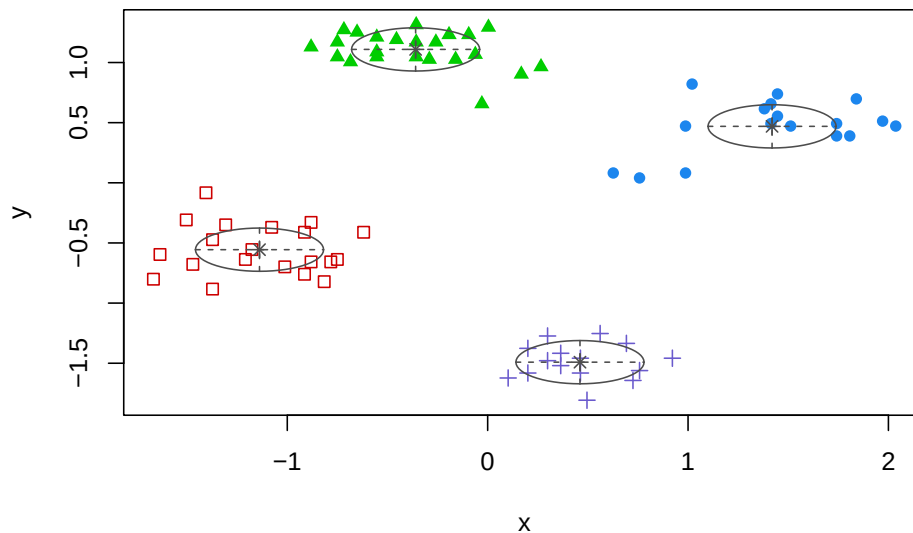
```
## Gaussian finite mixture model fitted by EM algorithm
## -----
##
## Mclust EEI (diagonal, equal volume and shape) model with 5
## components:
##
## log-likelihood n df      BIC      ICL
##          -91.26 75 16 -251.6 -251.7
##
## Clustering table:
##  1  2  3  4  5
##  3 20 14 23 15
plot(m, what = "classification")
```



Rerun with a fixed number of 4 clusters

```
m <- Mclust(ruspini_scaled, G=4)
summary(m)
## -----
## Gaussian finite mixture model fitted by EM algorithm
## -----
##
## Mclust EEI (diagonal, equal volume and shape) model with 4
## components:
##
## log-likelihood n df      BIC      ICL
##          -101.6 75 13 -259.3 -259.3
##
```

```
## Clustering table:
## 1 2 3 4
## 17 20 23 15
plot(m, what = "classification")
```



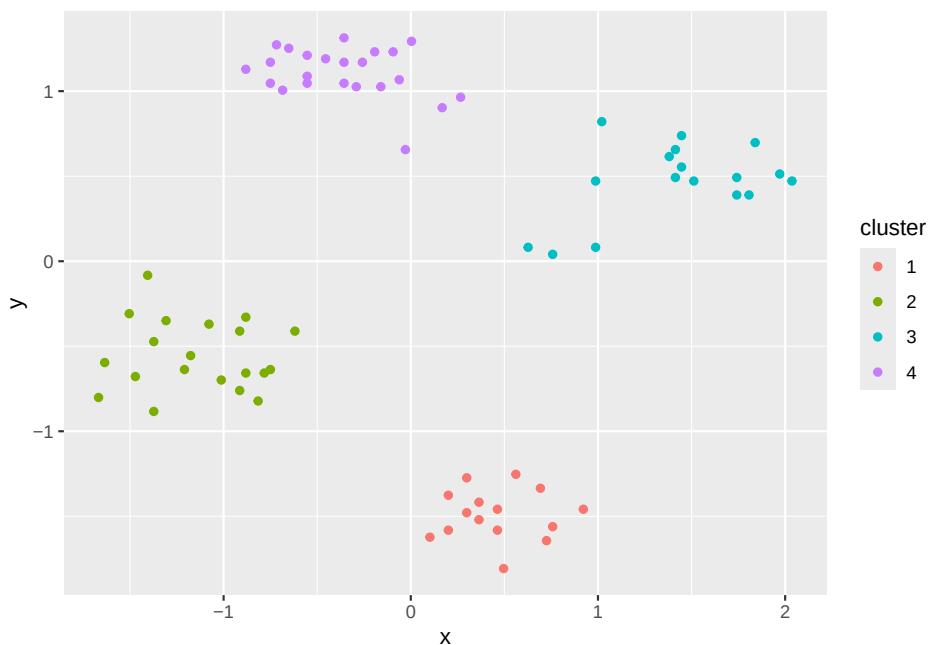
7.6.3 Spectral Clustering

Spectral clustering²⁵ works by embedding the data points of the partitioning problem into the subspace of the k largest eigenvectors of a normalized affinity/kernel matrix. Then uses a simple clustering method like k -means.

```
library("kernlab")
##
## Attaching package: 'kernlab'
## The following object is masked from 'package:arulesSequences':
##
##     size
## The following object is masked from 'package:scales':
##
##     alpha
## The following object is masked from 'package:arules':
##
##     size
## The following object is masked from 'package:purrr':
##
```

²⁵https://en.wikipedia.org/wiki/Spectral_clustering


```
##      cross
## The following object is masked from 'package:ggplot2':
##
##      alpha
cluster_spec <- specc(as.matrix(ruspini_scaled), centers = 4)
cluster_spec
## Spectral Clustering object of class "specc"
##
## Cluster memberships:
##
## 3 2 3 2 3 2 4 3 3 1 1 4 4 3 2 4 3 4 4 3 1 2 2 3 2 1 2 2 4 4 1 4 4 1 1 2 4 1 4 4 4 2 1 1 3 4 3
##
## Gaussian Radial Basis kernel function.
## Hyperparameter : sigma = 74.123288288287
##
## Centers:
##      [,1]      [,2]
## [1,]  0.4607 -1.4912
## [2,] -1.1386 -0.5560
## [3,]  1.4194  0.4693
## [4,] -0.3595  1.1091
##
## Cluster size:
## [1] 15 20 17 23
##
## Within-cluster sum of squares:
## [1] 54.465  9.459 20.360 55.170
ggplot(ruspini_scaled |>
  add_column(cluster = factor(cluster_spec)),
  aes(x, y, color = cluster)) +
  geom_point()
```



7.6.4 Deep Clustering Methods

Deep clustering²⁶ is often used for high-dimensional data. It uses a deep autoencoder to learn a cluster-friendly representation of the data before applying a standard clustering algorithm (e.g., k-means) on the embedded data. The autoencoder is typically implemented using keras²⁷ and the autoencoder's loss function is modified to include the clustering loss which will make the embedding more clusterable.

7.6.5 Fuzzy C-Means Clustering

The fuzzy clustering²⁸ version of the k-means clustering problem. Each data point has a degree of membership to for each cluster.

```
library("e1071")
##
## Attaching package: 'e1071'
## The following object is masked from 'package:ggplot2':
##
```

²⁶<https://arxiv.org/pdf/2210.04142>

²⁷<https://keras3.posit.co/>

²⁸https://en.wikipedia.org/wiki/Fuzzy_clustering

```

##      element
cluster_cmeans <- cmeans(as.matrix(ruspini_scaled), centers = 4)
cluster_cmeans
## Fuzzy c-means clustering with 4 clusters
##
## Cluster centers:
##      x      y
## 1  0.4552 -1.4760
## 2 -0.3763  1.1143
## 3  1.5047  0.5161
## 4 -1.1371 -0.5550
##
## Memberships:
##      1      2      3      4
## [1,] 0.1773846 1.739e-01 5.412e-01 0.1075051
## [2,] 0.0336331 5.308e-02 1.798e-02 0.8953158
## [3,] 0.0405249 4.091e-02 8.956e-01 0.0229249
## [4,] 0.0127760 9.546e-03 4.511e-03 0.9731671
## [5,] 0.1188548 1.100e-01 7.056e-01 0.0654635
## [6,] 0.0740344 3.837e-02 2.108e-02 0.8665086
## [7,] 0.0448221 7.551e-01 1.329e-01 0.0672239
## [8,] 0.1837534 2.177e-01 4.703e-01 0.1283037
## [9,] 0.0135409 1.388e-02 9.650e-01 0.0075753
## [10,] 0.9997656 5.054e-05 7.426e-05 0.0001096
## [11,] 0.9470189 1.083e-02 1.341e-02 0.0287383
## [12,] 0.0139402 9.272e-01 3.405e-02 0.0247849
## [13,] 0.0155182 9.187e-01 2.380e-02 0.0419813
## [14,] 0.0004110 5.072e-04 9.988e-01 0.0002500
## [15,] 0.0358204 2.248e-02 1.143e-02 0.9302635
## [16,] 0.0114703 9.395e-01 2.853e-02 0.0204618
## [17,] 0.0087246 1.483e-02 9.703e-01 0.0061525
## [18,] 0.0078907 9.597e-01 1.728e-02 0.0151643
## [19,] 0.0041345 9.787e-01 6.878e-03 0.0103257
## [20,] 0.0318409 3.391e-02 9.158e-01 0.0184337
## [21,] 0.9531438 9.526e-03 1.187e-02 0.0254625
## [22,] 0.0339789 4.507e-02 1.645e-02 0.9045017
## [23,] 0.0102385 1.357e-02 4.908e-03 0.9712821
## [24,] 0.0009430 1.323e-03 9.971e-01 0.0006088
## [25,] 0.0155195 2.256e-02 7.844e-03 0.9540755
## [26,] 0.9125435 1.727e-02 2.038e-02 0.0498111
## [27,] 0.0290863 2.104e-02 1.010e-02 0.9397668
## [28,] 0.0220730 2.538e-02 9.897e-03 0.9426473
## [29,] 0.0103547 9.467e-01 1.729e-02 0.0256376
## [30,] 0.0004512 9.977e-01 8.879e-04 0.0009643
## [31,] 0.9472526 1.136e-02 1.646e-02 0.0249344

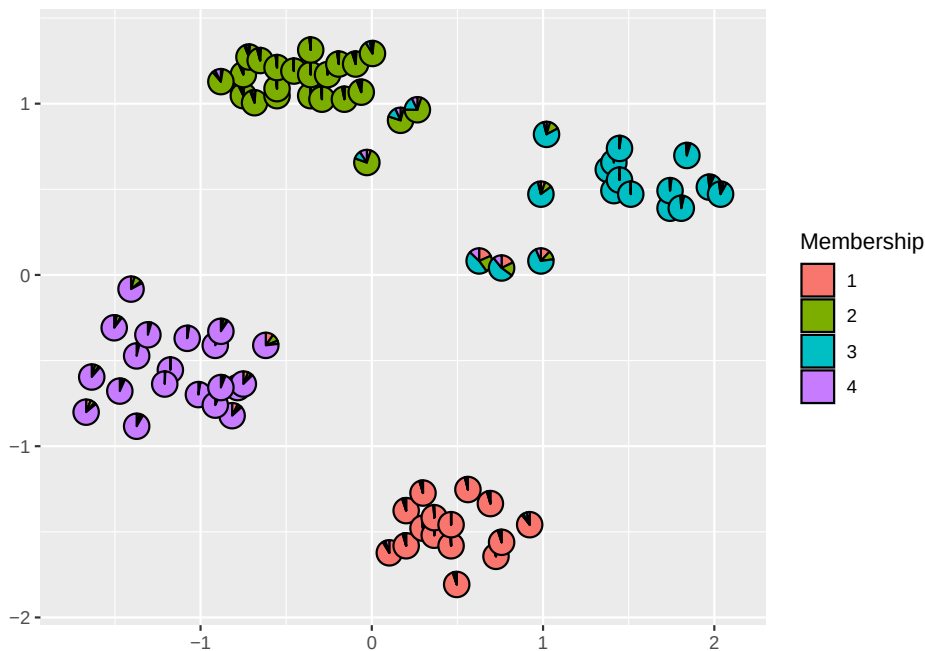
```

```
## [32,] 0.0199940 8.915e-01 5.510e-02 0.0333629
## [33,] 0.0059281 9.693e-01 1.354e-02 0.0112523
## [34,] 0.9503615 1.087e-02 1.818e-02 0.0205929
## [35,] 0.9934967 1.396e-03 2.029e-03 0.0030790
## [36,] 0.0633452 4.260e-02 2.130e-02 0.8727572
## [37,] 0.0007052 9.964e-01 1.322e-03 0.0015602
## [38,] 0.9591000 9.017e-03 1.454e-02 0.0173389
## [39,] 0.0146288 9.245e-01 2.371e-02 0.0371522
## [40,] 0.0519061 7.514e-01 1.047e-01 0.0920679
## [41,] 0.0046347 9.763e-01 9.540e-03 0.0095406
## [42,] 0.0004364 4.471e-04 1.838e-04 0.9989328
## [43,] 0.9839903 3.346e-03 4.423e-03 0.0082407
## [44,] 0.9537432 1.016e-02 1.775e-02 0.0183428
## [45,] 0.0099636 1.131e-02 9.729e-01 0.0058705
## [46,] 0.0021825 9.889e-01 4.269e-03 0.0046274
## [47,] 0.0461263 1.267e-01 7.878e-01 0.0394153
## [48,] 0.0192381 1.929e-02 9.508e-01 0.0106808
## [49,] 0.0536168 9.617e-02 8.110e-01 0.0392583
## [50,] 0.0256674 8.622e-01 3.625e-02 0.0758489
## [51,] 0.0022376 9.885e-01 4.752e-03 0.0044843
## [52,] 0.0047495 9.754e-01 7.760e-03 0.0120640
## [53,] 0.9936329 1.347e-03 1.853e-03 0.0031667
## [54,] 0.8921796 2.343e-02 4.591e-02 0.0384835
## [55,] 0.9492340 1.111e-02 1.926e-02 0.0203873
## [56,] 0.0509714 7.016e-01 1.760e-01 0.0713884
## [57,] 0.0971240 9.312e-02 4.138e-02 0.7683795
## [58,] 0.0549642 3.663e-02 1.831e-02 0.8900882
## [59,] 0.0050410 8.072e-03 9.834e-01 0.0034555
## [60,] 0.9921767 1.664e-03 2.298e-03 0.0038611
## [61,] 0.9523623 1.011e-02 1.337e-02 0.0241501
## [62,] 0.0205460 2.683e-02 9.395e-01 0.0130816
## [63,] 0.0272600 2.687e-02 1.153e-02 0.9343459
## [64,] 0.0138086 1.710e-02 6.486e-03 0.9626083
## [65,] 0.0168049 9.103e-01 2.448e-02 0.0483830
## [66,] 0.0133130 9.294e-01 1.973e-02 0.0376040
## [67,] 0.0431079 4.917e-02 1.999e-02 0.8877335
## [68,] 0.0034024 3.143e-03 1.359e-03 0.9920954
## [69,] 0.0047467 7.442e-03 9.846e-01 0.0032204
## [70,] 0.0048165 9.753e-01 8.408e-03 0.0114693
## [71,] 0.0405047 3.001e-02 1.461e-02 0.9148708
## [72,] 0.0014971 9.924e-01 2.767e-03 0.0033844
## [73,] 0.0018395 2.455e-03 9.945e-01 0.0011595
## [74,] 0.0593827 5.521e-02 2.497e-02 0.8604288
## [75,] 0.0452757 9.817e-02 2.771e-02 0.8288384
##
```

```
## Closest hard clustering:
## [1] 3 4 3 4 3 4 2 3 3 1 1 2 2 3 4 2 3 2 2 3 1 4 4 3 4 1 4 4
## [29] 2 2 1 2 2 1 1 4 2 1 2 2 2 4 1 1 3 2 3 3 3 2 2 2 1 1 1 2
## [57] 4 4 3 1 1 3 4 4 2 2 4 4 3 2 4 2 3 4 4
##
## Available components:
## [1] "centers"      "size"          "cluster"       "membership"
## [5] "iter"         "withinerror"  "call"
```

Plot membership (shown as small pie charts)

```
library("scatterpie")
## scatterpie v0.2.6 Learn more at https://yulab-smu.top/
ggplot() +
  geom_scatterpie(
    data = cbind(ruspini_scaled, cluster_cmeans$membership),
    aes(x = x, y = y),
    cols = colnames(cluster_cmeans$membership),
    legend_name = "Membership") +
  coord_equal()
```



7.7 Scale Issues in Clustering*

To demonstrate this issue, I will use the unscaled Ruspini dataset and assume that we measure x in millimeters and y in meters. I do this by multiplying x by 1000.

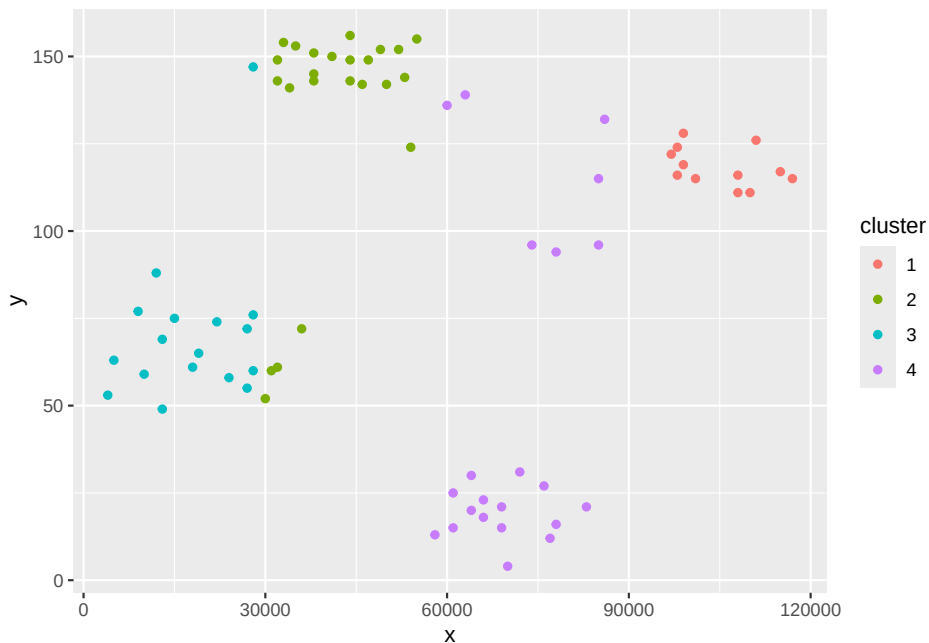
```
ruspini_scale_issue <- ruspini |> mutate(x = x * 1000)
summary(ruspini_scale_issue)
##           x           y
##  Min.   : 4000   Min.   : 4.0
## 1st Qu.:31500   1st Qu.:56.5
##  Median:52000   Median :96.0
##   Mean :54880   Mean   :92.0
## 3rd Qu.:76500   3rd Qu.:141.5
##   Max. :117000   Max.   :156.0
```

If we cluster the data now, then the algorithm fails!

```
km <- kmeans(ruspini_scale_issue, centers = 4)
```

```
ruspini_clustered <- ruspini_scale_issue |>
  add_column(cluster = factor(km$cluster))

ggplot(ruspini_clustered, aes(x = x, y = y)) +
  geom_point(aes(color = cluster))
```



The clusters form vertical bands across the whole plot. The reason is that the large numeric differences for the x-axis overpower the relatively small differences in the y-axis when distances are calculated. This issue can be avoided by scaling all variables first.

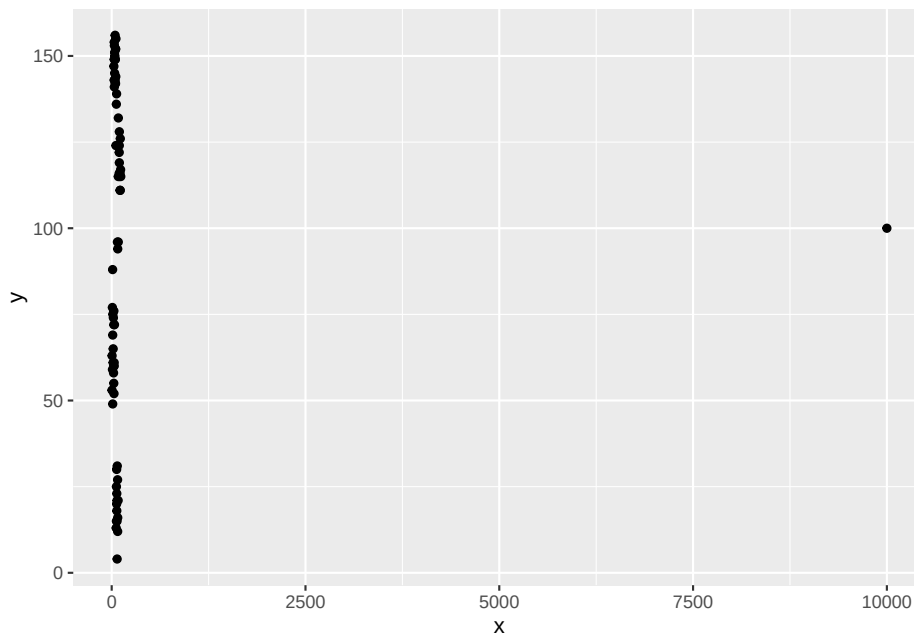
7.8 Outliers in Clustering*

Most clustering algorithms perform complete assignment (i.e., all data points need to be assigned to a cluster). Outliers will affect the clustering.

```
library(dbSCAN)
```

To show this effect, we add a clear outlier with a very large x value to the Ruspini dataset.

```
ruspini_outlier <- ruspini |> add_case(x=10000, y=100)
ggplot(ruspini_outlier, aes(x = x, y = y)) +
  geom_point()
```



The outlier presents a problem for k-means, even if we scale the data. Scaling data with an outlier will make the direction of the outlier (in this case x) the most important.

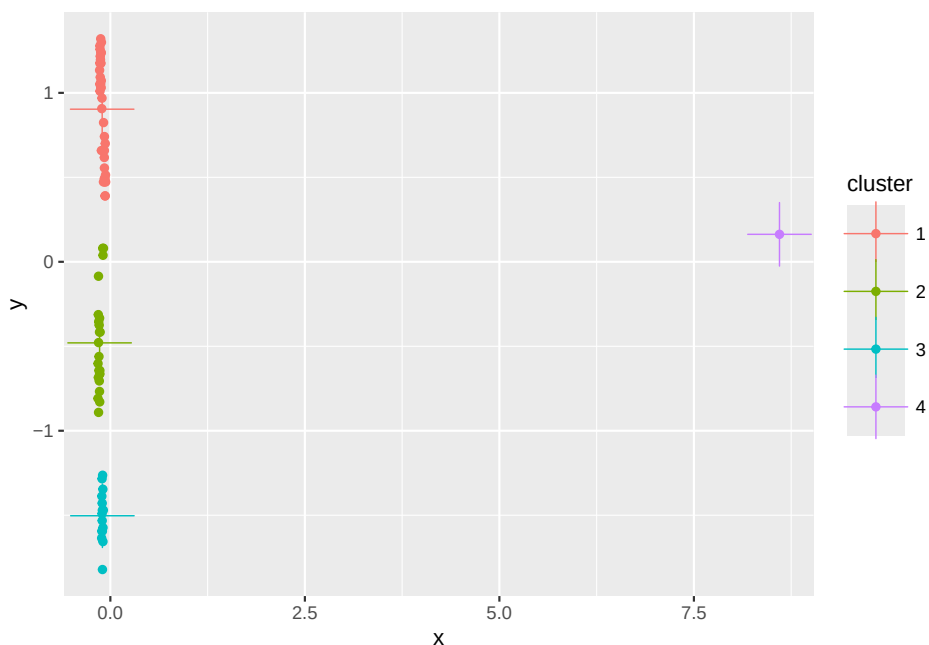
```

ruspini_scaled_outlier <- ruspini_outlier |> scale_numeric()

km <- kmeans(ruspini_scaled_outlier, centers = 4, nstart = 10)
ruspini_scaled_outlier_km <- ruspini_scaled_outlier|>
  add_column(cluster = factor(km$cluster))
centroids <- as_tibble(km$centers, rownames = "cluster")

ggplot(ruspini_scaled_outlier_km, aes(x = x, y = y, color = cluster)) +
  geom_point() +
  geom_point(data = centroids,
             aes(x = x, y = y, color = cluster),
             shape = 3, size = 10)

```



Often, outliers become their own clusters. It is tempting to deal with this issue by adding additional clusters, one per outlier. **But this does not work!**

Here is the result with one additional cluster.

```

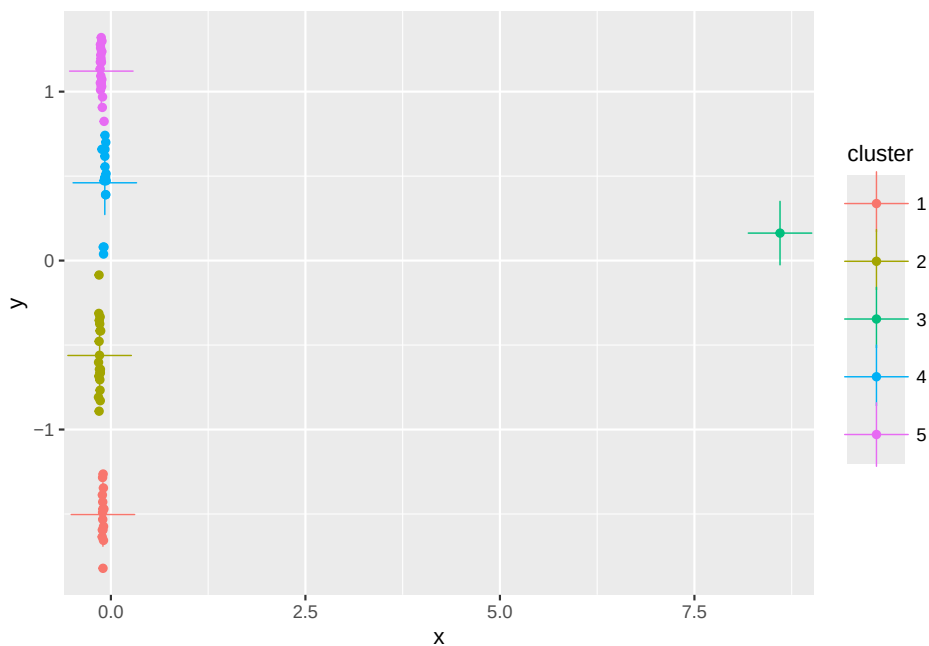
km <- kmeans(ruspini_scaled_outlier, centers = 5, nstart = 10)
ruspini_scaled_outlier_km <- ruspini_scaled_outlier|>
  add_column(cluster = factor(km$cluster))
centroids <- as_tibble(km$centers, rownames = "cluster")

ggplot(ruspini_scaled_outlier_km, aes(x = x, y = y, color = cluster)) +
  geom_point() +

```

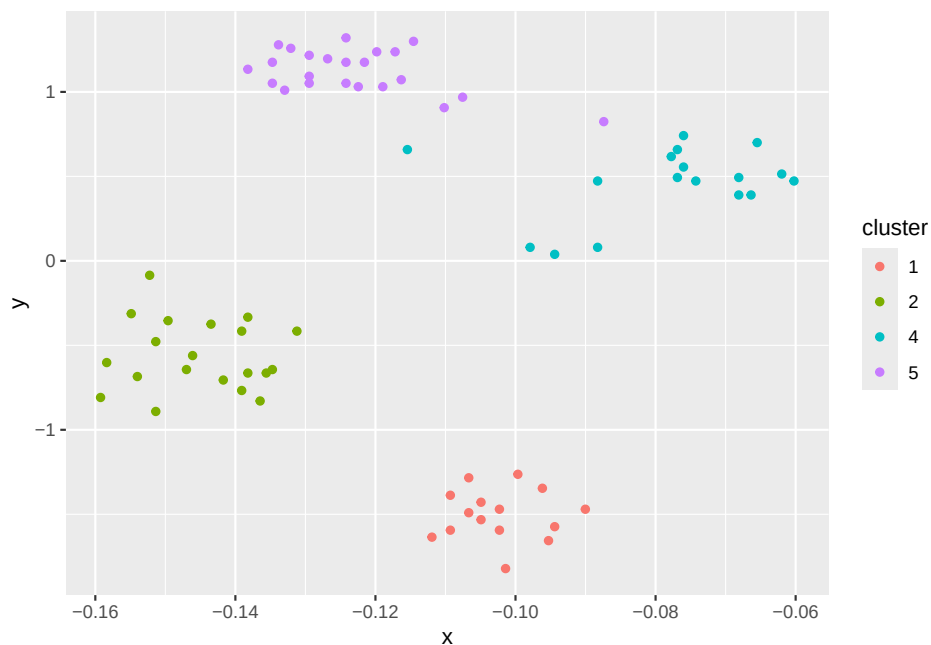


```
geom_point(data = centroids,
           aes(x = x, y = y, color = cluster),
           shape = 3, size = 10)
```



Here it seems to work, but this only happens because the clusters in the Ruspini dataset are nicely spread out over the y axis. The outlier suppresses the effect of variability in the x direction. The top two clusters in the following plot of the clustering without the outlier point shows that the clustering just uses the y axis to cut the data and completely ignores x .

```
ggplot(ruspini_scaled_outlier_km |> filter(row_number() <= n()-1), aes(x = x, y = y, color = cluster))
geom_point()
```

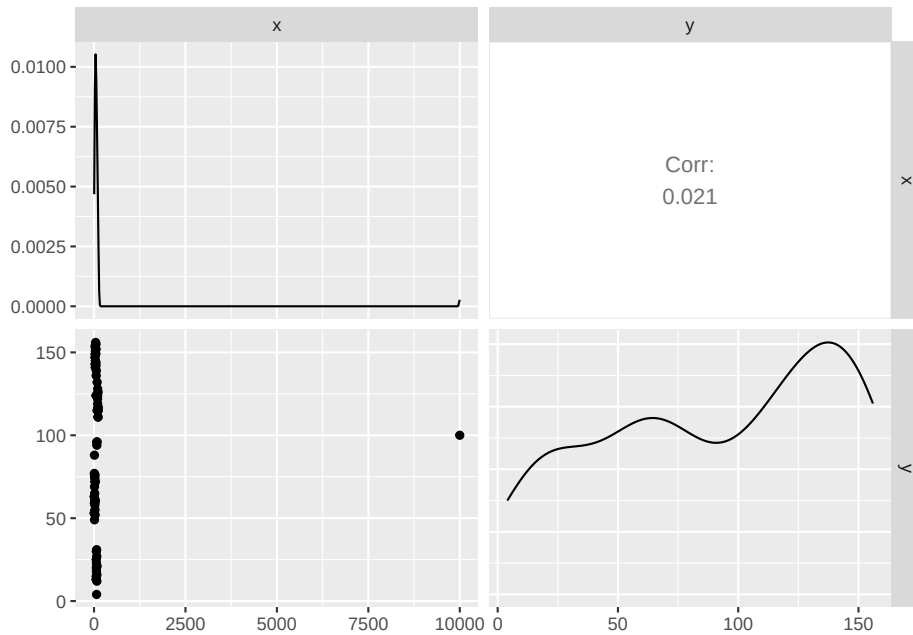


So this does not work! You need to identify and remove outliers before scaling the data. Methods to identify outliers are summary statistics, visual inspection, and outlier scores like the Local Outlier Factor (LOF) discussed in the following sections.

7.8.1 Visual Identification of Outliers

Outliers can be identified using summary statistics, histograms, scatterplots (pairs plots), and boxplots, etc. We use here a pairs plot (the diagonal contains smoothed histograms). The outlier is visible as the single separate point in the scatter plot and as the long tail of the smoothed histogram for x . Remember, in scaled data, we expect most observations to fall in the range $[-3, 3]$.

```
library("GGally")
ggpairs(ruspini_outlier, progress = FALSE)
```



The top-left panel in the plot shows that there is an outlier in the x axis. The x value of regular points is way less than 2500 and we can use this information to identify outliers. We can find the row number by

```
which(ruspini_outlier$x > 2500)
## [1] 76
```

and then manually remove it before scaling the data.

7.8.2 Local Outlier Factor

The Local Outlier Factor (LOF)²⁹ is related to concepts of DBSCAN can help to identify potential outliers. LOF compares the density of a data point to its neighboring data points. If a data point has a much lower density than its neighbors, it has a high LOF score and is considered an outlier.

It is useful to identify outliers and remove strong outliers prior to clustering. A density based method to identify outlier is LOF³⁰ (Local Outlier Factor). It is related to dbSCAN and compares the density around a point with the densities around its neighbors (you have to specify the neighborhood size k). The LOF value for a regular data point is 1. The larger the LOF value gets, the more likely the point is an outlier.

²⁹https://en.wikipedia.org/wiki/Local_outlier_factor

³⁰https://en.wikipedia.org/wiki/Local_outlier_factor

To calculate the LOF, a local neighborhood size (MinPts in DBSCAN) for density estimation needs to be chosen. I use 10 here since I expect my clusters to have each more than 10 points. Note that LOF uses distances, so we need to scale the data first.

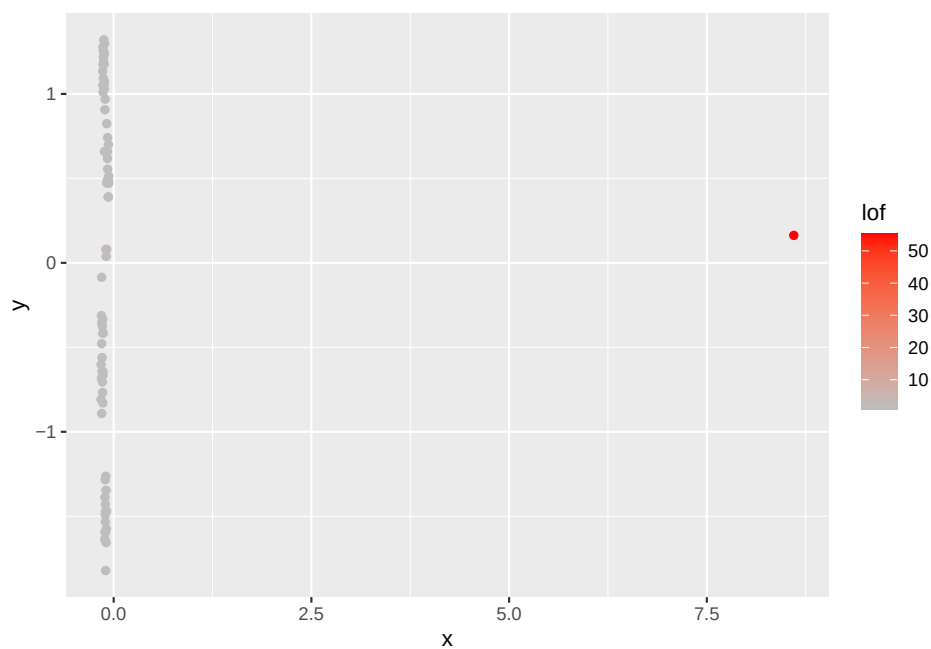
```

ruspini_outlier_scaled <- ruspini_outlier |> scale_numeric()

lof <- lof(ruspini_outlier_scaled, minPts= 10)
lof
## [1] 1.5467 1.1708 0.9915 1.0139 1.8383 1.0785 1.3697
## [8] 1.8401 1.0356 0.9838 1.0938 0.9935 0.9289 0.9915
## [15] 1.0483 1.0289 1.2333 1.0128 1.0003 1.0080 0.9313
## [22] 1.1593 1.1800 0.9082 1.1362 1.0781 0.9667 1.1303
## [29] 1.0533 0.9286 1.6549 1.1418 1.0299 1.1230 0.9313
## [36] 0.9200 1.0122 1.3537 1.0820 1.0523 1.2781 1.0028
## [43] 0.9451 1.1230 1.0078 1.0128 1.3508 1.0356 0.9915
## [50] 0.9999 0.9289 1.0097 0.9626 0.9828 0.9468 1.0623
## [57] 1.1313 0.9664 1.0423 1.0068 1.2959 1.1291 0.9845
## [64] 1.1179 1.0078 1.0342 0.9521 0.9200 0.9541 0.9567
## [71] 1.3222 0.9570 1.0081 1.0159 1.2170 55.3714
ruspini_outlier_scaled <- ruspini_outlier_scaled |> add_column(lof = lof)

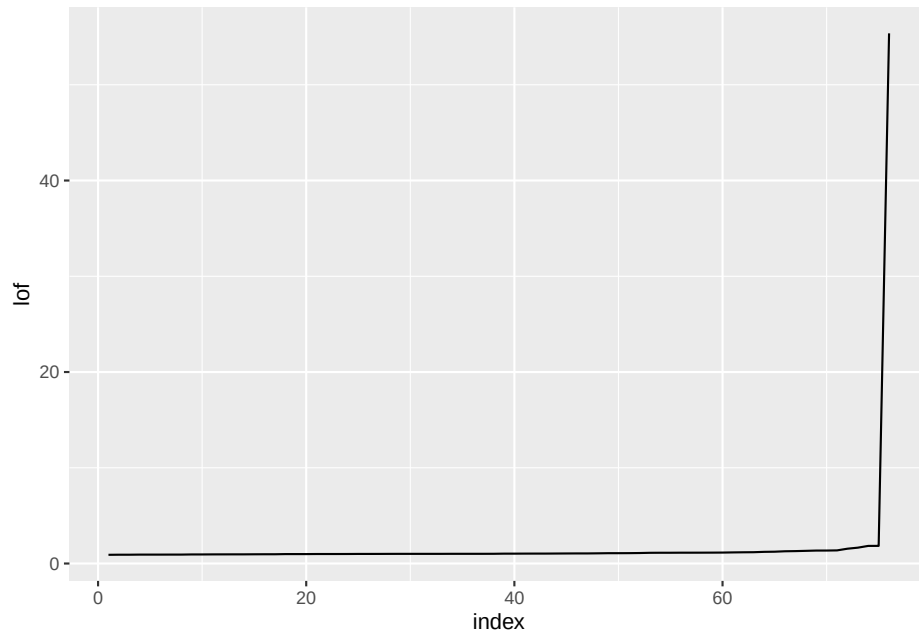
ggplot(ruspini_outlier_scaled,
       aes(x, y, color = lof)) +
  geom_point() +
  scale_color_gradient(low = "gray", high = "red")

```



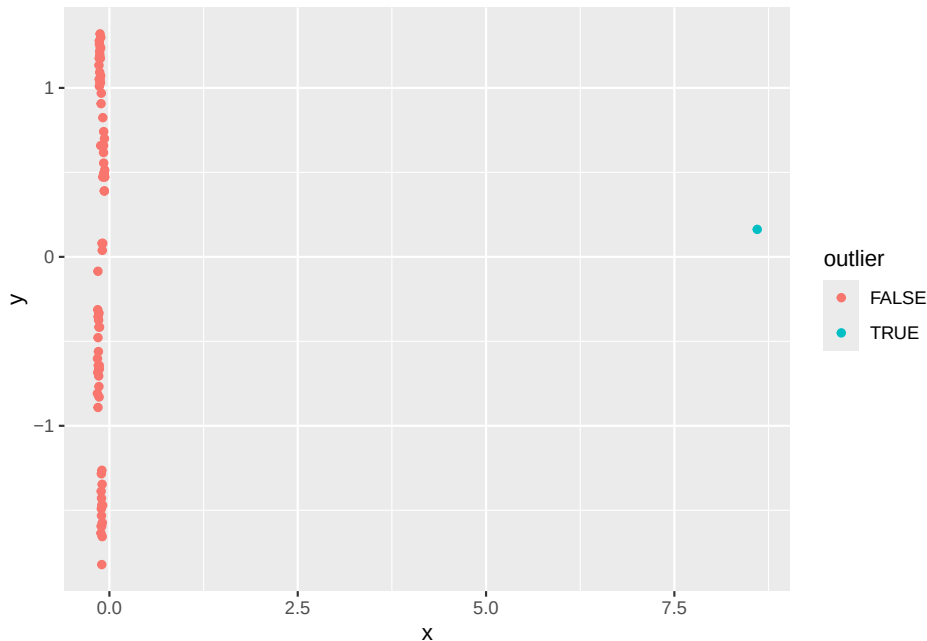
Plot the points sorted by increasing LOF and look for a knee.

```
ggplot(tibble(index = seq_len(length(lof)), lof = sort(lof)),  
  aes(index, lof)) +  
  geom_line()
```



We choose here a threshold above 2.

```
ggplot(ruspini_outlier_scaled |> add_column(outlier = lof >= 2),  
  aes(x, y, color = outlier)) +  
  geom_point()
```



You should analyze the found outliers. They are often interesting and important data points.

To model the regular data, perform clustering on the regular data points without outliers. Note that the data needs to be rescaled after the outlier is removed. This will remove the scaling issues caused by the outlier.

```
ruspini_scaled_clean <- ruspini_outlier_scaled |>
  filter(lof < 2) |>
  select(x, y) |>
  scale_numeric()

km <- kmeans(ruspini_scaled_clean, centers = 4, nstart = 10)
ruspini_scaled_clean_km <- ruspini_scaled_clean|>
  add_column(cluster = factor(km$cluster))
centroids <- as_tibble(km$centers, rownames = "cluster")

ggplot(ruspini_scaled_clean_km, aes(x = x, y = y, color = cluster)) +
  geom_point() +
  geom_point(data = centroids,
            aes(x = x, y = y, color = cluster),
            shape = 3, size = 10)
```



You may have to repeat the identification, removal and rescaling steps several times to find outliers of different scale.

There are many other outlier removal strategies available. See, e.g., package `outliers`³¹.

7.9 Exercises*

We will again use the Palmer penguin data for the exercises.

```
library(palmerpenguins)
head(penguins)
## # A tibble: 6 x 8
##   species island    bill_length_mm bill_depth_mm
##   <chr>    <chr>          <dbl>          <dbl>
## 1 Adelie  Torgersen         39.1           18.7
## 2 Adelie  Torgersen         39.5           17.4
## 3 Adelie  Torgersen         40.3            18
## 4 Adelie  Torgersen          NA            NA
## 5 Adelie  Torgersen         36.7           19.3
## 6 Adelie  Torgersen         39.3           20.6
## # i 4 more variables: flipper_length_mm <dbl>,
## #   body_mass_g <dbl>, sex <chr>, year <dbl>
```

³¹<https://cran.r-project.org/package=outliers>

Create a R markdown file with the code and discussion for the following below.

1. What features do you use for clustering? What about missing values? Discuss your answers. Do you need to scale the data before clustering? Why?
2. What distance measure do you use to reflect similarities between penguins? See Measures of Similarity and Dissimilarity in Chapter 2.
3. Apply k-means clustering. Use an appropriate method to determine the number of clusters. Compare the clustering using unscaled data and scaled data. What is the difference? Visualize and describe the results.
4. Apply hierarchical clustering. Create a dendrogram and discuss what it means.
5. Apply DBSCAN. How do you choose the parameters? How well does it work?

Appendix A

Data Exploration and Visualization

The following code covers the important part of data exploration. For space reasons, this chapter was moved from the printed textbook to this Data Exploration Web Chapter.¹

Packages Used in this Chapter

```
pkgs <- c("arules", "GGally",  
         "ggcorrplot", "hexbin", "palmerpenguins", "plotly", "seriation", "tidyverse")  
  
pkgs_install <- pkgs[!(pkgs %in% installed.packages()[,"Package"])]  
if(length(pkgs_install)) install.packages(pkgs_install)
```

The packages used for this chapter are:

- *arules* (Hahsler et al. 2025)
- *GGally* (Schloerke et al. 2025)
- *ggcorrplot* (Kassambara 2023)
- *hexbin* (Carr et al. 2024)
- *palmerpenguins* (Horst, Hill, and Gorman 2022)
- *plotly* (Sievert et al. 2025)
- *seriation* (Hahsler, Buchta, and Hornik 2025)
- *tidyverse* (Wickham 2023b)

¹https://mhahsler.github.io/Introduction_to_Data_Mining_R_Examples/DM_chapters/data_exploration_1st_edition.pdf

We will use again the iris dataset.

```
library(tidyverse)
data(iris)
iris <- as_tibble(iris)
iris
## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##         <dbl>         <dbl>         <dbl>         <dbl> <fct>
## 1         5.1         3.5         1.4         0.2 setosa
## 2         4.9         3         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         3.6         1.4         0.2 setosa
## 6         5.4         3.9         1.7         0.4 setosa
## 7         4.6         3.4         1.4         0.3 setosa
## 8          5         3.4         1.5         0.2 setosa
## 9         4.4         2.9         1.4         0.2 setosa
## 10        4.9         3.1         1.5         0.1 setosa
## # i 140 more rows
```

A.1 Exploring Data

A.1.1 Basic statistics

Get summary statistics (using base R)

```
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
##
##
##
```

Get mean and standard deviation for sepal length.

```
iris |>
  summarize(avg_Sepal.Length = mean(Sepal.Length),
            sd_Sepal.Length = sd(Sepal.Length))
## # A tibble: 1 x 2
##   avg_Sepal.Length sd_Sepal.Length
##           <dbl>           <dbl>
## 1           5.84           0.828
```

Data with missing values will result in statistics of NA. Adding the parameter `na.rm = TRUE` can be used in most statistics functions to ignore missing values.

```
mean(c(1, 2, NA, 3, 4, 5))
## [1] NA
mean(c(1, 2, NA, 3, 4, 5), na.rm = TRUE)
## [1] 3
```

Outliers are typically the smallest or the largest values of a feature. To make the mean more robust against outliers, we can trim 10% of observations from each end of the distribution.

```
iris |>
  summarize(
    avg_Sepal.Length = mean(Sepal.Length),
    trimmed_avg_Sepal.Length = mean(Sepal.Length, trim = .1)
  )
## # A tibble: 1 x 2
##   avg_Sepal.Length trimmed_avg_Sepal.Length
##           <dbl>           <dbl>
## 1           5.84           5.81
```

Sepal length does not have outliers, so the trimmed mean is almost identical.

To calculate a summary for a set of features (e.g., all numeric features), tidyverse provides `across(where(is.numeric), fun)`.

```
iris |> summarize(across(where(is.numeric), mean))
## # A tibble: 1 x 4
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
##           <dbl>     <dbl>       <dbl>       <dbl>
## 1           5.84       3.06       3.76       1.20
iris |> summarize(across(where(is.numeric), sd))
## # A tibble: 1 x 4
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
##           <dbl>     <dbl>       <dbl>       <dbl>
## 1           0.828     0.436       1.77       0.762
```

```
iris |> summarize(across(where(is.numeric),
  list(min = min,
       median = median,
       max = max)))
## # A tibble: 1 x 12
##   Sepal.Length_min Sepal.Length_median Sepal.Length_max
##   <dbl>           <dbl>           <dbl>
## 1           4.3           5.8           7.9
## # i 9 more variables: Sepal.Width_min <dbl>,
## #   Sepal.Width_median <dbl>, Sepal.Width_max <dbl>,
## #   Petal.Length_min <dbl>, Petal.Length_median <dbl>,
## #   Petal.Length_max <dbl>, Petal.Width_min <dbl>,
## #   Petal.Width_median <dbl>, Petal.Width_max <dbl>
```

The median absolute deviation (MAD) is another measure of dispersion.

```
iris |> summarize(across(where(is.numeric), mad))
## # A tibble: 1 x 4
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
##   <dbl>         <dbl>         <dbl>         <dbl>
## 1       1.04       0.445       1.85       1.04
```

A.1.2 Grouped Operations and Calculations

We can use the nominal feature to form groups and then calculate group-wise statistics for the continuous features. We often use group-wise averages to see if they differ between groups.

```
iris |>
  group_by(Species) |>
  summarize(across(Sepal.Length, mean))
## # A tibble: 3 x 2
##   Species    Sepal.Length
##   <fct>         <dbl>
## 1 setosa         5.01
## 2 versicolor    5.94
## 3 virginica     6.59
iris |>
  group_by(Species) |>
  summarize(across(where(is.numeric), mean))
## # A tibble: 3 x 5
##   Species Sepal.Length Sepal.Width Petal.Length Petal.Width
##   <fct>         <dbl>         <dbl>         <dbl>         <dbl>
## 1 setosa         5.01           3.43           1.46           0.246
```

## 2 versico~	5.94	2.77	4.26	1.33
## 3 virginia~	6.59	2.97	5.55	2.03

We see that the species *Virginica* has the highest average for all, but *Sepal.Width*.

The statistical difference between the groups can be tested using ANOVA (analysis of variance)².

```
res.aov <- aov(Sepal.Length ~ Species, data = iris)
summary(res.aov)
##              Df Sum Sq Mean Sq F value Pr(>F)
## Species      2   63.2   31.61    119 <2e-16 ***
## Residuals   147   39.0    0.27
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
TukeyHSD(res.aov)
##   Tukey multiple comparisons of means
##     95% family-wise confidence level
##
## Fit: aov(formula = Sepal.Length ~ Species, data = iris)
##
## $Species
##              diff      lwr      upr p adj
## versicolor-setosa  0.930 0.6862 1.1738    0
## virginica-setosa   1.582 1.3382 1.8258    0
## virginica-versicolor 0.652 0.4082 0.8958    0
```

The summary shows that there is a significant difference for *Sepal.Length* between the groups. `TukeyHSD` evaluates differences between pairs of groups. In this case, all are significantly different. If the data only contains two groups, the `t.test` can be used.

A.1.3 Tabulate data

We can count the number of flowers for each species.

```
iris |>
  group_by(Species) |>
  summarize(n())
## # A tibble: 3 x 2
```

²<http://www.sthda.com/english/wiki/one-way-anova-test-in-r>

```
## Species      `n()`
## <fct>        <int>
## 1 setosa      50
## 2 versicolor  50
## 3 virginica   50
```

In base R, this can be also done using `count(iris$Species)`.

For the following examples, we discretize the data using `cut`.

```
iris_ord <- iris |>
  mutate(across(where(is.numeric),
    function(x) cut(x, 3, labels = c("short", "medium", "long"),
      ordered = TRUE)))

iris_ord
## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##   <ord>         <ord>         <ord>         <ord>         <fct>
## 1 short        medium      short         short        setosa
## 2 short        medium      short         short        setosa
## 3 short        medium      short         short        setosa
## 4 short        medium      short         short        setosa
## 5 short        medium      short         short        setosa
## 6 short        long         short         short        setosa
## 7 short        medium      short         short        setosa
## 8 short        medium      short         short        setosa
## 9 short        medium      short         short        setosa
## 10 short       medium      short         short        setosa
## # i 140 more rows
summary(iris_ord)
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## short :59      short :47      short :50      short :50
## medium:71      medium:88      medium:54     medium:54
## long  :20      long  :15      long  :46      long  :46
##   Species
## setosa   :50
## versicolor:50
## virginica :50
```

Cross tabulation is used to find out if two discrete features are related.

```
tbl <- iris_ord |>
  select(Sepal.Length, Species) |>
  table()
```



```
tbl
##           Species
## Sepal.Length setosa versicolor virginica
##      short      47         11         1
##      medium      3         36        32
##      long       0          3        17
```

The table contains the number of rows that contain the combination of values (e.g., the number of flowers with a short Sepal.Length and species Setosa is 47). If a few cells have very large counts and most others have very low counts, then there might be a relationship. For the iris data, we see that species Setosa has mostly a short Sepal.Length, while Versicolor and Virginica have longer sepals.

Creating a cross table with tidyverse is a little more involved and uses pivot operations and grouping.

```
iris_ord |>
  select(Species, Sepal.Length) |>
  ### Relationship Between Nominal and Ordinal Features
  pivot_longer(cols = Sepal.Length) |>
  group_by(Species, value) |>
  count() |>
  ungroup() |>
  pivot_wider(names_from = Species, values_from = n)
## # A tibble: 3 x 4
##   value setosa versicolor virginica
##   <ord> <int>      <int>      <int>
## 1 short    47         11         1
## 2 medium     3         36        32
## 3 long     NA          3        17
```

We can use a statistical test to determine if there is a significant relationship between the two features. Pearson's chi-squared test³ for independence is performed with the null hypothesis that the joint distribution of the cell counts in a 2-dimensional contingency table is the product of the row and column marginals. The null hypothesis H_0 is independence between rows and columns.

```
tbl |>
  chisq.test()
##
## Pearson's Chi-squared test
##
## data:  tbl
## X-squared = 112, df = 4, p-value <2e-16
```

³https://en.wikipedia.org/wiki/Chi-squared_test

The small p-value indicates that the null hypothesis of independence needs to be rejected. For small counts (cells with counts <5), Fisher's exact test⁴ is better.

```
fisher.test(tbl)
##
## Fisher's Exact Test for Count Data
##
## data:  tbl
## p-value <2e-16
## alternative hypothesis: two.sided
```

A.1.4 Percentiles (Quantiles)

Quantiles⁵ are cutting points dividing the range of a probability distribution into continuous intervals with equal probability. For example, the median is the empirical 50% quantile dividing the observations into 50% of the observations being smaller than the median and the other 50% being larger than the median.

By default quartiles are calculated. 25% is typically called Q1, 50% is called Q2 or the median and 75% is called Q3.

```
iris |>
  pull(Petal.Length) |>
  quantile()
##    0%  25%  50%  75% 100%
## 1.00 1.60 4.35 5.10 6.90
```

The interquartile range is a measure for variability that is robust against outliers. It is defined the length Q3 - Q2 which covers the 50% of the data in the middle.

```
iris |>
  summarize(IQR =
    quantile(Petal.Length, probs = 0.75) -
    quantile(Petal.Length, probs = 0.25))
## # A tibble: 1 x 1
##       IQR
##   <dbl>
## 1    3.5
```

⁴https://en.wikipedia.org/wiki/Fisher%27s_exact_test

⁵<https://en.wikipedia.org/wiki/Quantile>

A.1.5 Correlation

A.1.5.1 Pearson Correlation

Correlation can be used for ratio/interval scaled features. We typically think of the Pearson correlation coefficient⁶ between features (columns).

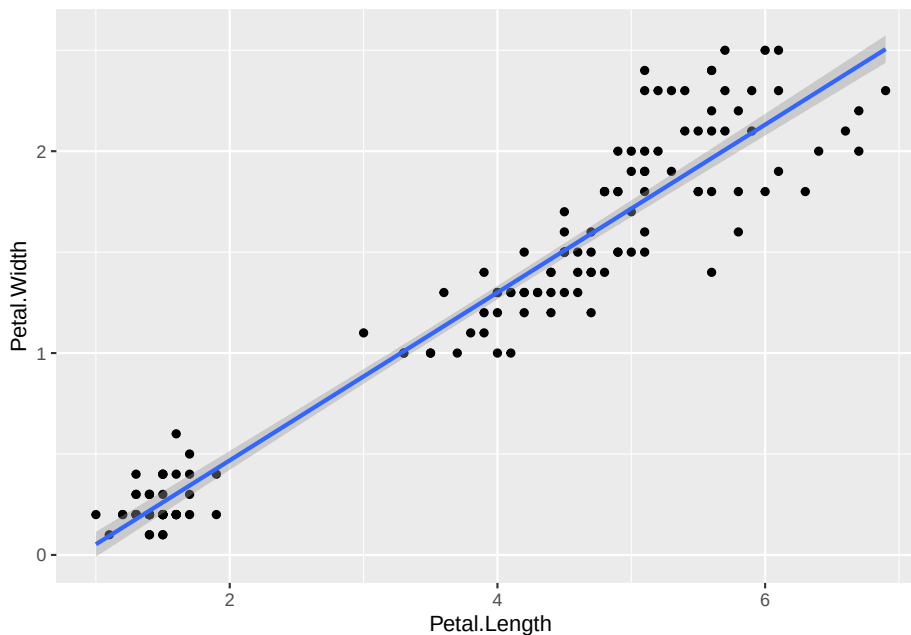
```
cc <- iris |>
  select(-Species) |>
  cor()
cc
##           Sepal.Length Sepal.Width Petal.Length
## Sepal.Length      1.0000      -0.1176      0.8718
## Sepal.Width       -0.1176       1.0000     -0.4284
## Petal.Length       0.8718      -0.4284      1.0000
## Petal.Width        0.8179     -0.3661      0.9629
##           Petal.Width
## Sepal.Length      0.8179
## Sepal.Width      -0.3661
## Petal.Length      0.9629
## Petal.Width       1.0000
```

`cor` calculates a correlation matrix with pairwise correlations between features. Correlation matrices are symmetric, but different to distances, the whole matrix is stored.

The correlation between `Petal.Length` and `Petal.Width` can be visualized using a scatter plot.

```
ggplot(iris, aes(Petal.Length, Petal.Width)) +
  geom_point() +
  geom_smooth(method = "lm")
## `geom_smooth()` using formula = 'y ~ x'
```

⁶https://en.wikipedia.org/wiki/Pearson_correlation_coefficient



`geom_smooth` adds a regression line by fitting a linear model (`lm`). Most points are close to this line indicating strong linear dependence (i.e., high correlation).

We can calculate individual correlations by specifying two vectors.

```
with(iris, cor(Petal.Length, Petal.Width))
## [1] 0.9629
```

Note: `with` lets you use columns using just their names and `with(iris, cor(Petal.Length, Petal.Width))` is the same as `cor(iris$Petal.Length, iris$Petal.Width)`.

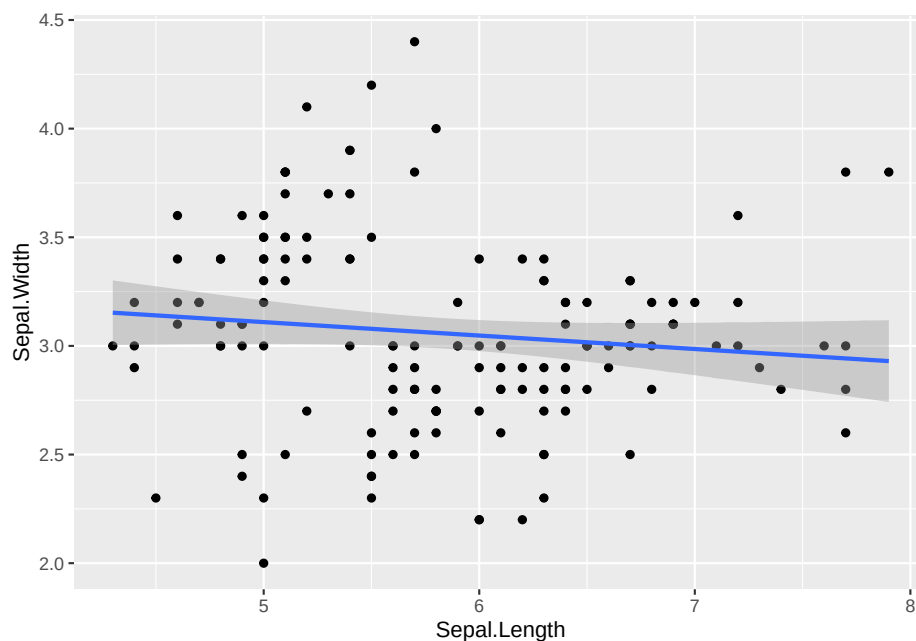
Finally, we can test if a correlation is significantly different from zero.

```
with(iris, cor.test(Petal.Length, Petal.Width))
##
## Pearson's product-moment correlation
##
## data: Petal.Length and Petal.Width
## t = 43, df = 148, p-value <2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
## 0.9491 0.9730
## sample estimates:
## cor
## 0.9629
```

A small p-value (less than 0.05) indicates that the observed correlation is significantly different from zero. This can also be seen by the fact that the 95% confidence interval does not span zero.

Sepal.Length and Sepal.Width show little correlation:

```
ggplot(iris, aes(Sepal.Length, Sepal.Width)) +
  geom_point() +
  geom_smooth(method = "lm")
## `geom_smooth()` using formula = 'y ~ x'
```



```
with(iris, cor(Sepal.Length, Sepal.Width))
## [1] -0.1176
with(iris, cor.test(Sepal.Length, Sepal.Width))
##
## Pearson's product-moment correlation
##
## data: Sepal.Length and Sepal.Width
## t = -1.4, df = 148, p-value = 0.2
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
## -0.27269 0.04351
## sample estimates:
## cor
## -0.1176
```

A.1.5.2 Rank Correlation

Rank correlation is used for ordinal features or if the correlation is not linear. To show this, we first convert the continuous features in the Iris dataset into ordered factors (ordinal) with three levels using the function `cut`.

```
iris_ord <- iris |>
  mutate(across(where(is.numeric),
    function(x) cut(x, 3,
      labels = c("short", "medium", "long"),
      ordered = TRUE)))
```

```
iris_ord
## # A tibble: 150 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##   <ord>         <ord>         <ord>         <ord>         <fct>
## 1 short        medium      short         short        setosa
## 2 short        medium      short         short        setosa
## 3 short        medium      short         short        setosa
## 4 short        medium      short         short        setosa
## 5 short        medium      short         short        setosa
## 6 short        long        short         short        setosa
## 7 short        medium      short         short        setosa
## 8 short        medium      short         short        setosa
## 9 short        medium      short         short        setosa
## 10 short       medium      short         short        setosa
## # i 140 more rows
summary(iris_ord)
##   Sepal.Length Sepal.Width Petal.Length Petal.Width
## short :59      short :47      short :50      short :50
## medium:71      medium:88      medium:54     medium:54
## long  :20      long  :15      long  :46      long  :46
##   Species
## setosa   :50
## versicolor:50
## virginica :50
iris_ord |>
  pull(Sepal.Length)
## [1] short short short short short short short
## [8] short short short short short short short
## [15] medium medium short short medium short short
## [22] short short short short short short short
## [29] short short short short short short short
## [36] short short short short short short short
## [43] short short short short short short short
## [50] short long medium long short medium medium
```

```
## [57] medium short medium short short medium medium
## [64] medium medium medium medium medium medium medium
## [71] medium medium medium medium medium medium long
## [78] medium medium medium short short medium medium
## [85] short medium medium medium medium short short
## [92] medium medium short medium medium medium medium
## [99] short medium medium medium long medium medium
## [106] long short long medium long medium medium
## [113] long medium medium medium medium long long
## [120] medium long medium long medium medium long
## [127] medium medium medium long long long medium
## [134] medium medium long medium medium medium long
## [141] medium long medium long medium medium medium
## [148] medium medium medium
## Levels: short < medium < long
```

Two measures for rank correlation are Kendall's Tau and Spearman's Rho.

Kendall's Tau Rank Correlation Coefficient⁷ measures the agreement between two rankings (i.e., ordinal features).

```
iris_ord |>
  select(-Species) |>
  sapply(xtfrm) |>
  cor(method = "kendall")
##           Sepal.Length Sepal.Width Petal.Length
## Sepal.Length      1.0000      -0.1438      0.7419
## Sepal.Width       -0.1438       1.0000     -0.3299
## Petal.Length       0.7419     -0.3299      1.0000
## Petal.Width        0.7295     -0.3154      0.9198
##           Petal.Width
## Sepal.Length      0.7295
## Sepal.Width       -0.3154
## Petal.Length       0.9198
## Petal.Width        1.0000
```

Note: We have to use `xtfrm` to transform the ordered factors into ranks, i.e., numbers representing the order.

Spearman's Rho⁸ is equal to the Pearson correlation between the rank values of those two features.

⁷https://en.wikipedia.org/wiki/Kendall_rank_correlation_coefficient

⁸https://en.wikipedia.org/wiki/Spearman%27s_rank_correlation_coefficient

```
iris_ord |>
  select(-Species) |>
  sapply(xtfrm) |>
  cor(method = "spearman")
##           Sepal.Length Sepal.Width Petal.Length
## Sepal.Length      1.0000      -0.1570      0.7938
## Sepal.Width       -0.1570       1.0000     -0.3663
## Petal.Length       0.7938     -0.3663      1.0000
## Petal.Width        0.7843     -0.3517      0.9399
##           Petal.Width
## Sepal.Length      0.7843
## Sepal.Width       -0.3517
## Petal.Length       0.9399
## Petal.Width        1.0000
```

Spearman's Rho is much faster to compute on large datasets than Kendall's Tau.

Comparing the rank correlation results with the Pearson correlation on the original data shows that they are very similar. This indicates that discretizing data does not result in the loss of too much information.

```
iris |>
  select(-Species) |>
  cor()
##           Sepal.Length Sepal.Width Petal.Length
## Sepal.Length      1.0000     -0.1176      0.8718
## Sepal.Width       -0.1176       1.0000     -0.4284
## Petal.Length       0.8718     -0.4284      1.0000
## Petal.Width        0.8179     -0.3661      0.9629
##           Petal.Width
## Sepal.Length      0.8179
## Sepal.Width       -0.3661
## Petal.Length       0.9629
## Petal.Width        1.0000
```

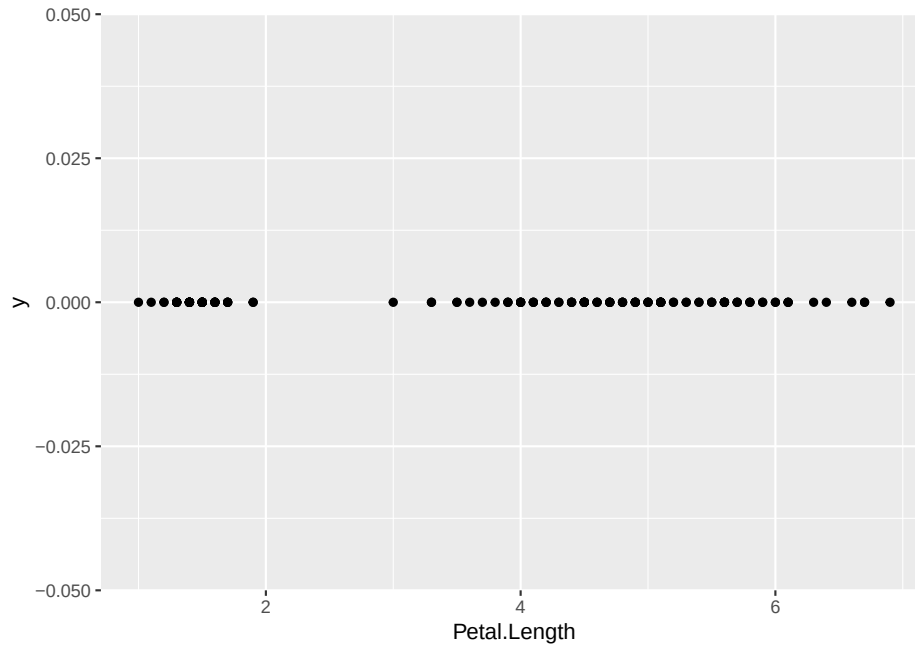
A.1.6 Density

Density estimation⁹ estimate the probability density function (distribution) of a continuous variable from observed data.

Just plotting the data using points is not very helpful for a single feature.

⁹https://en.wikipedia.org/wiki/Density_estimation


```
ggplot(iris, aes(x = Petal.Length, y = 0)) + geom_point()
```

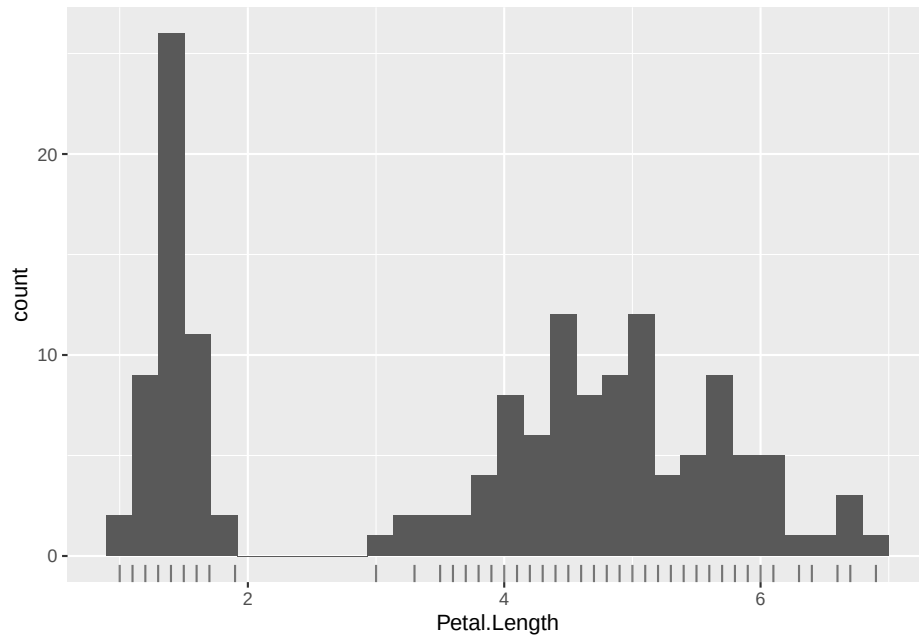


A.1.6.1 Histograms

A histograms¹⁰ shows more about the distribution by counting how many values fall within a bin and visualizing the counts as a bar chart. We use `geom_rug` to place marks for the original data points at the bottom of the histogram.

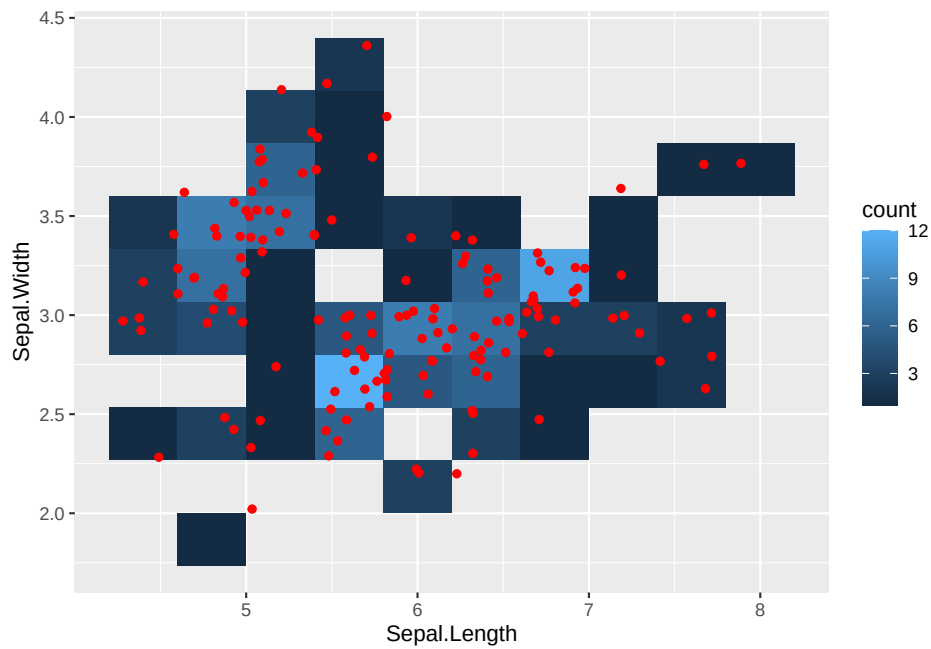
```
ggplot(iris, aes(x = Petal.Length)) +
  geom_histogram() +
  geom_rug(alpha = 1/2)
## `stat_bin()` using `bins = 30`. Pick better value
## `binwidth`.
```

¹⁰<https://en.wikipedia.org/wiki/Histogram>

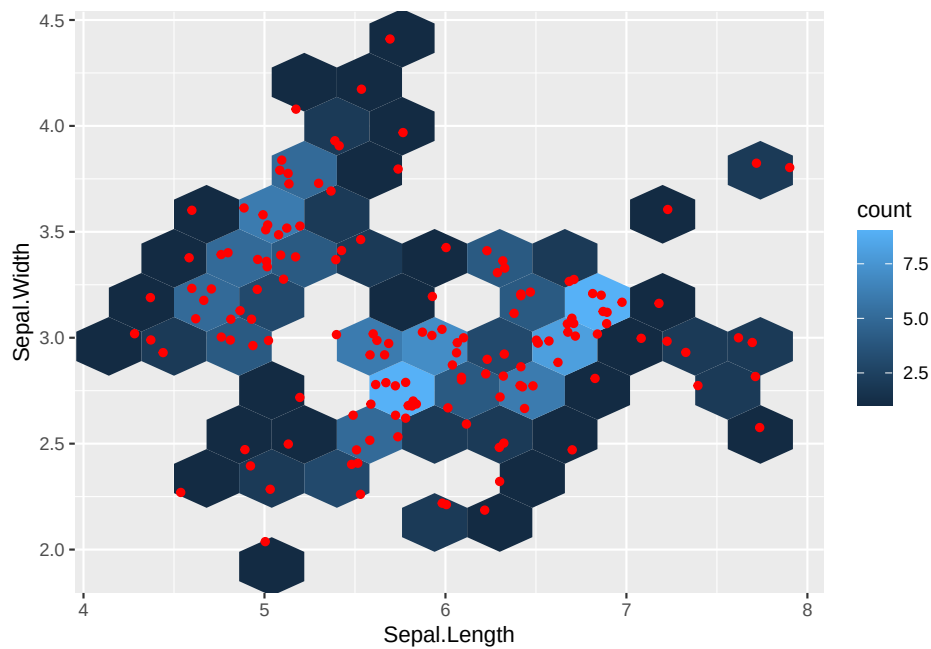


Two-dimensional distributions can be visualized using 2-d binning or hexagonal bins.

```
ggplot(iris, aes(Sepal.Length, Sepal.Width)) +  
  geom_bin2d(bins = 10) +  
  geom_jitter(color = "red")
```



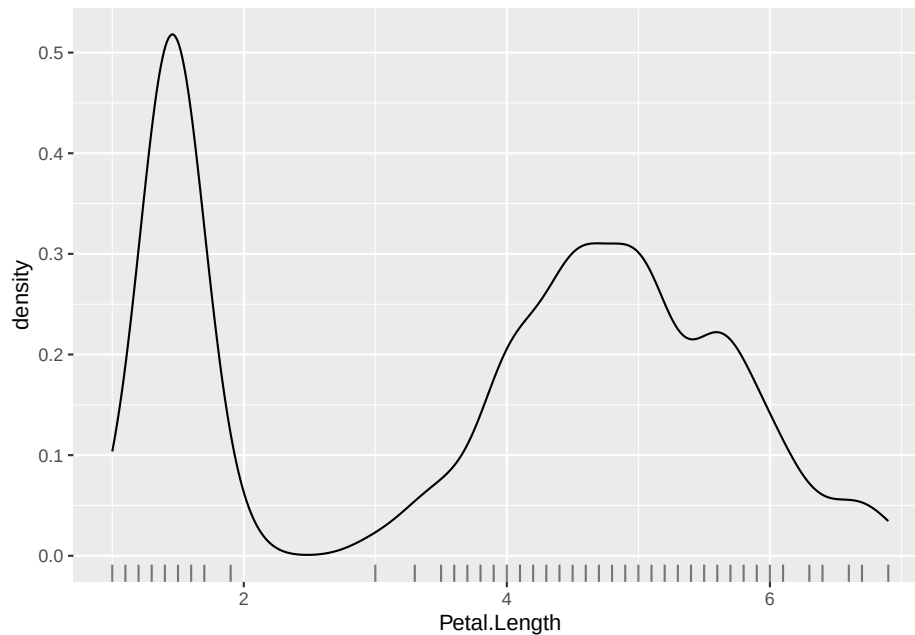
```
ggplot(iris, aes(Sepal.Length, Sepal.Width)) +  
  geom_hex(bins = 10) +  
  geom_jitter(color = "red")
```



A.1.6.2 Kernel Density Estimate (KDE)

Kernel density estimation¹¹ is used to estimate the probability density function (distribution) of a feature. It works by replacing each value with a kernel function (often a Gaussian) and then adding them up. The result is an estimated probability density function that looks like a smoothed version of the histogram. The bandwidth (bw) of the kernel controls the amount of smoothing.

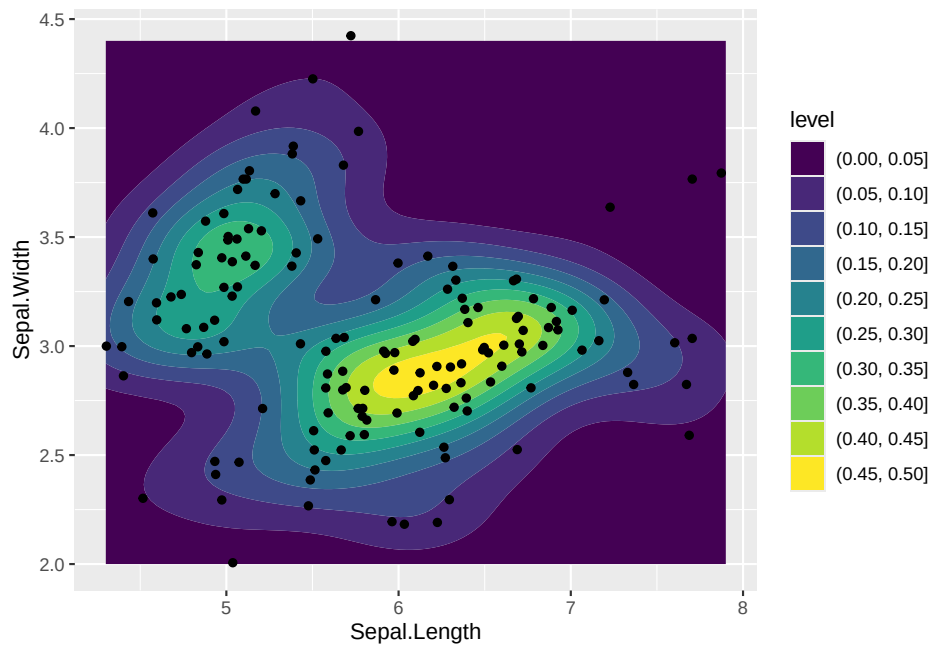
```
ggplot(iris, aes(Petal.Length)) +  
  geom_density(bw = .2) +  
  geom_rug(alpha = 1/2)
```



Kernel density estimates can also be done in two dimensions.

```
ggplot(iris, aes(Sepal.Length, Sepal.Width)) +  
  geom_density_2d_filled() +  
  geom_jitter()
```

¹¹https://en.wikipedia.org/wiki/Kernel_density_estimation



A.2 Visualization

Visualization uses several components to convey information:

- Symbols: Circles, dots, lines, bars, ...
- Position: Axes (labels and units) and the origin (0/0) are important.
- Length, Size and Area: Should only be used if it faithfully represents information.
- Color: A color should not overpower another. Limit to 3 or 4 colors.
- Angle: Human eyes are not good at comparing angles!

All components of the plot need to convey information. E.g., do not use color just to make it more colorful. All good visualizations show the important patterns clearly.

For space reasons, the chapter on data exploration and visualization was moved from the printed textbook and can now be found in the Data Exploration Web Chapter.¹²

The most important R-base plot functions are `plot()`, `barplot()`, `hist()`, and `pairs()`. Here, we will use mainly the more flexible `ggplot2` library. The *gg* in `ggplot2` stands for **Grammar of Graphics** introduced by Wilkinson (2005). The main idea is that every graph is built from the same basic components:

¹²https://www-users.cse.umn.edu/~kumar001/dmbook/data_exploration_1st_edition.pdf

- the data,
- a coordinate system, and
- visual marks representing the data (geoms).

In `ggplot2`, the components are combined using the `+` operator.

```
ggplot(data, mapping = aes(x = ..., y = ..., color = ...)) +  
  geom_point()
```

Since we typically use a Cartesian coordinate system, `ggplot` uses that by default. Each `geom_` function uses a `stat_` function to calculate what is visualized. For example, `geom_bar` uses `stat_count` to create a bar chart by counting how often each value appears in the data (see `? geom_bar`). `geom_point` just uses the `stat "identity"` to display the points using the coordinates as they are.

Additional components like the main title, axis labels, and different scales can be also added. A great introduction can be found in the Section on Data Visualization¹³ (Wickham, Çetinkaya-Rundel, and Grolemund 2023), and very useful is RStudio's Data Visualization Cheatsheet¹⁴.

Next, we go through the basic visualizations used when working with data.

A.2.1 Histogram

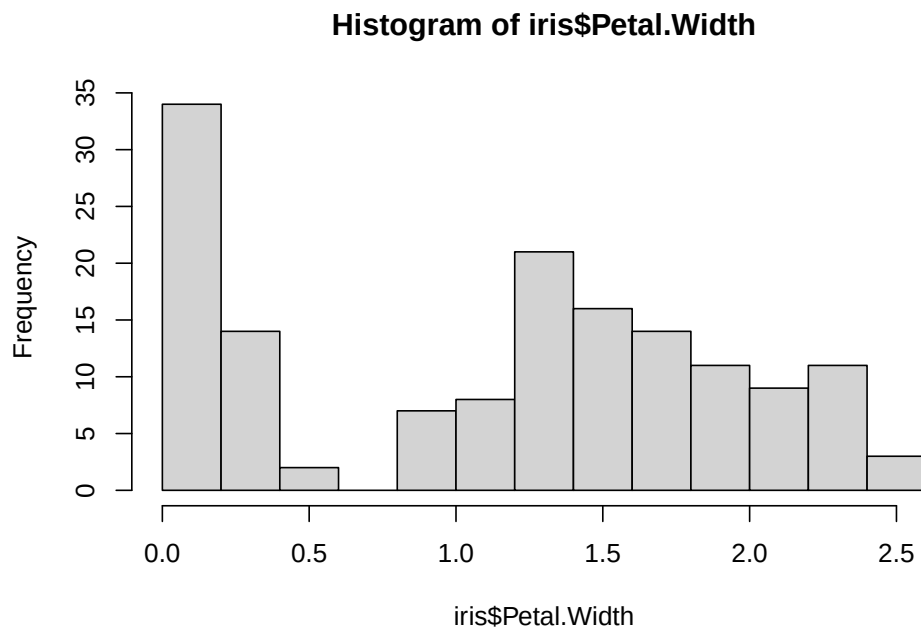
Histograms show the distribution of a single continuous feature. The x-axis cuts the variable into discrete buckets and the y-axis shows how many observations fall into each bucket. This plot should be used to inspect each continuous variable to:

- Detect outliers (i.e., single observation that is very far to the right in the plot) or recording mistakes (e.g., a too large frequency at 0 may indicate that missing values were by mistake converted to 0).
- Understand the distribution. Do we have a single normal distribution or are there several peaks which can indicate that there multiple groups in the data.

```
hist(iris$Petal.Width)
```

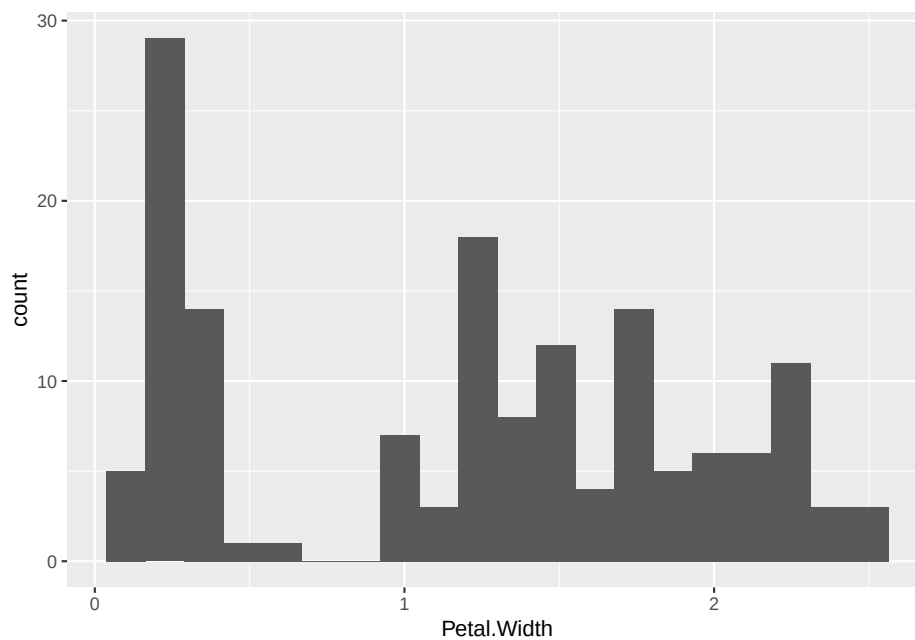
¹³<https://r4ds.hadley.nz/data-visualize>

¹⁴<https://rstudio.github.io/cheatsheets/html/data-visualization.html>



The ggplot equivalent uses the histogram geometry.

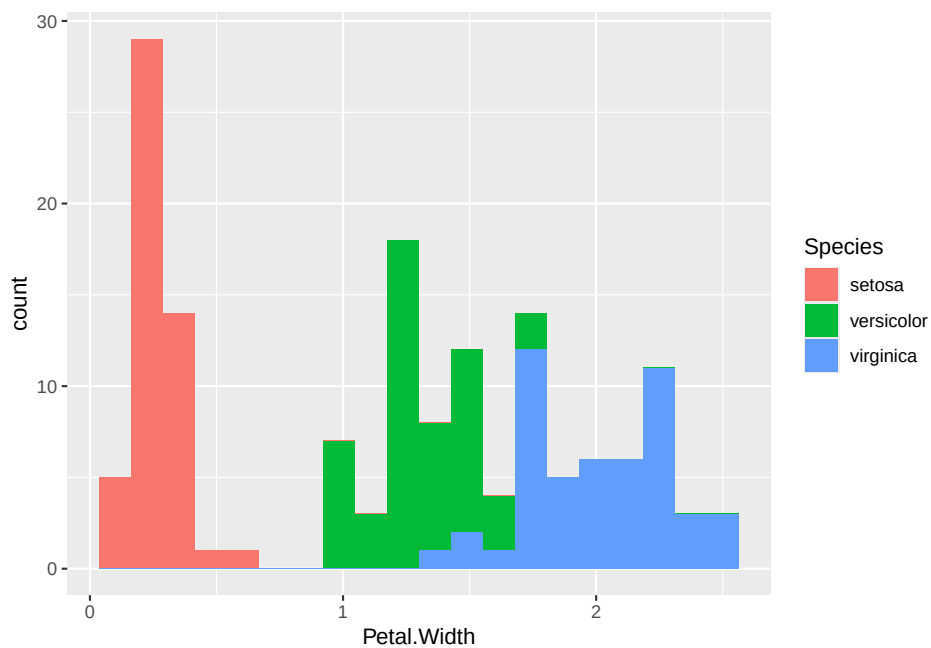
```
ggplot(iris, aes(Petal.Width)) + geom_histogram(bins = 20)
```



The plot shows not just a single normal distribution, but at least two peaks indicating that the data may be a mixture of two or three different groups.

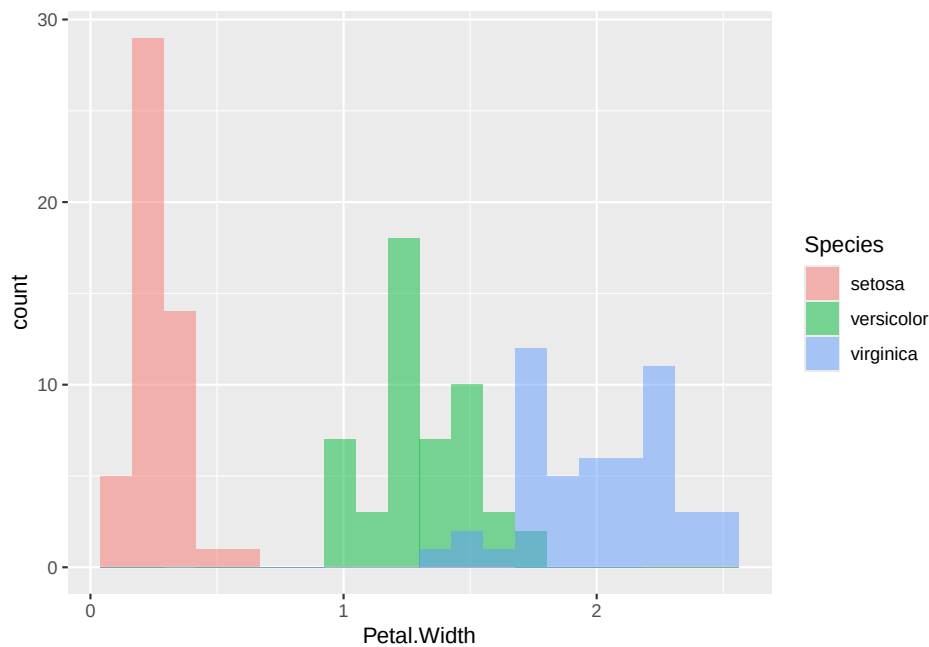
We know that the Iris data set contains three groups (types of iris) and we have a variable indicating the group. This group information can be easily added as an aesthetic to the histogram.

```
ggplot(iris, aes(Petal.Width)) +  
  geom_histogram(bins = 20, aes(fill = Species))
```



The bars appear stacked with a green blocks on top of blue blocks. To display the three distributions behind each other, we change `position` for placement and make the bars slightly translucent using `alpha`.

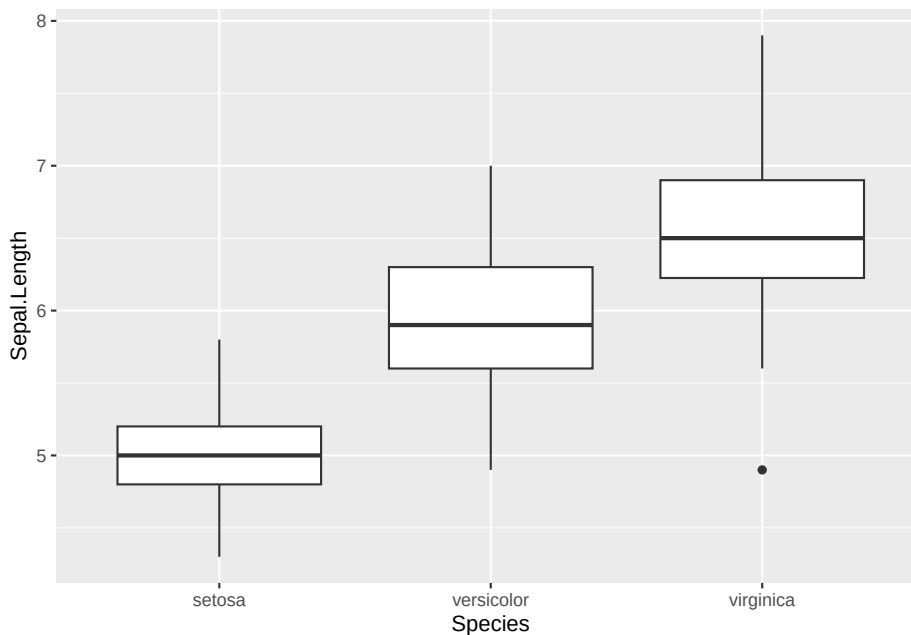
```
ggplot(iris, aes(Petal.Width)) +  
  geom_histogram(bins = 20, aes(fill = Species), alpha = .5, position = 'identity')
```

A.2.2 Boxplot

Boxplots are used to compare the distribution of a feature between different groups. The horizontal line in the middle of the boxes are the group-wise medians, the boxes span the interquartile range. The whiskers (vertical lines) span typically 1.4 times the interquartile range. Points that fall outside that range are typically outliers shown as dots.

```
ggplot(iris, aes(Species, Sepal.Length)) +  
  geom_boxplot()
```



The Iris data has very few outliers and only the group Virginia shows a single dot. We can also see that the box for Setosa is much lower than the other two, indicating that this group has a much smaller sepal length.

The group-wise medians can also be calculated directly.

```
iris |> group_by(Species) |>
  summarize(across(where(is.numeric), median))
```

A tibble: 3 x 5

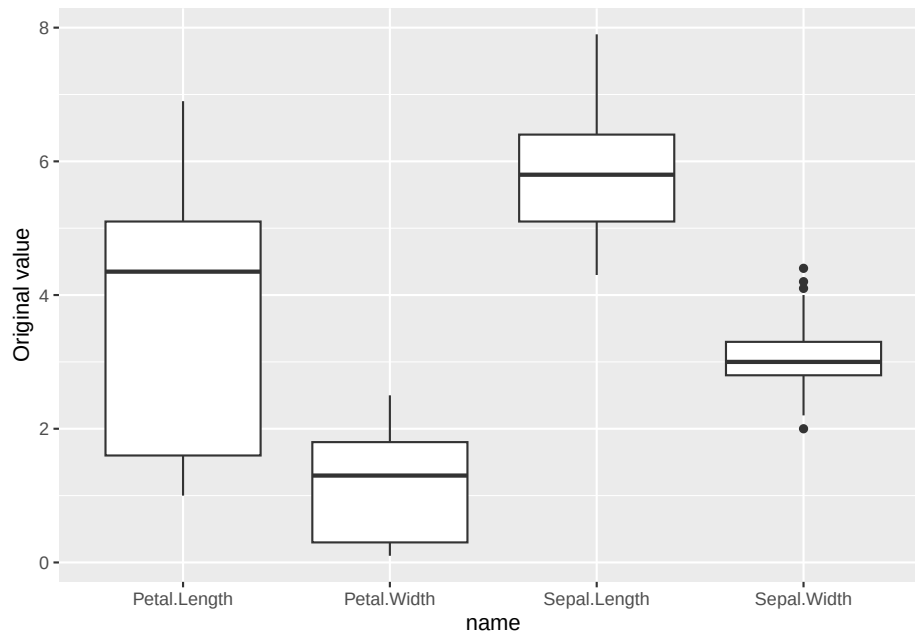
Species	Sepal.Length	Sepal.Width	Petal.Length	Petal.Width
setosa	5	3.4	1.5	0.2
versicolor	5.9	2.8	4.35	1.3
virginica	6.5	3	5.55	2

To compare the distribution of the four features using a ggplot boxplot, we first have to transform the data into long format (i.e., all feature values are combined into a single column).

```
library(tidyr)
iris_long <- iris |>
  mutate(id = row_number()) |>
  pivot_longer(1:4)

ggplot(iris_long, aes(name, value)) +
```

```
geom_boxplot() +
labs(y = "Original value")
```



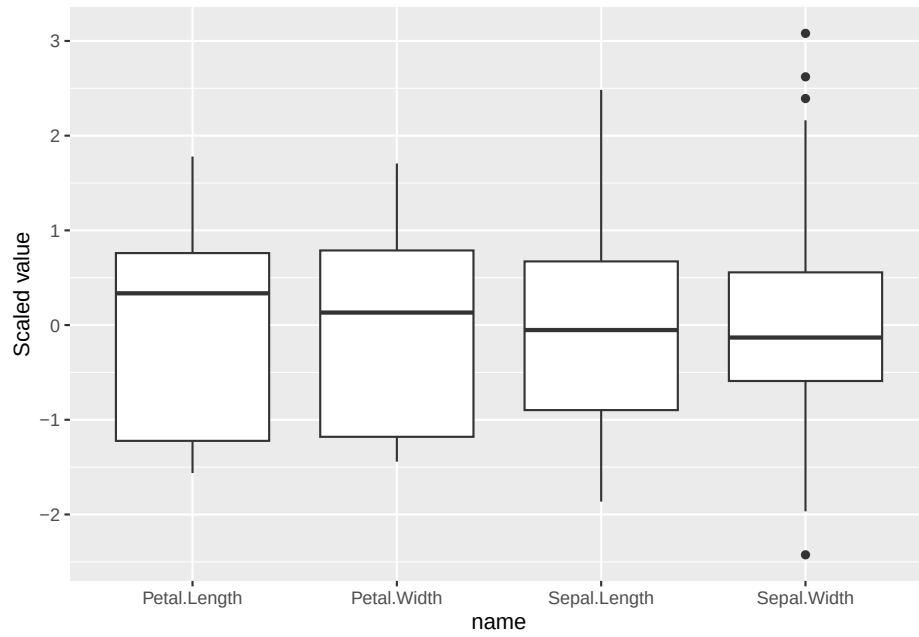
This visualization is only useful if all features have roughly the same range. The data can be scaled first to compare the distributions.

```
library(tidyr)

scale_numeric <- function(x)
  x |>
  mutate(across(where(is.numeric),
    function(y) (y - mean(y, na.rm = TRUE)) / sd(y, na.rm = TRUE)))

iris_long_scaled <- iris |>
  scale_numeric() |>
  mutate(id = row_number()) |> pivot_longer(1:4)

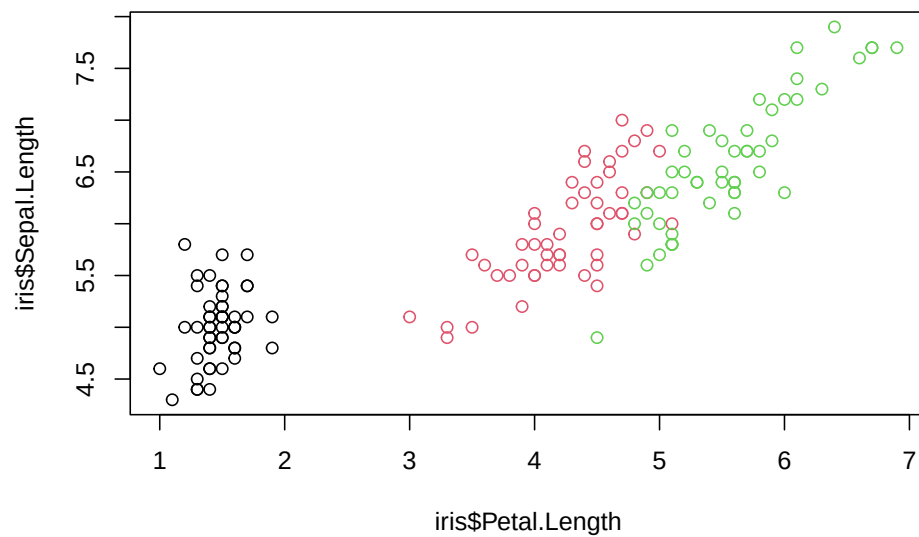
ggplot(iris_long_scaled, aes(name, value)) +
  geom_boxplot() +
  labs(y = "Scaled value")
```



A.2.3 Scatter plot

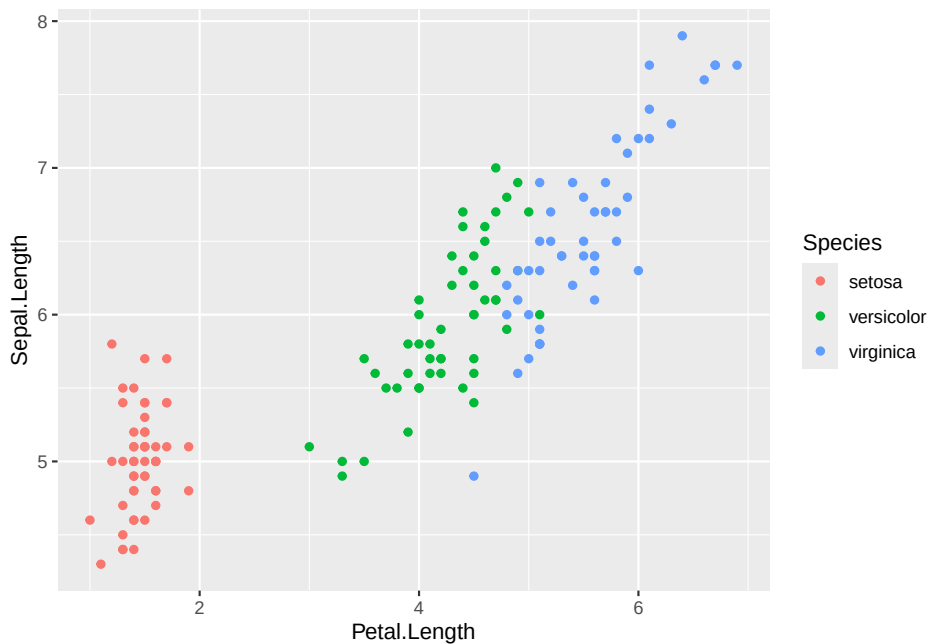
Scatter plots show the relationship between two continuous features.

```
plot(iris$Petal.Length, iris$Sepal.Length, col = iris$Species)
```



ggplot uses the point geometry.

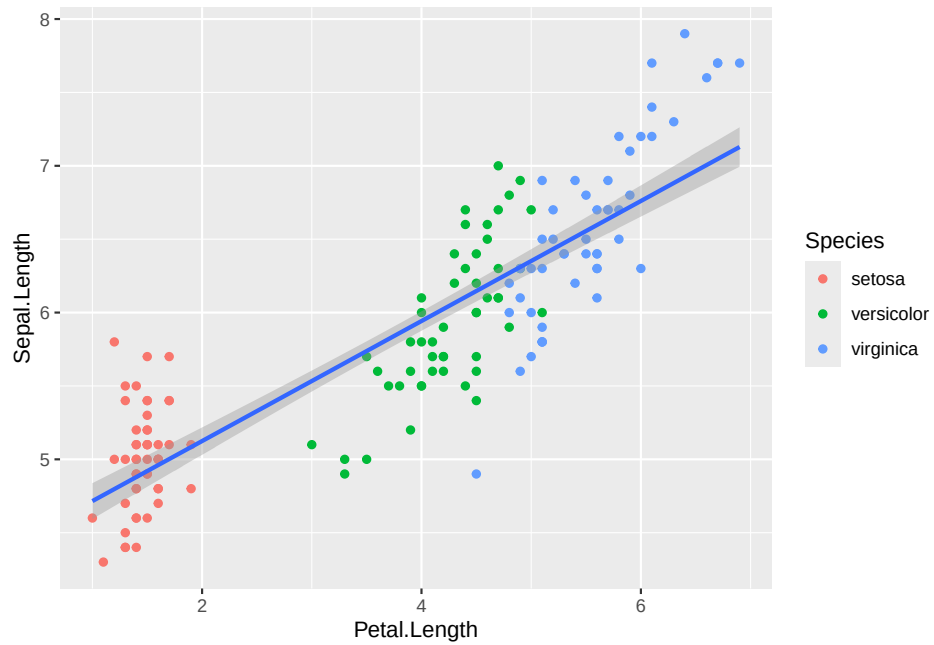
```
ggplot(iris, aes(x = Petal.Length, y = Sepal.Length)) +  
  geom_point(aes(color = Species))
```



We see that Setosa flowers are well separated from the other two flowers. It would be easy to identify them by a simple threshold of 2.5 cm on petal length. The other two species are closer to each other and mix somewhat together. Outliers would be single points that are far from other points. This data set does not contain outliers.

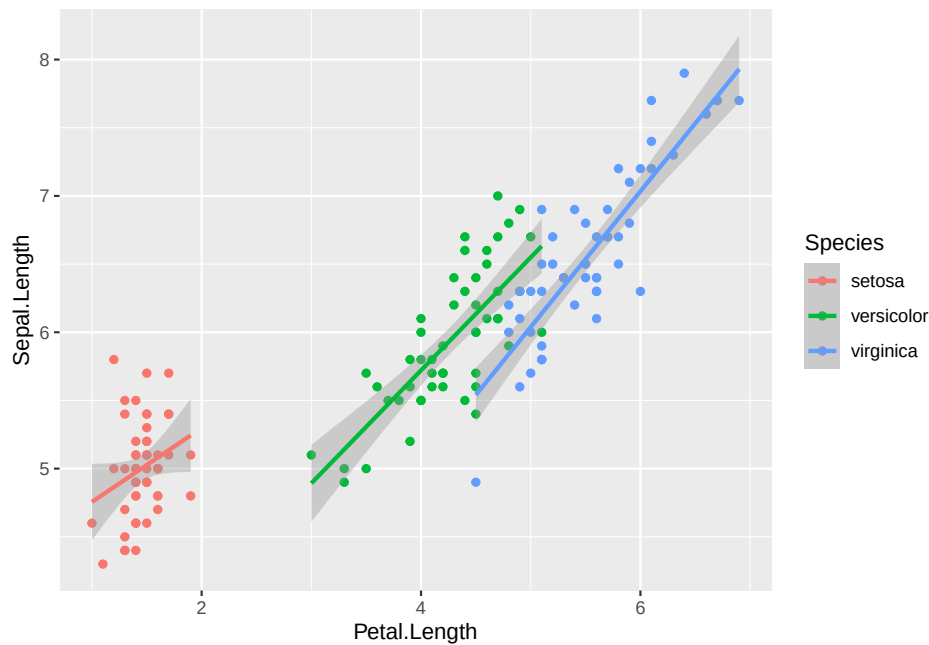
We can also identify correlation between the variables using this plot. Here, we see that Sepal length tends to increase when petal length increases indicating a positive linear correlation between the variables. This relationship can be visualized by adding a regression line using `geom_smooth` with the linear model method ("lm"). A confidence interval for the regression is also shown. This can be suppressed using `se = FALSE`.

```
ggplot(iris, aes(x = Petal.Length, y = Sepal.Length)) +  
  geom_point(aes(color = Species)) +  
  geom_smooth(method = "lm")  
## `geom_smooth()` using formula = 'y ~ x'
```



We can also perform group-wise linear regression by adding the color aesthetic also to `geom_smooth`.

```
ggplot(iris, aes(x = Petal.Length, y = Sepal.Length)) +  
  geom_point(aes(color = Species)) +  
  geom_smooth(method = "lm", aes(color = Species))  
## `geom_smooth()` using formula = 'y ~ x'
```



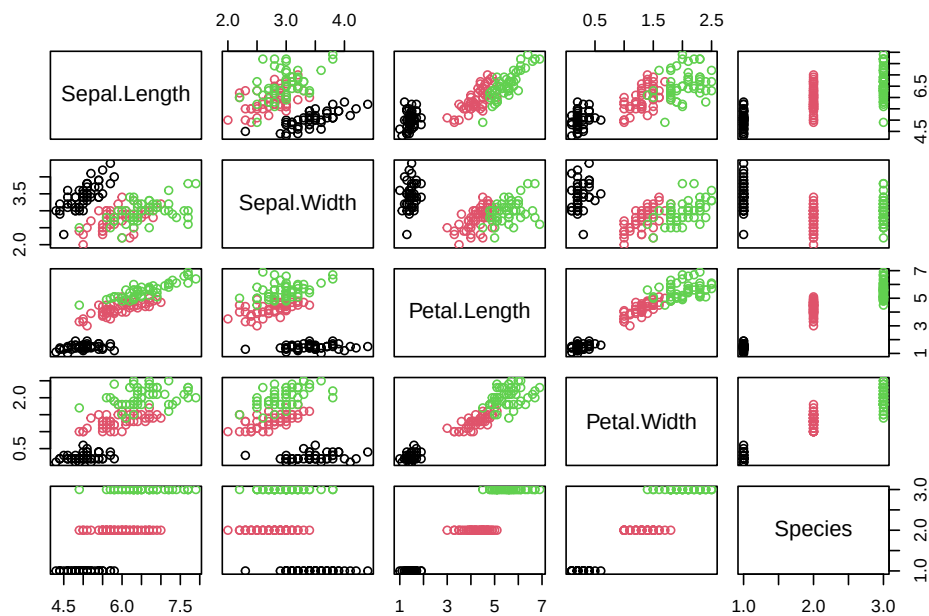
We see that within the groups Versicolor and Virginica there is a much stronger linear relationship than for the group Setosa.

The same can be achieved by using the color aesthetic in the `ggplot` call, then it applies to all geoms.

A.2.4 Scatter Plot Matrix

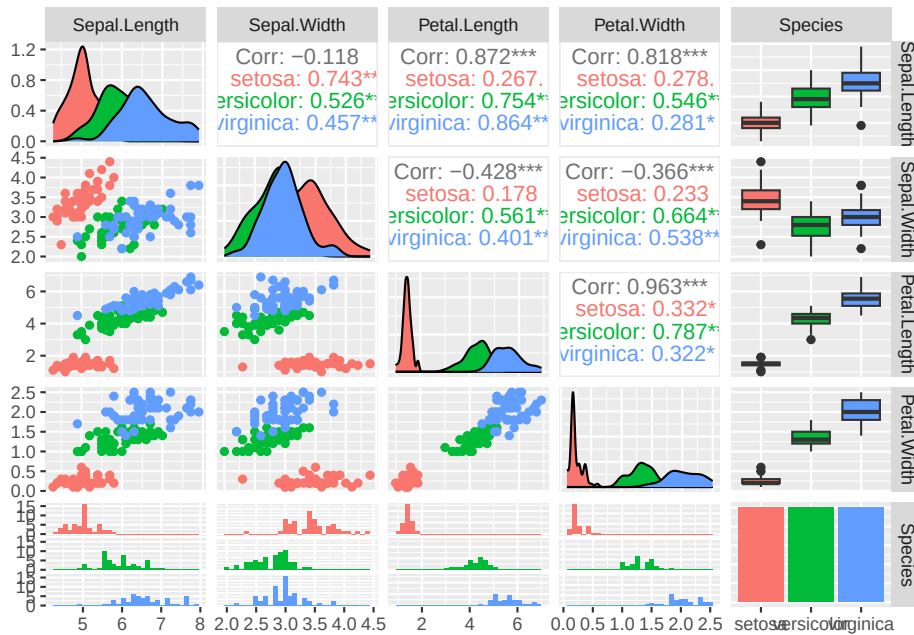
A scatter plot matrix shows the relationship between all pairs of features by arranging panels in a matrix. First, let's look at a regular R-base plot.

```
pairs(iris, col = iris$Species)
```



The package GGally provides a way more sophisticated visualization.

```
library("GGally")
ggpairs(iris, aes(color = Species), progress = FALSE)
## `stat_bin()` using `bins = 30`. Pick better value
## `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value
## `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value
## `binwidth`.
## `stat_bin()` using `bins = 30`. Pick better value
## `binwidth`.
```

Additional plots (histograms, density estimates and box plots) and correlation coefficients are shown in different panels. See the Data Quality section for a description of how to interpret the different panels.

A.2.5 Matrix Visualization

Matrix visualization shows the values in the matrix using a color scale.

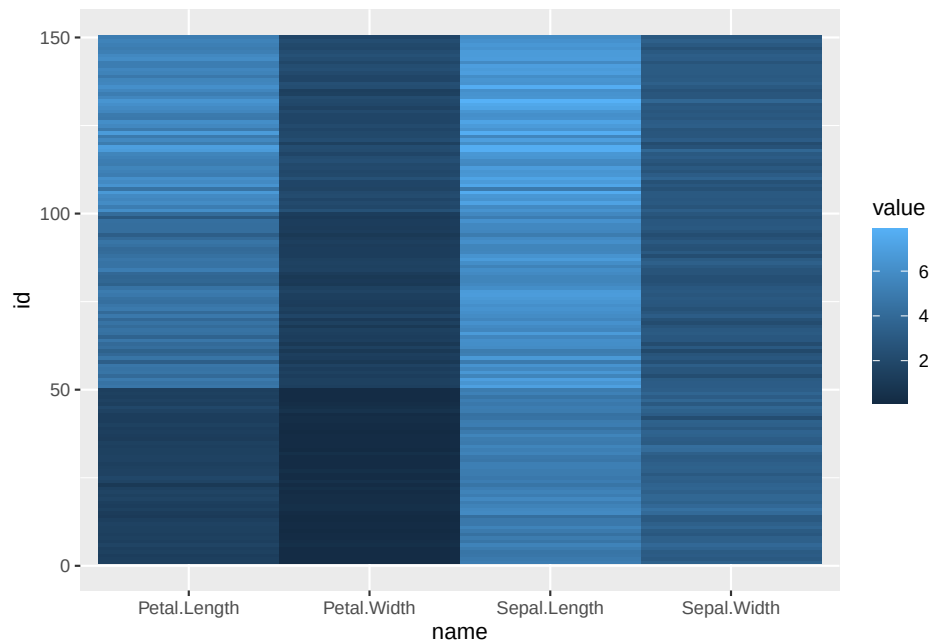
```
iris_matrix <- iris |> select(-Species) |> as.matrix()
```

We need the long format for tidyverse.

```
iris_long <- as_tibble(iris_matrix) |>
  mutate(id = row_number()) |>
  pivot_longer(1:4)

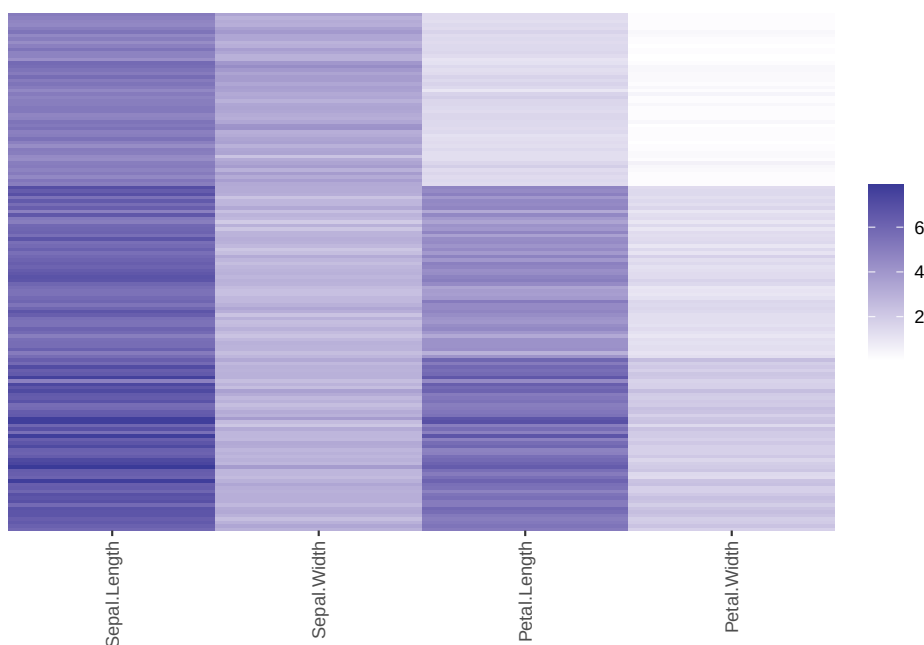
head(iris_long)
## # A tibble: 6 x 3
##   id name      value
##   <int> <chr>    <dbl>
## 1     1 Sepal.Length  5.1
## 2     1 Sepal.Width  3.5
## 3     1 Petal.Length  1.4
## 4     1 Petal.Width   0.2
```

```
## 5      2 Sepal.Length  4.9  
## 6      2 Sepal.Width   3  
ggplot(iris_long, aes(x = name, y = id)) +  
  geom_tile(aes(fill = value))
```



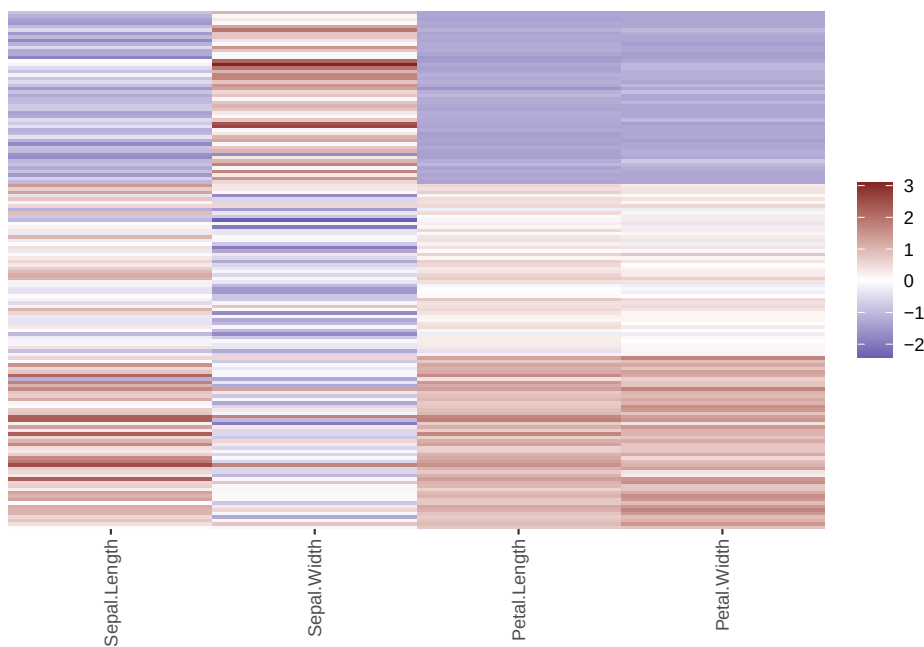
Smaller values are darker. Package **seriation** provides a simpler plotting function.

```
library(seriation)  
ggpimage(iris_matrix, prop = FALSE)
```



We can scale the features to z-scores to make them better comparable.

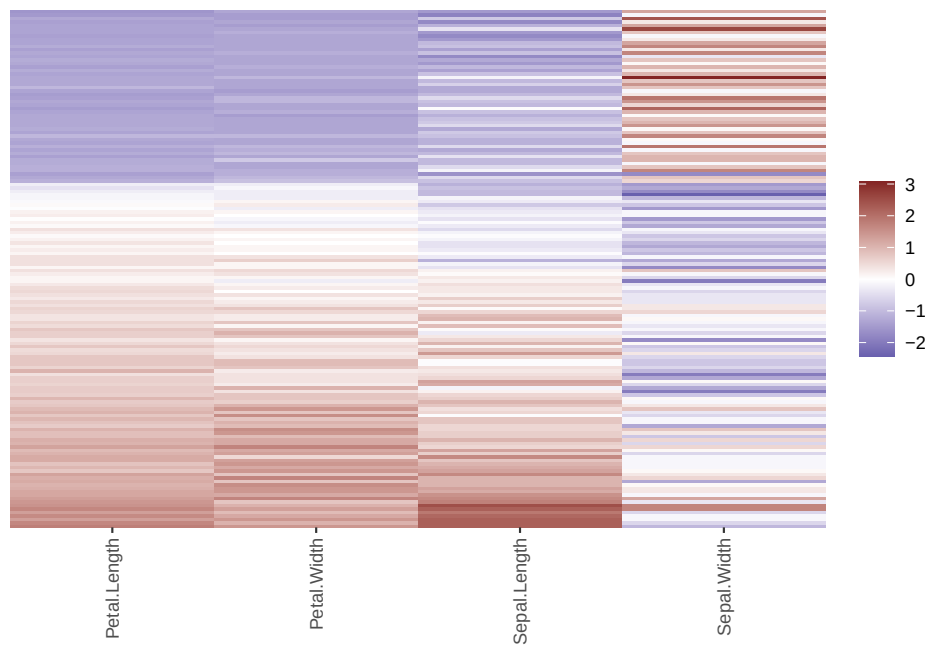
```
iris_scaled <- scale(iris_matrix)
ggpimage(iris_scaled, prop = FALSE)
```



This reveals red and blue blocks. Each row is a flower and the flowers in the Iris dataset are sorted by species. The blue blocks for the top 50 flowers show that these flowers are smaller than average for all but Sepal.Width and the red blocks show that the bottom 50 flowers are larger for most features.

Often, reordering data matrices help with visualization. A reordering technique is called seriation. It reorders rows and columns to place more similar points closer together.

```
ggpimage(iris_scaled, order = seriate(iris_scaled), prop = FALSE)
```



We see that the rows (flowers) are organized from very blue to very red and the features are reordered to move Sepal.Width all the way to the right because it is very different from the other features.

A.2.6 Correlation Matrix

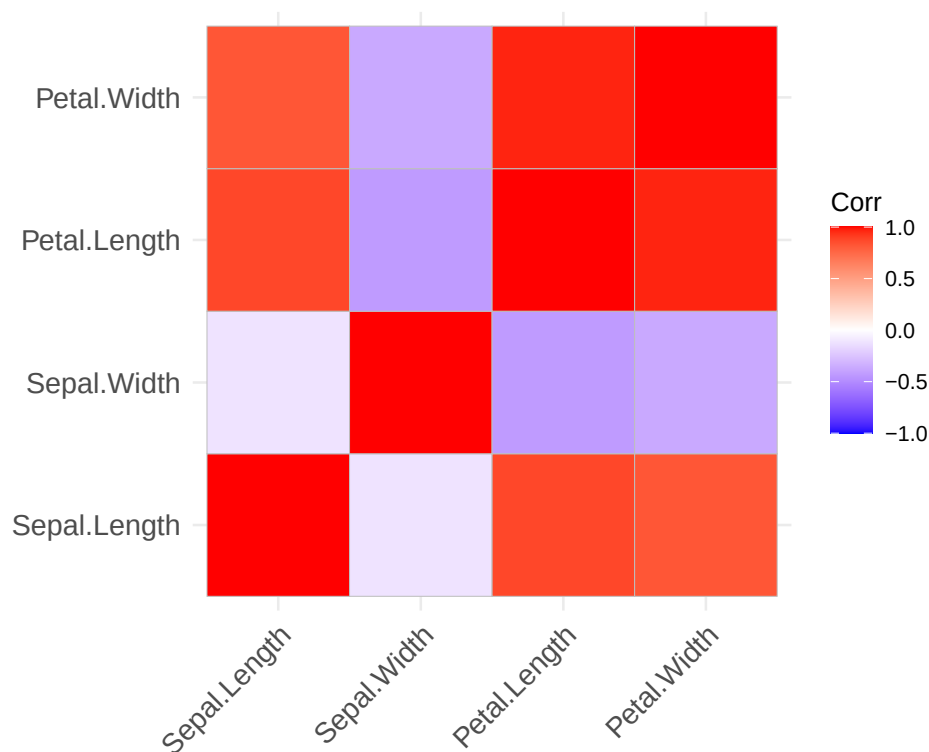
A correlation matrix contains the correlation between features.

```
cm1 <- iris |>
  select(-Species) |>
  as.matrix() |>
  cor()
cm1
```

```
##           Sepal.Length Sepal.Width Petal.Length
## Sepal.Length      1.0000    -0.1176     0.8718
## Sepal.Width      -0.1176     1.0000    -0.4284
## Petal.Length       0.8718    -0.4284     1.0000
## Petal.Width       0.8179    -0.3661     0.9629
##           Petal.Width
## Sepal.Length      0.8179
## Sepal.Width      -0.3661
## Petal.Length       0.9629
## Petal.Width       1.0000
```

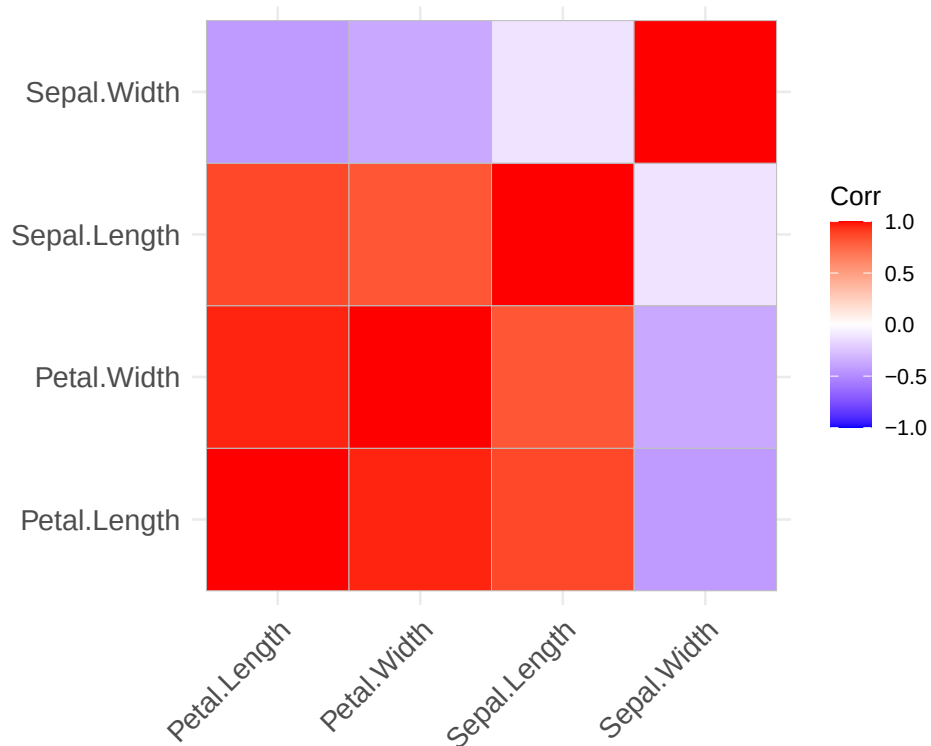
Package `ggcorrplot` provides a ggplot-based visualization for correlation matrices.

```
library(ggcorrplot)
ggcorrplot(cm1)
```



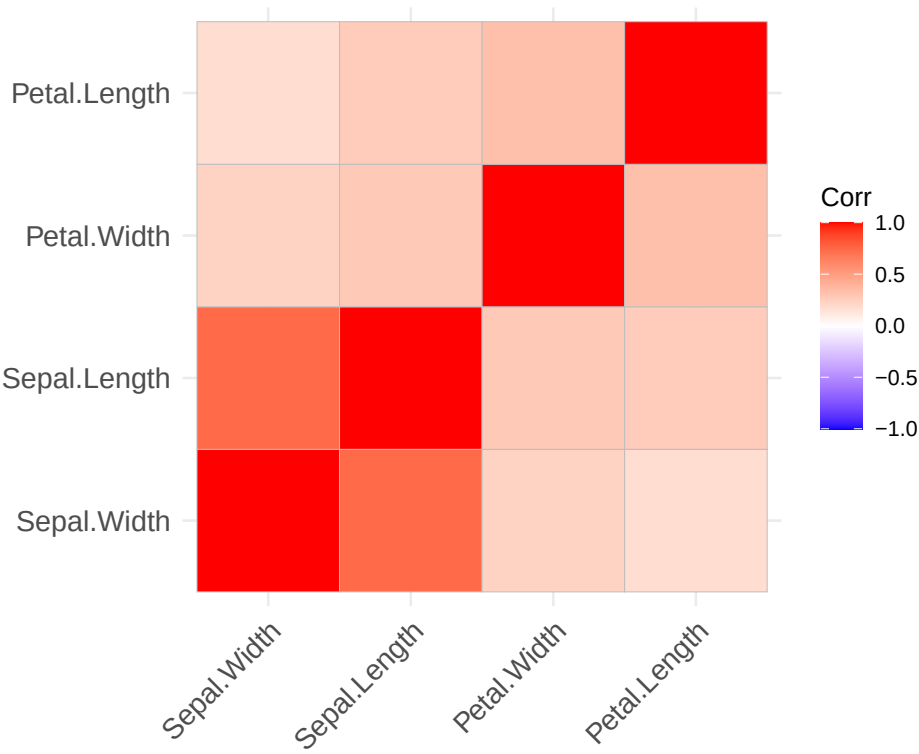
Package `seriation` provides a method to reorder the variables in the correlation matrix so groups of highly correlated variables become easier to see.

```
cm1 |> permute(order = "AOE") |> ggcorrplot()
```

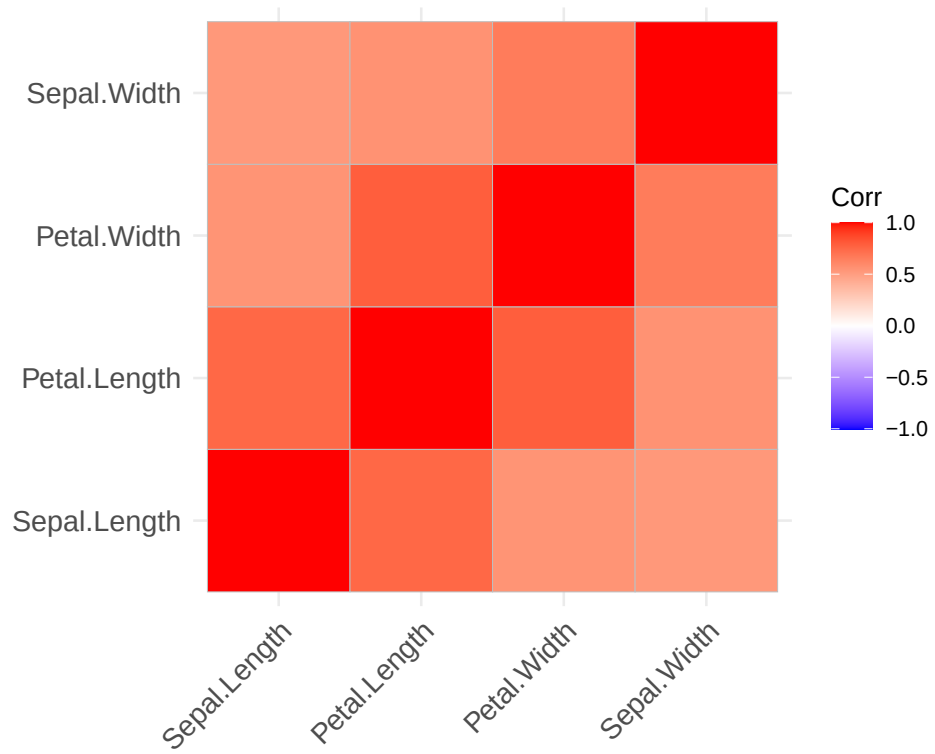


We would assume that as flowers grow, all measurements increase. The visualization shows that sepal length, petal length, and petal width are highly correlated with each other. Interestingly, sepal width has a negative correlation to the other variables which seems to contradict the initial assumption. However, calculating correlation matrices for each group independently show that this is just an artifact resulting from group *Setosa* having a very large sepal width.

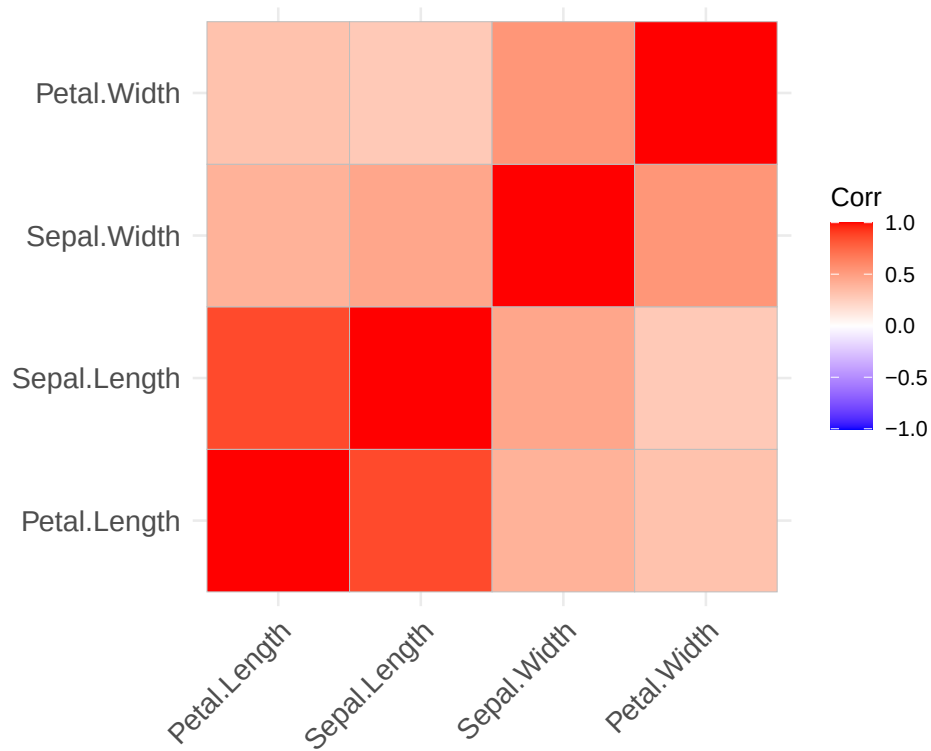
```
iris_by_group <- split(iris, f = iris$Species)
lapply(iris_by_group, FUN = function(x) x |>
  select(-Species) |>
  as.matrix() |>
  cor() |>
  permute(order = "AOE") |>
  ggcorrplot()
)
## $setosa
```



```
##  
## $versicolor
```

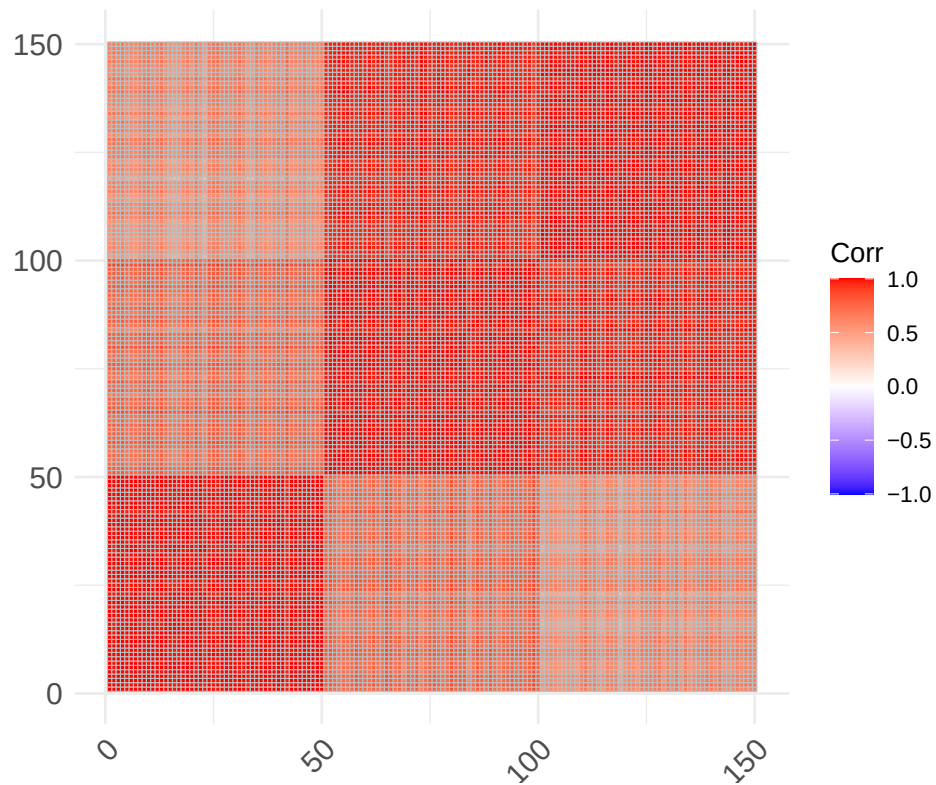


```
##  
## $virginica
```

Correlations can also be calculated between objects by transposing the data matrix.

```
cm2 <- iris |>  
  select(~Species) |>  
  as.matrix() |>  
  t() |>  
  cor()  
  
ggcorrplot(cm2)
```



Object-to-object correlations can be used as a measure of similarity. The dark red blocks indicate different species.

A.2.7 Parallel Coordinates Plot

Parallel coordinate plots can visualize several features in a single plot. Lines connect the values for each object (flower).

```
library(GGally)
ggparcoord(iris, columns = 1:4, groupColumn = 5)
```



The plot can be improved by reordering the variables to place correlated features next to each other.

```
o <- seriate(as.dist(1-cor(iris[,1:4])), method = "BBURCG")
get_order(o)
## Petal.Length Petal.Width Sepal.Length Sepal.Width
##           3           4           1           2
ggparcoord(iris,
            columns = as.integer(get_order(o)),
            groupColumn = 5)
```



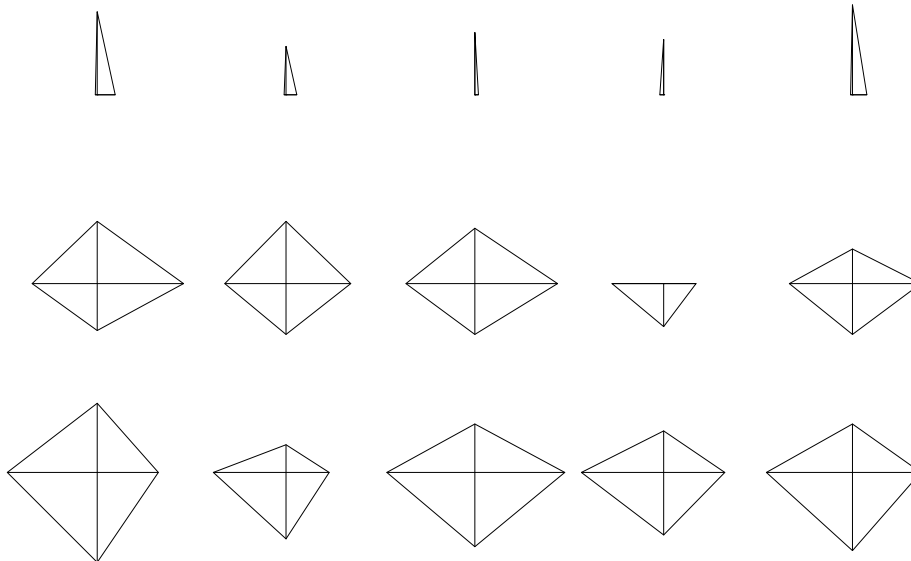
A.2.8 Star Plot

Star plots are a type of radar chart¹⁵ to visualize three or more quantitative variables represented on axes starting from the plot's origin. R-base offers a simple star plot. We plot the first 5 flowers of each species.

```
flowers_5 <- iris[c(1:5, 51:55, 101:105), ]
flowers_5
## # A tibble: 15 x 5
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
##   <dbl>         <dbl>         <dbl>         <dbl> <fct>
## 1         5.1         3.5         1.4         0.2 setosa
## 2         4.9         3         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         3.6         1.4         0.2 setosa
## 6         7         3.2         4.7         1.4 versic~
## 7         6.4         3.2         4.5         1.5 versic~
## 8         6.9         3.1         4.9         1.5 versic~
## 9         5.5         2.3         4         1.3 versic~
## 10        6.5         2.8         4.6         1.5 versic~
## 11        6.3         3.3         6         2.5 virgin~
```

¹⁵https://en.wikipedia.org/wiki/Radar_chart

```
## 12      5.8      2.7      5.1      1.9 virgin~
## 13      7.1      3      5.9      2.1 virgin~
## 14      6.3      2.9      5.6      1.8 virgin~
## 15      6.5      3      5.8      2.2 virgin~
stars(flowers_5[, 1:4], ncol = 5)
```



A.2.9 More Visualizations

A well organized collection of visualizations with code can be found at The R Graph Gallery¹⁶.

A.3 Exercises*

The R package **palmerpenguins** contains measurements for penguin of different species from the Palmer Archipelago, Antarctica. Install the package. It provides a CSV file which can be read in the following way:

```
library("palmerpenguins")
penguins <- read_csv(path_to_file("penguins.csv"))
## Rows: 344 Columns: 8
## -- Column specification -----
## Delimiter: ","
```

¹⁶<https://www.r-graph-gallery.com/>

```
## chr (3): species, island, sex
## dbl (5): bill_length_mm, bill_depth_mm, flipper_length_m...
##
## i Use `spec()` to retrieve the full column specification for this data.
## i Specify the column types or set `show_col_types = FALSE` to quiet this message.
head(penguins)
## # A tibble: 6 x 8
##   species island   bill_length_mm bill_depth_mm
##   <chr>    <chr>         <dbl>         <dbl>
## 1 Adelie  Torgersen         39.1          18.7
## 2 Adelie  Torgersen         39.5          17.4
## 3 Adelie  Torgersen         40.3           18
## 4 Adelie  Torgersen         NA            NA
## 5 Adelie  Torgersen         36.7          19.3
## 6 Adelie  Torgersen         39.3          20.6
## # i 4 more variables: flipper_length_mm <dbl>,
## #   body_mass_g <dbl>, sex <chr>, year <dbl>
```

Create in RStudio a new R Markdown document. Apply the code in the sections of this chapter to the data set to answer the following questions.

1. Group the penguins by species, island or sex. What can you find out?
2. Create histograms and boxplots for each continuous variable. Interpret the distributions.
3. Create scatterplots and a scatterplot matrix. Can you identify correlations?
4. Create a reordered correlation matrix visualization. What does the visualizations show?

Make sure your markdown document contains now a well formatted report. Use the Knit button to create a HTML document.

Appendix B

Regression

Regression is an important statistical method that is covered the freely available Appendix D¹ of the data mining textbook.

In this extra chapter, we introduce the regression problem and multiple linear regression. In addition, alternative models like regression trees and regularized regression are discussed.

Packages Used in this Chapter

```
pkgs <- c('lars', 'rpart', 'rpart.plot', 'nnet')

pkgs_install <- pkgs[!(pkgs %in% installed.packages()[,"Package"])]
if(length(pkgs_install)) install.packages(pkgs_install)
```

The packages used for this chapter are:

- *lars* (Hastie and Efron 2022)
- *nnet* (Ripley 2025)
- *rpart* (Therneau and Atkinson 2025)
- *rpart.plot* (Milborrow 2025)

B.1 Introduction

Recall that classification predicts one of a small set of discrete (i.e., nominal) labels (e.g., yes or no, small, medium or large). Regression is also a supervised

¹https://mhahsler.github.io/Introduction_to_Data_Mining_R_Examples/DM_chapters/appendices_2ed.pdf

learning problem, but the goal is to predict the value of a continuous value given a set of predictors. We start with the popular linear regression and will later discuss alternative regression methods.

Linear regression models the value of a dependent variable y (also called the response) as a linear function of independent variables $X_1, X_2, \dots, X_p$ (also called the regressors, predictors, exogenous variables, explanatory variables, or covariates). If we use more than one explanatory variable, then we often call this multiple or multivariate linear regression. The linear regression model is:

$$\hat{y}_i = f(\mathbf{x}_i) = \beta_0 + \beta_1 x_{i1} + \dots + \beta_p x_{ip} + \epsilon_i = \beta_0 + \sum_{j=1}^p (\beta_j x_{ij}) + \epsilon_i,$$

where β_0 is the intercept, β is a $p + 1$ -dimensional parameter vector learned from the data, and ϵ is the error term (called the residuals). This is often also written in vector notation as

$$\hat{\mathbf{y}} = \mathbf{X}\beta + \epsilon,$$

where $\hat{\mathbf{y}}$ and β are vectors and $\mathbf{X}$ is the matrix with the covariates (called the design matrix).

The error that is often minimized in regression problems is the squared error defined as:

$$SE = \sum_i (y_i - f(\mathbf{x}_i))^2$$

The parameter vector is found by minimizing the squared error the training data.

Linear regression makes several assumptions that should be checked:

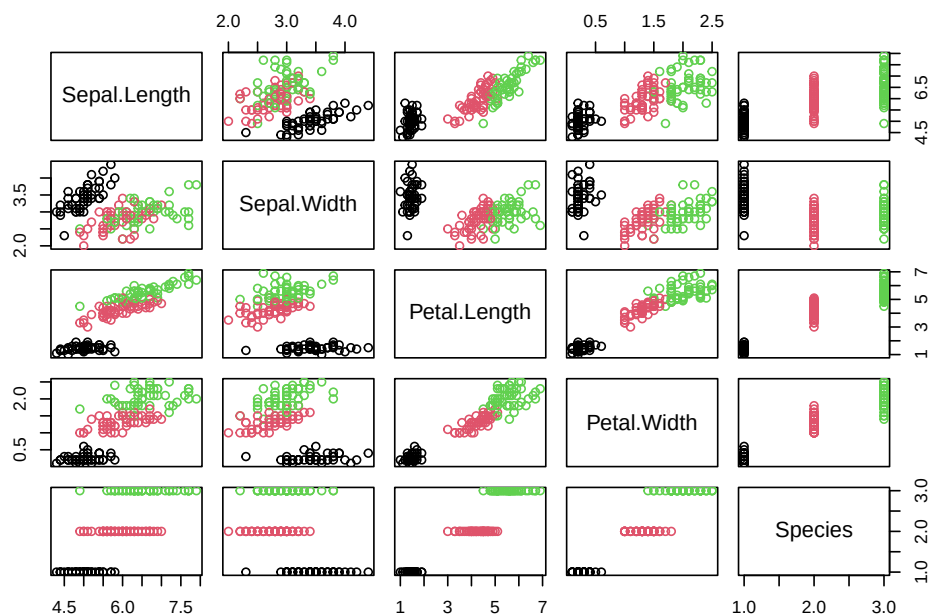
- *Linearity*: There is a linear relationship between dependent and independent variables.
- *Homoscedasticity*: The variance of the error (ϵ) does not change (increase) with the predicted value.
- *Independence of errors*: Errors between observations are uncorrelated.
- *No multicollinearity of predictors*: Predictors cannot be perfectly correlated or the parameter vector cannot be identified. *Note* that highly correlated predictors lead to unstable results and should be avoided using, e.g., variable selection.

B.2 A First Linear Regression Model

We will use the Iris dataset and try to predict the `Petal.Width` using the other variables. We first load and shuffle data since the flowers in the dataset are in order by species.


```
data(iris)
set.seed(2000) # make the sampling reproducible

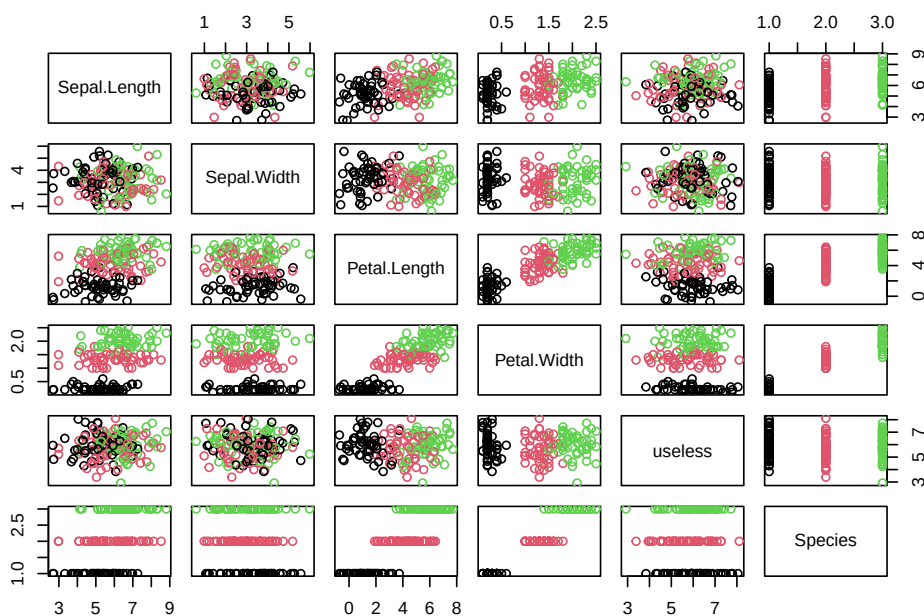
x <- iris[sample(1:nrow(iris)),]
plot(x, col=x$Species)
```



The Iris data is very clean, so we make the data a little messy by adding a random error to each variable and introduce a useless, completely random feature.

```
x[,1] <- x[,1] + rnorm(nrow(x))
x[,2] <- x[,2] + rnorm(nrow(x))
x[,3] <- x[,3] + rnorm(nrow(x))
x <- cbind(x[,1:4],
           useless = mean(x[,1]) + rnorm(nrow(x)),
           Species = x[,5])

plot(x, col=x$Species)
```



```
summary(x)
```

```
##   Sepal.Length   Sepal.Width   Petal.Length
##   Min.   :2.68    Min.   :0.611   Min.   : -0.713
##   1st Qu.:5.05    1st Qu.:2.306   1st Qu.: 1.876
##   Median :5.92    Median :3.171   Median : 4.102
##   Mean   :5.85    Mean   :3.128   Mean   : 3.781
##   3rd Qu.:6.70    3rd Qu.:3.945   3rd Qu.: 5.546
##   Max.   :8.81    Max.   :5.975   Max.   : 7.708
##   Petal.Width   useless        Species
##   Min.   :0.1    Min.   :2.92   setosa    :50
##   1st Qu.:0.3    1st Qu.:5.23   versicolor:50
##   Median :1.3    Median :5.91   virginica :50
##   Mean   :1.2    Mean   :5.88
##   3rd Qu.:1.8    3rd Qu.:6.56
##   Max.   :2.5    Max.   :8.11
```

```
head(x)
```

```
##      Sepal.Length Sepal.Width Petal.Length Petal.Width
## 85          2.980      1.464      5.227      1.5
## 104         5.096      3.044      5.187      1.8
## 30          4.361      2.832      1.861      0.2
## 53          8.125      2.406      5.526      1.5
## 143         6.372      1.565      6.147      1.9
## 142         6.526      3.697      5.708      2.3
##      useless Species
## 85      5.712 versicolor
## 104     6.569  virginica
```

```
## 30    4.299    setosa
## 53    6.124 versicolor
## 143   6.553 virginica
## 142   5.222 virginica
```

We split the data into training and test data. Since the data is shuffled, we effectively perform holdout sampling. This is often not done in statistical applications, but we use the machine learning approach here.

```
train <- x[1:100,]
test  <- x[101:150,]
```

Linear regression is done in R using the `lm()` (linear model) function which is part of the R core package `stats`. Like other modeling functions in R, `lm()` uses a formula interface. Here we create a formula to predict `Petal.Width` by all other variables other than `Species`.

```
model1 <- lm(Petal.Width ~ Sepal.Length
              + Sepal.Width + Petal.Length + useless,
              data = train)

model1
##
## Call:
## lm(formula = Petal.Width ~ Sepal.Length + Sepal.Width + Petal.Length +
##      useless, data = train)
##
## Coefficients:
## (Intercept) Sepal.Length Sepal.Width Petal.Length
##      -0.20589      0.01957      0.03406      0.30814
##      useless
##      0.00392
```

The result is a model with the fitted β coefficients. More information can be displayed using the summary function.

```
summary(model1)
##
## Call:
## lm(formula = Petal.Width ~ Sepal.Length + Sepal.Width + Petal.Length +
##      useless, data = train)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.0495 -0.2033  0.0074  0.2038  0.8564
```

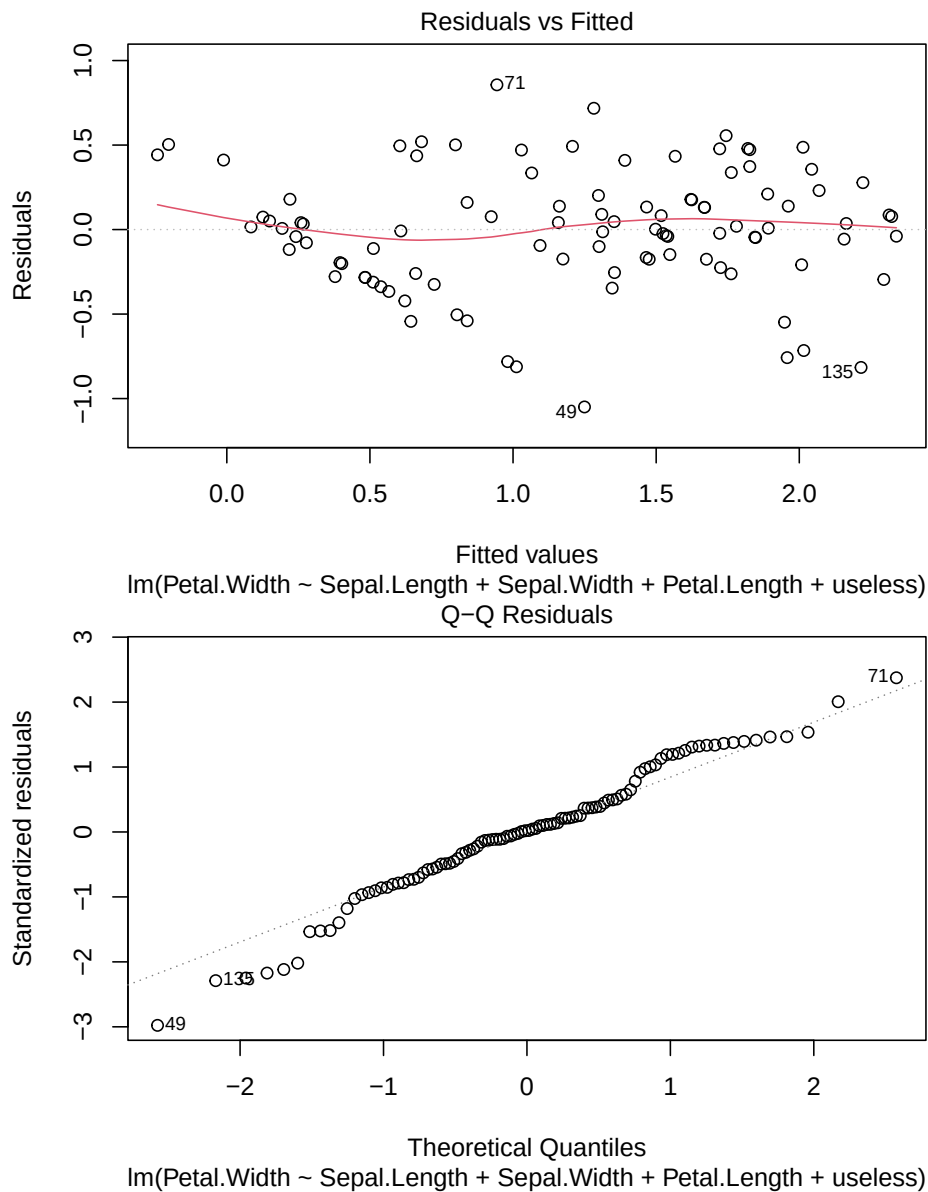
```
##
## Coefficients:
##           Estimate Std. Error t value Pr(>|t|)
## (Intercept) -0.20589    0.28723   -0.72    0.48
## Sepal.Length  0.01957    0.03265    0.60    0.55
## Sepal.Width   0.03406    0.03549    0.96    0.34
## Petal.Length  0.30814    0.01819   16.94 <2e-16 ***
## useless       0.00392    0.03558    0.11    0.91
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.366 on 95 degrees of freedom
## Multiple R-squared:  0.778, Adjusted R-squared:  0.769
## F-statistic: 83.4 on 4 and 95 DF, p-value: <2e-16
```

The summary shows:

- Which coefficients are significantly different from 0. Here only `Petal.Length` is significant and the coefficient for `useless` is very close to 0. Look at the scatter plot matrix above and see why this is the case.
- R-squared (coefficient of determination): Proportion (in the range $[0, 1]$) of the variability of the dependent variable explained by the model. It is better to look at the adjusted R-square (adjusted for number of dependent variables). Typically, an R-squared of greater than 0.7 is considered good, but this is just a rule of thumb and we should rather use a test set for evaluation.

Plotting the model produces diagnostic plots (see `plot.lm()`). For example, to check that the error term has a mean of 0 and is homoscedastic, the residual vs. predicted value scatter plot should have a red line that stays close to 0 and not look like a funnel where they are increasing when the fitted values increase. To check if the residuals are approximately normally distributed, the Q-Q plot should be close to the straight diagonal line.

```
plot(model1, which = 1:2)
```



In this case, the two plots look fine.

B.3 Comparing Nested Models

Here we perform model selection to compare several linear models. Nested means that all models use a subset of the same set of features. We create a

simpler model by removing the feature useless from the model above.

```
model2 <- lm(Petal.Width ~ Sepal.Length +
             Sepal.Width + Petal.Length,
             data = train)
summary(model2)
##
## Call:
## lm(formula = Petal.Width ~ Sepal.Length + Sepal.Width + Petal.Length,
##     data = train)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.0440 -0.2024  0.0099  0.1998  0.8513
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -0.1842     0.2076   -0.89    0.38
## Sepal.Length   0.0199     0.0323    0.62    0.54
## Sepal.Width    0.0339     0.0353    0.96    0.34
## Petal.Length   0.3080     0.0180   17.07 <2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.365 on 96 degrees of freedom
## Multiple R-squared:  0.778, Adjusted R-squared:  0.771
## F-statistic: 112 on 3 and 96 DF, p-value: <2e-16
```

We can remove the intercept by adding -1 to the formula.

```
model3 <- lm(Petal.Width ~ Sepal.Length +
             Sepal.Width + Petal.Length - 1,
             data = train)
summary(model3)
##
## Call:
## lm(formula = Petal.Width ~ Sepal.Length + Sepal.Width + Petal.Length -
##     1, data = train)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.0310 -0.1961 -0.0051  0.2264  0.8246
##
## Coefficients:
```

```
##           Estimate Std. Error t value Pr(>|t|)
## Sepal.Length -0.00168    0.02122  -0.08    0.94
## Sepal.Width   0.01859    0.03073   0.61    0.55
## Petal.Length  0.30666    0.01796  17.07 <2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.364 on 97 degrees of freedom
## Multiple R-squared:  0.938, Adjusted R-squared:  0.936
## F-statistic: 486 on 3 and 97 DF, p-value: <2e-16
```

Here is a very simple model.

```
model4 <- lm(Petal.Width ~ Petal.Length -1,
             data = train)
summary(model4)
##
## Call:
## lm(formula = Petal.Width ~ Petal.Length - 1, data = train)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.9774 -0.1957  0.0078  0.2536  0.8455
##
## Coefficients:
##           Estimate Std. Error t value Pr(>|t|)
## Petal.Length  0.31586    0.00822   38.4 <2e-16 ***
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.362 on 99 degrees of freedom
## Multiple R-squared:  0.937, Adjusted R-squared:  0.936
## F-statistic: 1.47e+03 on 1 and 99 DF, p-value: <2e-16
```

We need a statistical test to compare all these nested models. The appropriate test is called ANOVA² (analysis of variance) which is a generalization of the t-test to check if all the treatments (i.e., models) have the same effect. **Important note:** This only works for *nested models*. Models are nested only if one model contains all the predictors of the other model.

²https://en.wikipedia.org/wiki/Analysis_of_variance

```
anova(model1, model2, model3, model4)
## Analysis of Variance Table
##
## Model 1: Petal.Width ~ Sepal.Length + Sepal.Width + Petal.Length + useless
## Model 2: Petal.Width ~ Sepal.Length + Sepal.Width + Petal.Length
## Model 3: Petal.Width ~ Sepal.Length + Sepal.Width + Petal.Length - 1
## Model 4: Petal.Width ~ Petal.Length - 1
##   Res.Df  RSS Df Sum of Sq   F Pr(>F)
## 1      95 12.8
## 2      96 12.8 -1   -0.0016 0.01   0.91
## 3      97 12.9 -1   -0.1046 0.78   0.38
## 4      99 13.0 -2   -0.1010 0.38   0.69
```

Models 1 is not significantly better than model 2. Model 2 is not significantly better than model 3. Model 3 is not significantly better than model 4! Use model 4, the simplest model. See `anova.lm()` for the manual page for ANOVA for linear models.

B.4 Stepwise Variable Selection

Stepwise variable section performs backward (or forward) model selection for linear models and uses the smallest AIC (Akaike information criterion³) to decide what variable to remove and when to stop.

```
s1 <- stepAIC(lm(Petal.Width ~ . -Species, data = train))
## Start:  AIC=-195.9
## Petal.Width ~ (Sepal.Length + Sepal.Width + Petal.Length + useless +
##   Species) - Species
##
##           Df Sum of Sq  RSS   AIC
## - useless      1      0.0 12.8 -197.9
## - Sepal.Length  1      0.0 12.8 -197.5
## - Sepal.Width   1      0.1 12.9 -196.9
## <none>              12.8 -195.9
## - Petal.Length   1     38.6 51.3 -58.7
##
## Step:  AIC=-197.9
## Petal.Width ~ Sepal.Length + Sepal.Width + Petal.Length
##
##           Df Sum of Sq  RSS   AIC
## - Sepal.Length  1      0.1 12.8 -199.5
## - Sepal.Width   1      0.1 12.9 -198.9
```

³https://en.wikipedia.org/wiki/Akaike_information_criterion


```
## <none>                12.8 -197.9
## - Petal.Length  1      38.7 51.5  -60.4
##
## Step:  AIC=-199.5
## Petal.Width ~ Sepal.Width + Petal.Length
##
##              Df Sum of Sq  RSS    AIC
## - Sepal.Width  1         0.1 12.9 -200.4
## <none>                12.8 -199.5
## - Petal.Length  1      44.7 57.5  -51.3
##
## Step:  AIC=-200.4
## Petal.Width ~ Petal.Length
##
##              Df Sum of Sq  RSS    AIC
## <none>                12.9 -200.4
## - Petal.Length  1      44.6 57.6  -53.2
summary(s1)
##
## Call:
## lm(formula = Petal.Width ~ Petal.Length, data = train)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.9848 -0.1873  0.0048  0.2466  0.8343
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   0.0280     0.0743   0.38    0.71
## Petal.Length   0.3103     0.0169  18.38 <2e-16 ***
## ---
## Signif. codes:
##  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.364 on 98 degrees of freedom
## Multiple R-squared:  0.775, Adjusted R-squared:  0.773
## F-statistic: 338 on 1 and 98 DF, p-value: <2e-16
```

Each table represents one step and shows the AIC when each variable is removed and the one with the smallest AIC (in the first table `useless`) is removed. It stops with a small model that only uses `Petal.Length`.

B.5 Modeling with Interaction Terms

Linear regression models the effect of each predictor separately using a β coefficient. What if two variables are only important together? This is called an interaction between predictors and is modeled using interaction terms. In R's `formula()` we can use `:` and `*` to specify interactions. For linear regression, an interaction means that a new predictor is created by multiplying the two original predictors.

We can create a model with an interaction terms between `Sepal.Length`, `Sepal.Width` and `Petal.Length`.

```
model5 <- step(lm(Petal.Width ~ Sepal.Length *
                  Sepal.Width * Petal.Length,
                  data = train))
## Start: AIC=-196.4
## Petal.Width ~ Sepal.Length * Sepal.Width * Petal.Length
##
##                                     Df Sum of Sq  RSS
## <none>                                     11.9
## - Sepal.Length:Sepal.Width:Petal.Length  1    0.265 12.2
##                                     AIC
## <none>                                     -196
## - Sepal.Length:Sepal.Width:Petal.Length -196
summary(model5)
##
## Call:
## lm(formula = Petal.Width ~ Sepal.Length * Sepal.Width * Petal.Length,
##     data = train)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -1.1064 -0.1882  0.0238  0.1767  0.8577
##
## Coefficients:
##                                     Estimate Std. Error
## (Intercept)                       -2.4207     1.3357
## Sepal.Length                       0.4484     0.2313
## Sepal.Width                        0.5983     0.3845
## Petal.Length                       0.7275     0.2863
## Sepal.Length:Sepal.Width           -0.1115     0.0670
## Sepal.Length:Petal.Length          -0.0833     0.0477
## Sepal.Width:Petal.Length           -0.0941     0.0850
## Sepal.Length:Sepal.Width:Petal.Length  0.0201     0.0141
##                                     t value Pr(>|t|)
## (Intercept)                       -1.81    0.073 .
```

```
## Sepal.Length          1.94    0.056 .
## Sepal.Width           1.56    0.123
## Petal.Length          2.54    0.013 *
## Sepal.Length:Sepal.Width -1.66    0.100 .
## Sepal.Length:Petal.Length -1.75    0.084 .
## Sepal.Width:Petal.Length -1.11    0.272
## Sepal.Length:Sepal.Width:Petal.Length 1.43    0.157
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.36 on 92 degrees of freedom
## Multiple R-squared:  0.792, Adjusted R-squared:  0.777
## F-statistic: 50.2 on 7 and 92 DF, p-value: <2e-16
anova(model5, model4)
## Analysis of Variance Table
##
## Model 1: Petal.Width ~ Sepal.Length * Sepal.Width * Petal.Length
## Model 2: Petal.Width ~ Petal.Length - 1
##   Res.Df  RSS Df Sum of Sq   F Pr(>F)
## 1      92 11.9
## 2      99 13.0 -7      -1.01 1.11  0.36
```

Some interaction terms are significant in the new model, but ANOVA shows that model 5 is not significantly better than model 4

B.6 Prediction

We preform here a prediction for the held out test set.

```
test[1:5,]
##      Sepal.Length Sepal.Width Petal.Length Petal.Width
## 128      8.017      1.541      3.515      1.8
## 92       5.268      4.064      6.064      1.4
## 50       5.461      4.161      1.117      0.2
## 134      6.055      2.951      4.599      1.5
## 8        4.900      5.096      1.086      0.2
##      useless      Species
## 128    6.110 virginica
## 92     4.938 versicolor
## 50     6.373  setosa
## 134    5.595 virginica
## 8      7.270  setosa
```

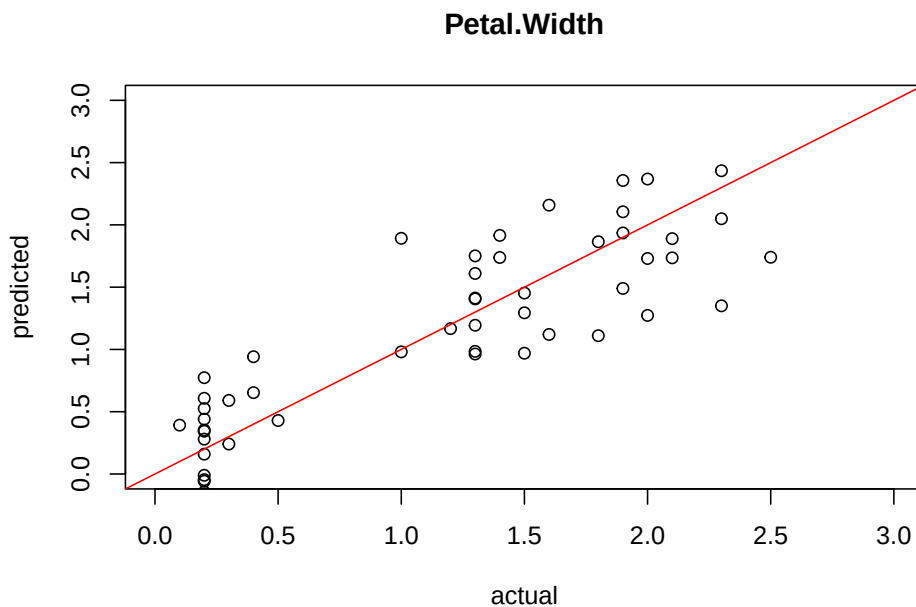
```
test[1:5,]$Petal.Width
## [1] 1.8 1.4 0.2 1.5 0.2
predict(model4, test[1:5,])
##      128      92      50     134      8
## 1.1104 1.9155 0.3529 1.4526 0.3429
```

The most used error measure for regression is the RMSE⁴ root-mean-square error.

```
RMSE <- function(predicted, true) mean((predicted-true)^2)^.5
RMSE(predict(model4, test), test$Petal.Width)
## [1] 0.3874
```

We can also visualize the quality of the prediction using a simple scatter plot of predicted vs. actual values.

```
plot(test[, "Petal.Width"], predict(model4, test),
     xlim=c(0,3), ylim=c(0,3),
     xlab = "actual", ylab = "predicted",
     main = "Petal.Width")
abline(0,1, col="red")
```



⁴https://en.wikipedia.org/wiki/Root_mean_square_deviation

```
cor(test[, "Petal.Width"], predict(model4, test))
## [1] 0.8636
```

Perfect predictions would be on the red line, the farther they are away, the larger the error.

B.7 Using Nominal Variables

Dummy variables⁵ also called one-hot encoding in machine learning is used for factors (i.e., levels are translated into individual 0-1 variable). The first level of factors is automatically used as the reference and the other levels are presented as 0-1 dummy variables called contrasts.

```
levels(train$Species)
## [1] "setosa" "versicolor" "virginica"
```

`model.matrix()` is used internally to create dummy variables when the design matrix for the regression is created..

```
head(model.matrix(Petal.Width ~ ., data=train))
##      (Intercept) Sepal.Length Sepal.Width Petal.Length
## 85              1         2.980         1.464         5.227
## 104             1         5.096         3.044         5.187
## 30              1         4.361         2.832         1.861
## 53              1         8.125         2.406         5.526
## 143             1         6.372         1.565         6.147
## 142             1         6.526         3.697         5.708
##      useless Speciesversicolor Speciesvirginica
## 85      5.712              1              0
## 104      6.569              0              1
## 30      4.299              0              0
## 53      6.124              1              0
## 143      6.553              0              1
## 142      5.222              0              1
```

Note that there is no dummy variable for species *Setosa*, because it is used as the reference (when the other two dummy variables are 0). It is often useful to set the reference level. A simple way is to use the function `relevel()` to change which factor is listed first.

Let us perform model selection using AIC on the training data and then evaluate the final model on the held out test set to estimate the generalization error.

⁵[https://en.wikipedia.org/wiki/Dummy_variable_\(statistics\)](https://en.wikipedia.org/wiki/Dummy_variable_(statistics))

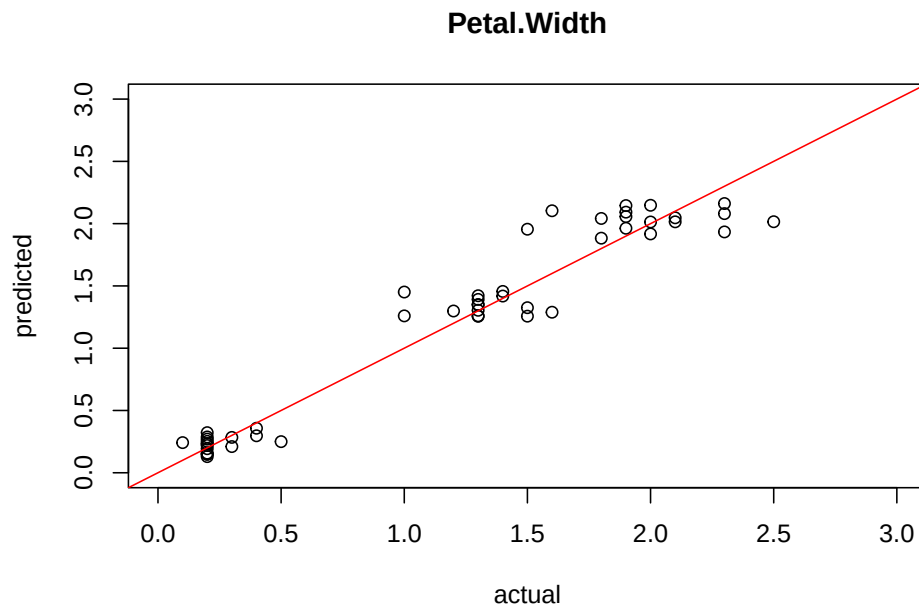
```

model6 <- step(lm(Petal.Width ~ ., data=train))
## Start: AIC=-308.4
## Petal.Width ~ Sepal.Length + Sepal.Width + Petal.Length + useless +
## Species
##
##           Df Sum of Sq  RSS  AIC
## - Sepal.Length  1      0.01  3.99 -310
## - Sepal.Width  1      0.01  3.99 -310
## - useless      1      0.02  4.00 -310
## <none>                      3.98 -308
## - Petal.Length  1      0.47  4.45 -299
## - Species      2      8.78 12.76 -196
##
## Step: AIC=-310.2
## Petal.Width ~ Sepal.Width + Petal.Length + useless + Species
##
##           Df Sum of Sq  RSS  AIC
## - Sepal.Width  1      0.01  4.00 -312
## - useless      1      0.02  4.00 -312
## <none>                      3.99 -310
## - Petal.Length  1      0.49  4.48 -300
## - Species      2      8.82 12.81 -198
##
## Step: AIC=-311.9
## Petal.Width ~ Petal.Length + useless + Species
##
##           Df Sum of Sq  RSS  AIC
## - useless      1      0.02  4.02 -313
## <none>                      4.00 -312
## - Petal.Length  1      0.48  4.48 -302
## - Species      2      8.95 12.95 -198
##
## Step: AIC=-313.4
## Petal.Width ~ Petal.Length + Species
##
##           Df Sum of Sq  RSS  AIC
## <none>                      4.02 -313
## - Petal.Length  1      0.50  4.52 -304
## - Species      2      8.93 12.95 -200
model6
##
## Call:
## lm(formula = Petal.Width ~ Petal.Length + Species, data = train)
##
## Coefficients:

```

```
##      (Intercept)      Petal.Length Speciesversicolor
##           0.1597           0.0664           0.8938
## Speciesvirginica
##           1.4903
summary(model6)
##
## Call:
## lm(formula = Petal.Width ~ Petal.Length + Species, data = train)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -0.7208 -0.1437  0.0005   0.1254   0.5460
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)      0.1597     0.0441   3.62  0.00047 ***
## Petal.Length      0.0664     0.0192   3.45  0.00084 ***
## Speciesversicolor  0.8938     0.0746  11.98 < 2e-16 ***
## Speciesvirginica  1.4903     0.1020  14.61 < 2e-16 ***
## ---
## Signif. codes:
##  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.205 on 96 degrees of freedom
## Multiple R-squared:  0.93,    Adjusted R-squared:  0.928
## F-statistic: 427 on 3 and 96 DF,  p-value: <2e-16
```

```
RMSE(predict(model6, test), test$Petal.Width)
## [1] 0.1885
plot(test[, "Petal.Width"], predict(model6, test),
     xlim=c(0,3), ylim=c(0,3),
     xlab = "actual", ylab = "predicted",
     main = "Petal.Width")
abline(0,1, col="red")
```



```
cor(test[, "Petal.Width"], predict(model6, test))
## [1] 0.9696
```

We see that the `Species` variable provides information to improve the regression model.

B.8 Alternative Regression Models

B.8.1 Regression Trees

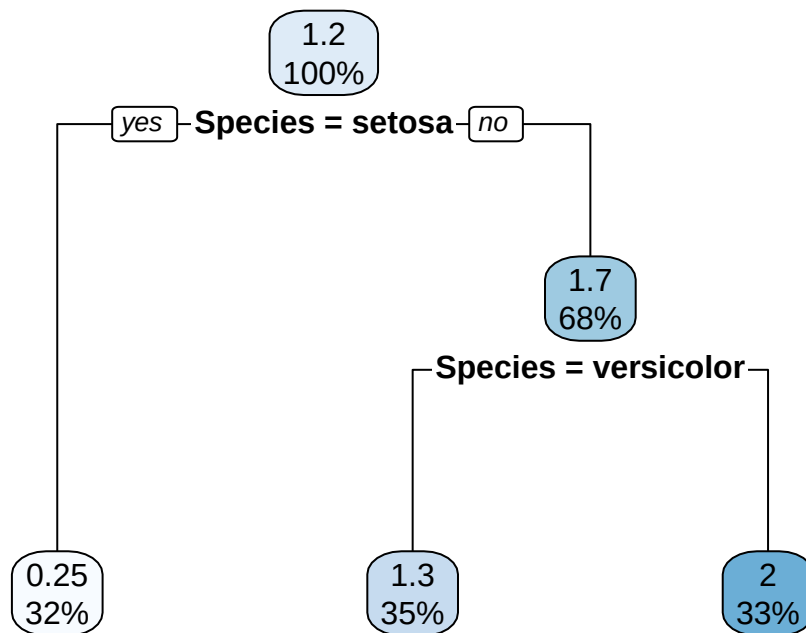
Many models used for classification can also perform regression. For example CART implemented in `rpart` performs regression by estimating a value for each leaf node. Regression is always performed by `rpart()` when the response variable is not a `factor()`.

```
library(rpart)
library(rpart.plot)

model7 <- rpart(Petal.Width ~ ., data = train,
  control = rpart.control(cp = 0.01))
model7
## n= 100
##
## node), split, n, deviance, yval
```



```
##      * denotes terminal node
##
## 1) root 100 57.5700 1.2190
## 2) Species=setosa 32 0.3797 0.2469 *
## 3) Species=versicolor,virginica 68 12.7200 1.6760
## 6) Species=versicolor 35 1.5350 1.3310 *
## 7) Species=virginica 33 2.6010 2.0420 *
rpart.plot(model7)
```



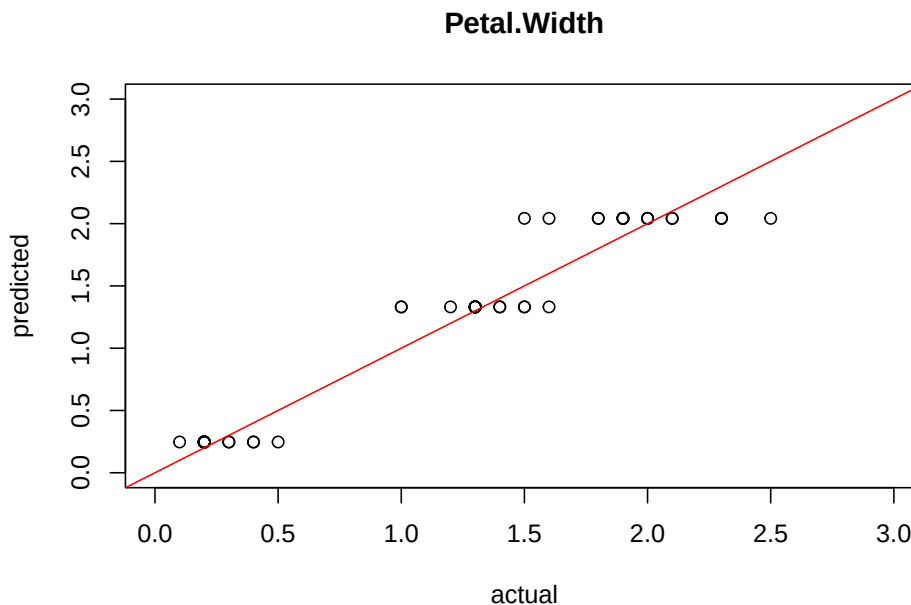
The regression tree shows the predicted value in as the top number in the node. Also, remember that tree-based models automatically variable selection by choosing the splits.

Let's evaluate the regression tree by calculating the RMSE.

```
pred <- predict(model7, test)
RMSE(pred, test$Petal.Width)
## [1] 0.182
```

And visualize the quality.

```
plot(test[, "Petal.Width"], pred,
     xlim = c(0,3), ylim = c(0,3),
     xlab = "actual", ylab = "predicted",
     main = "Petal.Width")
abline(0,1, col = "red")
```



```
cor(test[, "Petal.Width"], pred)
## [1] 0.9717
```

The plot and the correlation coefficient indicate that the model is very good. In the plot we see an important property of this method which is that it predicts exactly the same value when the data falls in the same leaf node.

B.8.2 Regularized Regression

LASSO⁶ (least absolute shrinkage and selection operator) uses L1 regularization to reduce to perform automatic variable selection. The regularization adds a penalty for the L1 norm of the parameter vector β (i.e., the sum of all parameter values) to the optimization. Increasing the weight of this penalty term (the weight is called λ) results in more and more weights being pushed to 0 which effectively reduces the number of parameters used in the regression. An implementation called the elastic net⁷ is available as the function `lars()` in package `lars`.

```
library(lars)
## Loaded lars 1.3
```

We create a design matrix (with dummy variables and interaction terms). `lm()`

⁶[https://en.wikipedia.org/wiki/Lasso_\(statistics\)](https://en.wikipedia.org/wiki/Lasso_(statistics))

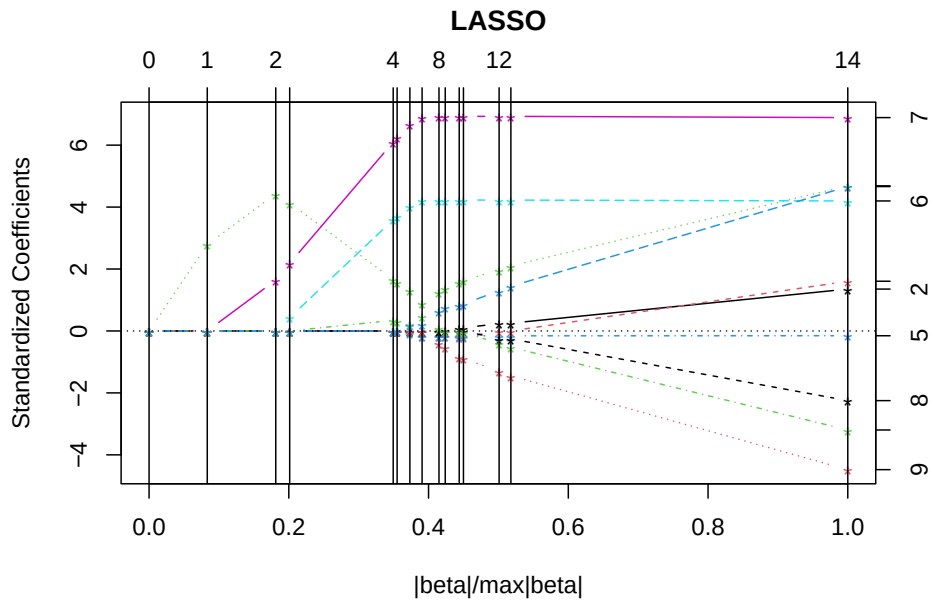
⁷https://en.wikipedia.org/wiki/Elastic_net_regularization

did this automatically for us, but for this `lars()` implementation we have to do it manually.

```
x <- model.matrix(~ . + Sepal.Length*Sepal.Width*Petal.Length ,
  data = train[, -4])
head(x)
##      (Intercept) Sepal.Length Sepal.Width Petal.Length
## 85             1         2.980         1.464         5.227
## 104            1         5.096         3.044         5.187
## 30             1         4.361         2.832         1.861
## 53             1         8.125         2.406         5.526
## 143            1         6.372         1.565         6.147
## 142            1         6.526         3.697         5.708
##      useless Speciesversicolor Speciesvirginica
## 85      5.712             1             0
## 104     6.569             0             1
## 30      4.299             0             0
## 53      6.124             1             0
## 143     6.553             0             1
## 142     5.222             0             1
##      Sepal.Length:Sepal.Width Sepal.Length:Petal.Length
## 85              4.362              15.578
## 104             15.511              26.431
## 30              12.349               8.114
## 53              19.546             44.900
## 143              9.972             39.168
## 142             24.126             37.253
##      Sepal.Width:Petal.Length
## 85              7.650
## 104             15.787
## 30              5.269
## 53             13.294
## 143              9.620
## 142             21.103
##      Sepal.Length:Sepal.Width:Petal.Length
## 85              22.80
## 104             80.45
## 30              22.98
## 53             108.01
## 143             61.30
## 142            137.72
y <- train[, 4]
```

```
model_lars <- lars(x, y)
```

```
plot(model_lars)
```



```

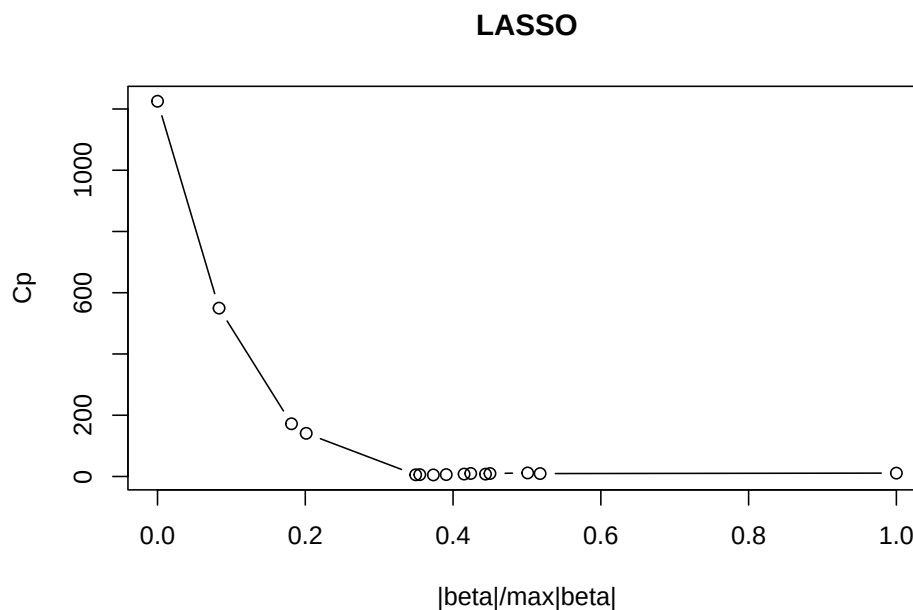
model_lars
##
## Call:
## lars(x = x, y = y)
## R-squared: 0.933
## Sequence of LASSO moves:
##      Petal.Length Speciesvirginica Speciesversicolor
## Var           4                      7                6
## Step          1                      2                3
##      Sepal.Width:Petal.Length
## Var                      10
## Step                     4
##      Sepal.Length:Sepal.Width:Petal.Length useless
## Var                      11                5
## Step                     5                6
##      Sepal.Width Sepal.Length:Petal.Length Sepal.Length
## Var              3                      9                2
## Step             7                      8                9
##      Sepal.Width:Petal.Length Sepal.Width:Petal.Length
## Var                      -10                10
## Step                     10                11
##      Sepal.Length:Sepal.Width Sepal.Width Sepal.Width
## Var              8                      -3                3

```

## Step	12	13	14
----------------	-----------	-----------	-----------

The fitted model's plot shows how variables are added (from left to right to the model). The text output shows that `Petal.Length` is the most important variable added to the model in step 1. Then `Speciesvirginica` is added and so on. This creates a sequence of nested models where one variable is added at a time. To select the best model Mallows's Cp statistic⁸ can be used.

```
plot(model_lars, plottype = "Cp")
```



```
best <- which.min(model_lars$Cp)

coef(model_lars, s = best)
##                (Intercept)
##                0.0000000
##                Sepal.Length
##                0.0000000
##                Sepal.Width
##                0.0000000
##                Petal.Length
##                0.0603837
##                useless
```

⁸[https://en.wikipedia.org/wiki/Mallows's Cp](https://en.wikipedia.org/wiki/Mallows%27s_Cp)

```
##                -0.0086551
##                Speciesversicolor
##                0.8463786
##                Speciesvirginica
##                1.4181850
##                Sepal.Length:Sepal.Width
##                0.0000000
##                Sepal.Length:Petal.Length
##                0.0000000
##                Sepal.Width:Petal.Length
##                0.0029688
## Sepal.Length:Sepal.Width:Petal.Length
##                0.0003151
```

The variables that are not selected have a β coefficient of 0.

To make predictions with this model, we first have to convert the test data into a design matrix with the dummy variables and interaction terms.

```
x_test <- model.matrix(~ . + Sepal.Length*Sepal.Width*Petal.Length,
  data = test[, -4])
head(x_test)
##      (Intercept) Sepal.Length Sepal.Width Petal.Length
## 128           1         8.017         1.541         3.515
## 92            1         5.268         4.064         6.064
## 50            1         5.461         4.161         1.117
## 134           1         6.055         2.951         4.599
## 8             1         4.900         5.096         1.086
## 58            1         5.413         4.728         5.990
##      useless Speciesversicolor Speciesvirginica
## 128    6.110                0                1
## 92     4.938                1                0
## 50     6.373                0                0
## 134    5.595                0                1
## 8      7.270                0                0
## 58     7.092                1                0
##      Sepal.Length:Sepal.Width Sepal.Length:Petal.Length
## 128                12.35                28.182
## 92                 21.41                31.944
## 50                 22.72                 6.102
## 134                17.87                27.847
## 8                  24.97                 5.319
## 58                 25.59                32.424
##      Sepal.Width:Petal.Length
## 128                5.417
## 92                 24.647
```

```
## 50          4.649
## 134         13.572
## 8           5.531
## 58         28.323
##      Sepal.Length:Sepal.Width:Petal.Length
## 128          43.43
## 92         129.83
## 50          25.39
## 134         82.19
## 8           27.10
## 58        153.31
```

Now we can compute the predictions.

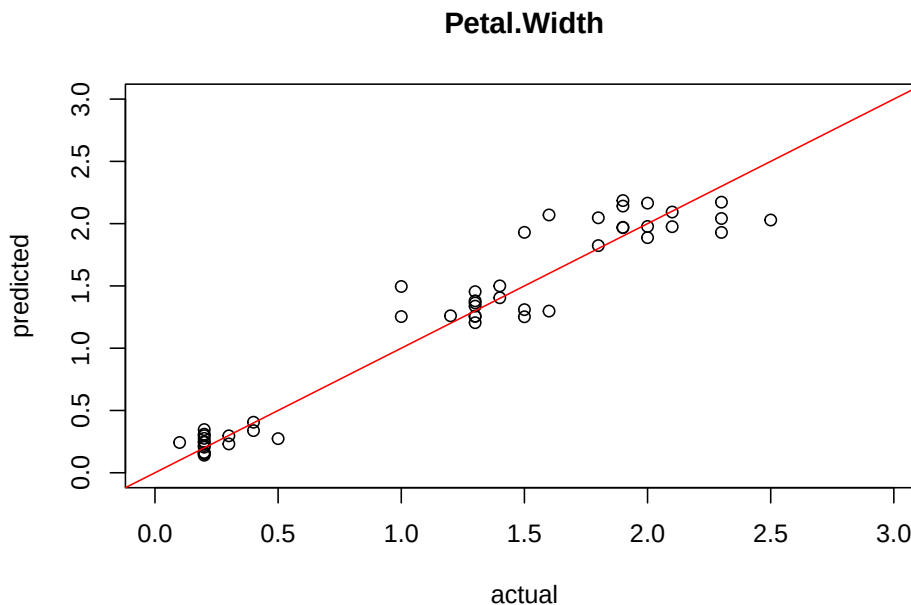
```
predict(model_lars, x_test[1:5,], s = best)
## $s
## 6
## 7
##
## $fraction
##      6
## 0.4286
##
## $mode
## [1] "step"
##
## $fit
##      128      92      50      134      8
## 1.8237 1.5003 0.2505 1.9300 0.2439
```

The prediction is the `fit` element. We can calculate the RMSE.

```
pred <- predict(model_lars, x_test, s = best)$fit
RMSE(pred, test$Petal.Width)
## [1] 0.1907
```

And visualize the prediction.

```
plot(test[, "Petal.Width"],
     pred,
     xlim=c(0,3), ylim=c(0,3),
     xlab = "actual", ylab = "predicted",
     main = "Petal.Width")
abline(0,1, col = "red")
```



```
cor(test[, "Petal.Width"],
     pred)
## [1] 0.9686
```

The model shows good predictive power on the test set.

B.8.3 ANNs

Regression can be performed using artificial neural networks⁹ with typically a linear final layer. We will create a network using a single hidden layer with 3 neurons (a manually tuned hyper parameter) and a linear output layer (set via `linout`).

```
library(nnet)

model_nnet <- nnet(Petal.Width ~ ., data = train, size = 3, linout = TRUE)
## # weights: 25
## initial value 82.633473
## iter 10 value 14.094963
## iter 20 value 3.767394
## iter 30 value 3.372093
## iter 40 value 3.264074
## iter 50 value 3.224384
```

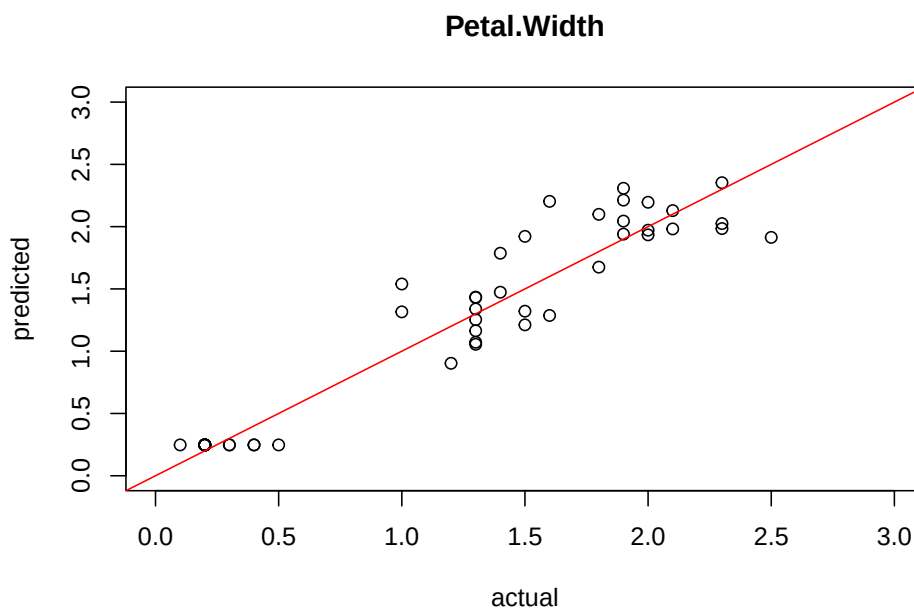
⁹[https://en.wikipedia.org/wiki/Neural_network_\(machine_learning\)](https://en.wikipedia.org/wiki/Neural_network_(machine_learning))


```
## iter 60 value 3.201191
## iter 70 value 3.182708
## iter 80 value 3.117583
## iter 90 value 3.087831
## iter 100 value 3.009898
## final value 3.009898
## stopped after 100 iterations
model_nnet
## a 6-3-1 network with 25 weights
## inputs: Sepal.Length Sepal.Width Petal.Length useless Speciesversicolor Speciesvirginica
## output(s): Petal.Width
## options were - linear output units

pred <- predict(model_nnet, test)
RMSE(pred, test$Petal.Width)
## [1] 0.2322
```

And visualize the quality.

```
plot(test[, "Petal.Width"], pred,
     xlim = c(0,3), ylim = c(0,3),
     xlab = "actual", ylab = "predicted",
     main = "Petal.Width")
abline(0,1, col = "red")
```



```
cor(test[, "Petal.Width"], pred)
##           [,1]
## [1,] 0.9546
```

Note: It is often necessary to scale the inputs to the ANN so it can learn effectively. It is very popular to scale all inputs to the ranges $[0, 1]$ or $[-1, 1]$. The ranges of the Iris dataset are fine, so there was no need for scaling.

B.8.4 Other Types of Regression

- Robust regression: robust against violation of assumptions like heteroscedasticity and outliers (`robustbase::robglm()` and `robustbase::robglm()`).
- Generalized linear models (`glm()`). An example is logistic regression discussed in the next chapter.
- Nonlinear least squares (`nls()`).

B.9 Exercises

We will again use the Palmer penguin data for the exercises.

```
library(palmerpenguins)
head(penguins)
## # A tibble: 6 x 8
##   species island    bill_length_mm bill_depth_mm
##   <chr>    <chr>          <dbl>         <dbl>
## 1 Adelie  Torgersen         39.1          18.7
## 2 Adelie  Torgersen         39.5          17.4
## 3 Adelie  Torgersen         40.3           18
## 4 Adelie  Torgersen          NA           NA
## 5 Adelie  Torgersen         36.7          19.3
## 6 Adelie  Torgersen         39.3          20.6
## # i 4 more variables: flipper_length_mm <dbl>,
## #   body_mass_g <dbl>, sex <chr>, year <dbl>
```

Create an R markdown document that performs the following:

1. Create a linear regression model to predict the weight of a penguin (`body_mass_g`).
2. How high is the R-squared. What does it mean.
3. What variables are significant, what are not?

4. Use stepwise variable selection to remove unnecessary variables.
5. Predict the weight for the following new penguin:

```
new_penguin <- tibble(  
  species = factor("Adelie",  
    levels = c("Adelie", "Chinstrap", "Gentoo")),  
  island = factor("Dream",  
    levels = c("Biscoe", "Dream", "Torgersen")),  
  bill_length_mm = 39.8,  
  bill_depth_mm = 19.1,  
  flipper_length_mm = 184,  
  body_mass_g = NA,  
  sex = factor("male", levels = c("female", "male")),  
  year = 2007  
)  
new_penguin  
## # A tibble: 1 x 8  
##   species island bill_length_mm bill_depth_mm  
##   <fct>   <fct>         <dbl>         <dbl>  
## 1 Adelie Dream           39.8           19.1  
## # i 4 more variables: flipper_length_mm <dbl>,  
## #   body_mass_g <lgl>, sex <fct>, year <dbl>
```

6. Create a regression tree. Look at the tree and explain what it does. Then use the regression tree to predict the weight for the above penguin.

Appendix C

Logistic Regression

This chapter introduces the popular classification method logistic regression more in detail. Logistic regression is introduced as an alternative classification method in Chapter 4 of Introduction to Data Mining.

```
pkgs <- c("glmnet", "caret")

pkgs_install <- pkgs[!(pkgs %in% installed.packages()[,"Package"])]
if(length(pkgs_install)) install.packages(pkgs_install)
```

The packages used for this chapter are:

- *caret* (Kuhn 2024)
- *glmnet* (Friedman et al. 2025)

C.1 Introduction

Logistic regression contains the word regression, but it is actually a statistical classification model to predict the probability p of a binary outcome given a set of features. It is a very powerful classification model that can be fit very quickly. It is one of the first classification models you should try on new data.

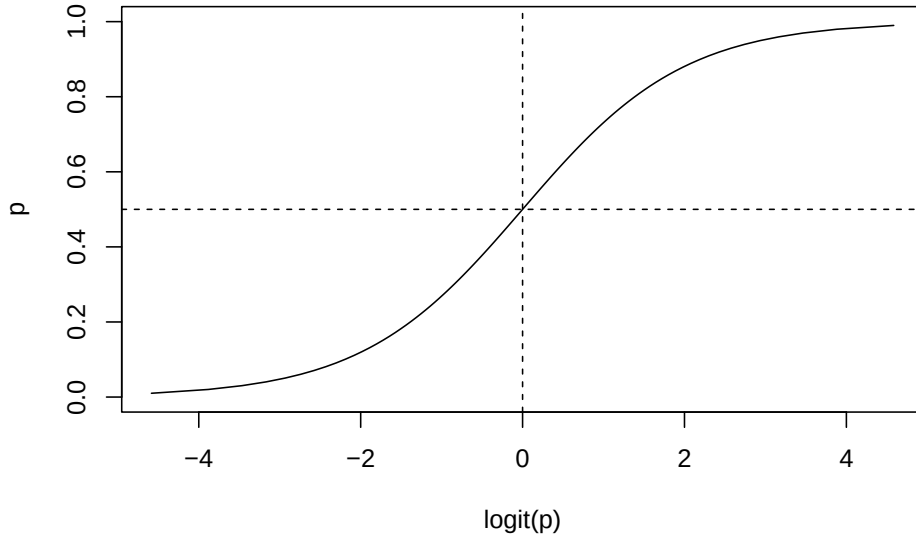
Logistic regression is a generalized linear model¹ with the logit as the link function and a binomial error distribution. It can be thought of as a linear regression with the log odds ratio (logit) of the binary outcome as the dependent variable:

$$\text{logit}(p) = \ln\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots$$

¹https://en.wikipedia.org/wiki/Generalized_linear_model

The logit function links the probability p to the linear regression by converting a number in the probability range $[0, 1]$ to the range $[-\infty, +\infty]$.

```
logit <- function(p) log(p/(1-p))
p <- seq(0, 1, length.out = 100)
plot(logit(p), p, type = "l")
abline(h = 0.5, lty = 2)
abline(v = 0, lty = 2)
```



The figure above shows actually the inverse of the logit function. The inverse of the logit function is called logistic (or sigmoid) function² ($\sigma(\cdot)$) which is often used in ML, and especially for artificial neural networks, to squash the set of real numbers to the $[0, 1]$ interval. Using the inverse function, we see that the probability of the outcome p is modeled by the logistic function of the linear regression:

$$p = \frac{e^{\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots}}{1 + e^{\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots}} = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots)}} = \sigma(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \dots)$$

After the $\beta = (\beta_0, \beta_1, \dots)$ parameter vector is fitted using training data by minimizing the log loss (i.e., cross-entropy loss), the equation above can be used to predict the probability p given a new data point $\mathbf{x} = (x_1, x_2, \dots)$. If the predicted $p > .5$ then we predict that the event happens, otherwise we predict that it does not happen.

The outcome itself is binary and therefore has a Bernoulli distribution. Since we have multiple examples in our data we draw several times from this distribution

²https://en.wikipedia.org/wiki/Logistic_function

resulting in a Binomial distribution for the number of successful events drawn. Logistic regression therefore uses a logit link function to link the probability of the event to the linear regression and the distribution family is Binomial.

C.2 Data Preparation

We load and shuffle the data. We also add a useless variable to see if the logistic regression removes it.

```
data(iris)
set.seed(100) # for reproducibility

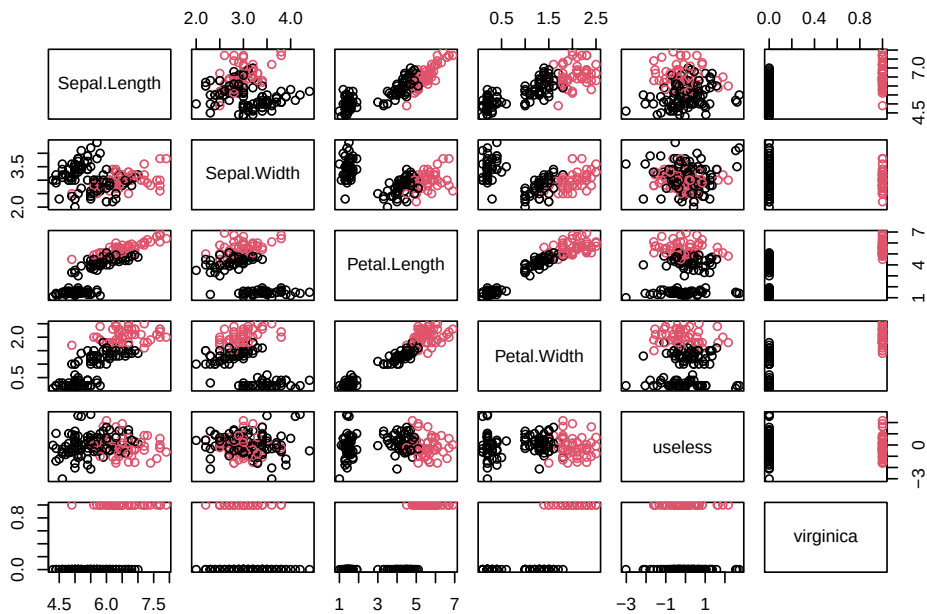
x <- iris[sample(1:nrow(iris)),]
x <- cbind(x, useless = rnorm(nrow(x)))
```

We create a binary classification problem by asking if a flower is of species *Virginica* or not. We create new logical variable called `virginica` and remove the `Species` column.

```
x$virginica <- x$Species == "virginica"
x$Species <- NULL
```

We can visualize the data using a scatter plot matrix and use the color red for `virginica == TRUE` and black for the other flowers.

```
plot(x, col=x$virginica + 1)
```



C.3 A first Logistic Regression Model

Logistic regression is a generalized linear model (GLM) with logit as the link function and a binomial distribution. The `glm()` function is provided by the R core package `stats` which is installed with R and automatically loads when R is started.

```
model <- glm(virginica ~ .,
  family = binomial(logit), data = x)
## Warning: glm.fit: fitted probabilities numerically 0 or 1
## occurred
```

About the warning: `glm.fit: fitted probabilities numerically 0 or 1 occurred` means that the data is possibly linearly separable.

```
model
##
## Call: glm(formula = virginica ~ ., family = binomial(logit), data = x)
##
## Coefficients:
## (Intercept) Sepal.Length Sepal.Width Petal.Length
## -41.649      -2.531      -6.448       9.376
## Petal.Width      useless
## 17.696         0.098
```



```
##
## Degrees of Freedom: 149 Total (i.e. Null); 144 Residual
## Null Deviance: 191
## Residual Deviance: 11.9 AIC: 23.9
```

Check which features are significant?

```
summary(model)
##
## Call:
## glm(formula = virginica ~ ., family = binomial(logit), data = x)
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)  -41.649    26.556   -1.57    0.117
## Sepal.Length  -2.531     2.458   -1.03    0.303
## Sepal.Width   -6.448     4.794   -1.34    0.179
## Petal.Length   9.376     4.763    1.97    0.049 *
## Petal.Width   17.696    10.632    1.66    0.096 .
## useless        0.098     0.807    0.12    0.903
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##      Null deviance: 190.954  on 149  degrees of freedom
## Residual deviance: 11.884  on 144  degrees of freedom
## AIC: 23.88
##
## Number of Fisher Scoring iterations: 12
```

AIC (Akaike information criterion³) is a measure of how good the model is. Smaller is better. It can be used for model selection.

The parameter estimates in the coefficients table are log odds. The * and . indicate if the effect of the parameter is significantly different from 0. Positive numbers mean that increasing the variable increases the predicted probability and negative numbers mean that the probability decreases. For example, observing a larger Petal.Length increases the predicted probability for the flower to be of class Virginica. This effect is significant and you can verify it in the scatter plot above. For Petal.Length, the red dots have larger values than the black dots.

³https://en.wikipedia.org/wiki/Akaike_information_criterion

C.4 Stepwise Variable Selection

Only two variables were flagged as significant. We can remove insignificant variables by trying to remove one variable at a time as long as the model does not significantly deteriorate (according to the AIC). This variable selection process is done automatically by the `step()` function.

```
model2 <- step(model, data = x)
## Start: AIC=23.88
## virginica ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width +
##      useless
## Warning: glm.fit: fitted probabilities numerically 0 or 1
## occurred
## Warning: glm.fit: fitted probabilities numerically 0 or 1
## occurred
## Warning: glm.fit: fitted probabilities numerically 0 or 1
## occurred
## Warning: glm.fit: fitted probabilities numerically 0 or 1
## occurred
## Warning: glm.fit: fitted probabilities numerically 0 or 1
## occurred
##
##           Df Deviance  AIC
## - useless      1    11.9 21.9
## - Sepal.Length  1    13.2 23.2
## <none>           11.9 23.9
## - Sepal.Width   1    14.8 24.8
## - Petal.Width   1    22.4 32.4
## - Petal.Length  1    25.9 35.9
## Warning: glm.fit: fitted probabilities numerically 0 or 1
## occurred
##
## Step: AIC=21.9
## virginica ~ Sepal.Length + Sepal.Width + Petal.Length + Petal.Width
## Warning: glm.fit: fitted probabilities numerically 0 or 1
## occurred
## Warning: glm.fit: fitted probabilities numerically 0 or 1
## occurred
## Warning: glm.fit: fitted probabilities numerically 0 or 1
## occurred
## Warning: glm.fit: fitted probabilities numerically 0 or 1
## occurred
##
##           Df Deviance  AIC
## - Sepal.Length  1    13.3 21.3
## <none>           11.9 21.9
## - Sepal.Width   1    15.5 23.5
```

```
## - Petal.Width 1 23.8 31.8
## - Petal.Length 1 25.9 33.9
## Warning: glm.fit: fitted probabilities numerically 0 or 1
## occurred
##
## Step: AIC=21.27
## virginica ~ Sepal.Width + Petal.Length + Petal.Width
## Warning: glm.fit: fitted probabilities numerically 0 or 1
## occurred
## Warning: glm.fit: fitted probabilities numerically 0 or 1
## occurred
##
##           Df Deviance  AIC
## <none>           13.3 21.3
## - Sepal.Width 1 20.6 26.6
## - Petal.Length 1 27.4 33.4
## - Petal.Width 1 31.5 37.5
summary(model2)
##
## Call:
## glm(formula = virginica ~ Sepal.Width + Petal.Length + Petal.Width,
##      family = binomial(logit), data = x)
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept)   -50.53      23.99   -2.11   0.035 *
## Sepal.Width    -8.38       4.76   -1.76   0.079 .
## Petal.Length    7.87       3.84    2.05   0.040 *
## Petal.Width    21.43      10.71    2.00   0.045 *
## ---
## Signif. codes:
## 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
##      Null deviance: 190.954  on 149  degrees of freedom
## Residual deviance:  13.266  on 146  degrees of freedom
## AIC: 21.27
##
## Number of Fisher Scoring iterations: 12
```

The estimates $(\beta_0, \beta_1, \dots)$ are log-odds and can be converted into odds using $\exp(\beta)$. A negative log-odds ratio means that the odds go down with an increase in the value of the predictor. A predictor with a positive log-odds ratio increases the odds. In this case, the odds of looking at a Virginica iris goes down with Sepal.Width and increases with the other two predictors.

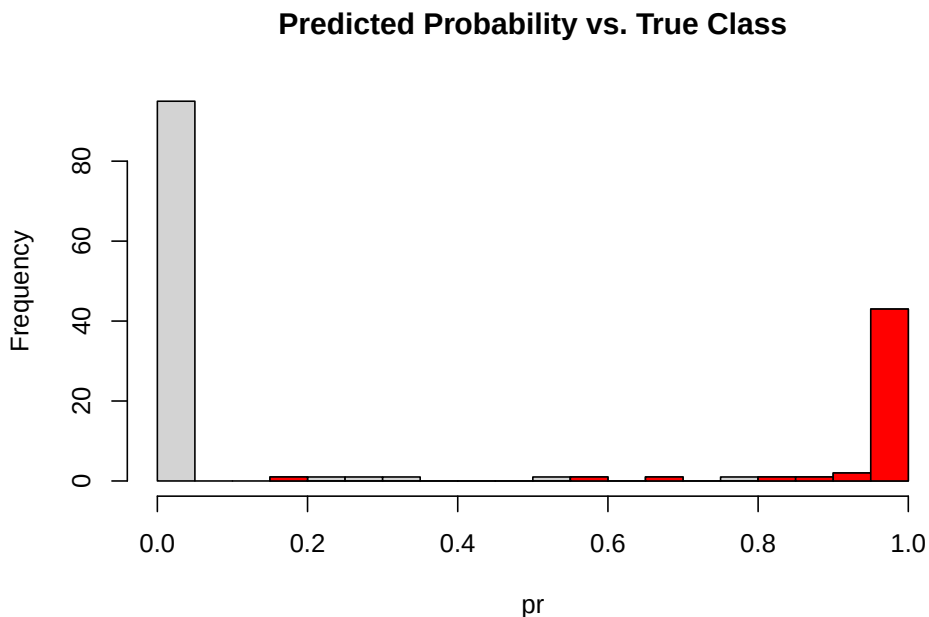
C.5 Calculate the Response

Note: we do here in-sample testing on the data we learned the data from. To get a generalization error estimate you should use a test set or cross-validation!

```
pr <- predict(model2, x, type = "response")
round(pr[1:10], 2)
## 102 112  4  55  70  98 135  7  43 140
## 1.00 1.00 0.00 0.00 0.00 0.00 0.86 0.00 0.00 1.00
```

The response is the predicted probability of the flower being of species Virginica. The probabilities of the first 10 flowers are shown. Below is a histogram of predicted probabilities. The color is used to show the examples that have the true class Virginica.

```
hist(pr, breaks = 20, main = "Predicted Probability vs. True Class")
hist(pr[x$virginica == TRUE], col = "red", breaks = 20, add = TRUE)
```



C.6 Check Classification Performance

Here we perform in-sample evaluation on the training set. To get an estimate for generalization error, we should calculate the performance on a held out test set.

The predicted class is calculated by checking if the predicted probability is larger than .5.

```
pred <- pr > .5
```

Now we can create a confusion table and calculate the accuracy.

```
tbl <- table(predicted = pred, actual = x$virginica)
tbl
##           actual
## predicted FALSE TRUE
##    FALSE     98    1
##    TRUE      2    49
sum(diag(tbl))/sum(tbl)
## [1] 0.98
```

We can also use caret's more advanced function `caret::confusionMatrix()`. Our code above uses logical vectors. For caret, we need to make sure that both, the reference and the predictions are coded as **factor**.

```
caret::confusionMatrix(
  reference = factor(x$virginica, labels = c("Yes", "No"), levels = c(TRUE, FALSE)),
  data = factor(pred, labels = c("Yes", "No"), levels = c(TRUE, FALSE)))
## Confusion Matrix and Statistics
##
##           Reference
## Prediction Yes No
##    Yes  49  2
##    No   1 98
##
##              Accuracy : 0.98
##              95% CI : (0.943, 0.996)
##    No Information Rate : 0.667
##    P-Value [Acc > NIR] : <2e-16
##
##              Kappa : 0.955
##
##    Mcnemar's Test P-Value : 1
##
##              Sensitivity : 0.980
##              Specificity : 0.980
##    Pos Pred Value : 0.961
##    Neg Pred Value : 0.990
##              Prevalence : 0.333
##    Detection Rate : 0.327
```

```
##      Detection Prevalence : 0.340
##      Balanced Accuracy : 0.980
##
##      'Positive' Class : Yes
##
```

We see that the model performs well with a very high accuracy and kappa value.

C.7 Regularized Logistic Regression

`glmnet::glmnet()` fits generalized linear models (including logistic regression) using regularization via penalized maximum likelihood. The regularization parameter λ is a hyperparameter and `glmnet` can use cross-validation to find an appropriate value. `glmnet` does not have a function interface, so we have to supply a matrix for `X` and a vector of responses for `y`.

```
library(glmnet)
## Loaded glmnet 4.1-10
X <- as.matrix(x[, 1:5])
y <- x$virginica

fit <- cv.glmnet(X, y, family = "binomial")
fit
##
## Call:  cv.glmnet(x = X, y = y, family = "binomial")
##
## Measure: Binomial Deviance
##
##      Lambda Index Measure      SE Nonzero
## min 0.00164    59   0.126 0.0456         5
## 1se 0.00664    44   0.167 0.0422         3
```

There are several selection rules for lambda, we look at the coefficients of the logistic regression using the lambda that gives the most regularized model such that the cross-validated error is within one standard error of the minimum cross-validated error.

```
coef(fit, s = fit$lambda.1se)
## 6 x 1 sparse Matrix of class "dgCMatrix"
##           s=0.00664
## (Intercept)   -16.961
## Sepal.Length      .
## Sepal.Width     -1.766
```

```
## Petal.Length      2.197
## Petal.Width       6.820
## useless           .
```

A dot means 0. We see that the predictors Sepal.Length and useless are not used in the prediction giving a models similar to stepwise variable selection above.

A predict function is provided. We need to specify what regularization to use and that we want to predict a class label.

```
predict(fit, newx = X[1:5,], s = fit$lambda.1se, type = "class")
##      s=0.00664
## 102 "TRUE"
## 112 "TRUE"
##  4  "FALSE"
## 55  "FALSE"
## 70  "FALSE"
```

Glmnet provides supports many types of generalized linear models. Examples can be found in the article An Introduction to glmnet⁴.

C.8 Multinomial Logistic Regression

Regular logistic regression predicts only one outcome of a binary event represented by two classes. Extending this model to data with more than two classes is called multinomial logistic regression⁵, (or log-linear model). A popular implementation uses simple artificial neural networks. Regular logistic regression is equivalent to a single neuron with a sigmoid (i.e., logistic) activation function optimized with cross-entropy loss. For multinomial logistic regression, one neuron is used for each class and the probability distribution is calculated with the softmax activation. This extension is implemented in `nnet::multinom()`.

```
set.seed(100)
x <- iris[sample(1:nrow(iris)), ]

model <- nnet::multinom(Species ~., data = x)
## # weights:  18 (10 variable)
## initial value 164.791843
## iter  10 value 16.177348
## iter  20 value 7.111438
## iter  30 value 6.182999
```

⁴<https://glmnet.stanford.edu/articles/glmnet.html>

⁵https://en.wikipedia.org/wiki/Multinomial_logistic_regression

```
## iter 40 value 5.984028
## iter 50 value 5.961278
## iter 60 value 5.954900
## iter 70 value 5.951851
## iter 80 value 5.950343
## iter 90 value 5.949904
## iter 100 value 5.949867
## final value 5.949867
## stopped after 100 iterations
model
## Call:
## nnet::multinom(formula = Species ~ ., data = x)
##
## Coefficients:
##              (Intercept) Sepal.Length Sepal.Width
## versicolor      18.69         -5.458         -8.707
## virginica      -23.84         -7.924        -15.371
##              Petal.Length Petal.Width
## versicolor      14.24          -3.098
## virginica       23.66          15.135
##
## Residual Deviance: 11.9
## AIC: 31.9
```

We get a β vector with weights for two of the three classes. The third class is used as the default class with all weights set to 0. This can be interpreted as comparing the log odds of each of the two classes with the default class. A positive number means that increasing the variable makes the class more likely and a negative number means the opposite.

Predict the class for the first 5 flowers in the training data.

```
x[1:5, ]
##      Sepal.Length Sepal.Width Petal.Length Petal.Width
## 102           5.8         2.7         5.1         1.9
## 112           6.4         2.7         5.3         1.9
## 4            4.6         3.1         1.5         0.2
## 55           6.5         2.8         4.6         1.5
## 70           5.6         2.5         3.9         1.1
##      Species
## 102 virginica
## 112 virginica
## 4      setosa
## 55 versicolor
## 70 versicolor
predict(model, x[1:5,])
```



```
## [1] virginica virginica setosa versicolor versicolor
## Levels: setosa versicolor virginica
```

The package `glmnet` implements also multinomial logistic regression using `glmnet(..., family = "multinomial")`.

C.9 Exercises

We will again use the Palmer penguin data for the exercises.

```
library(palmerpenguins)
head(penguins)
## # A tibble: 6 x 8
##   species island bill_length_mm bill_depth_mm
##   <chr>   <chr>         <dbl>         <dbl>
## 1 Adelie  Torgersen         39.1          18.7
## 2 Adelie  Torgersen         39.5          17.4
## 3 Adelie  Torgersen         40.3           18
## 4 Adelie  Torgersen          NA           NA
## 5 Adelie  Torgersen         36.7          19.3
## 6 Adelie  Torgersen         39.3          20.6
## # i 4 more variables: flipper_length_mm <dbl>,
## #   body_mass_g <dbl>, sex <chr>, year <dbl>
```

Create an R markdown document that performs the following:

1. Create a test and a training data set (see section Holdout Method in Chapter 3).
2. Create a logistic regression using the training set to predict the variable `sex`.
3. Use stepwise variable selection. What variables are selected?
4. What do the parameters for each of the selected features tell you?
5. Predict the sex of the penguins in the test set. Create a confusion table and calculate the accuracy and discuss how well the model works.

References

- Agrawal, Rakesh, Tomasz Imielinski, and Arun Swami. 1993. "Mining Association Rules Between Sets of Items in Large Databases." In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 207–16. Washington, D.C., United States: ACM Press.
- Bates, Douglas, Martin Maechler, and Mikael Jagan. 2025. *Matrix: Sparse and Dense Matrix Classes and Methods*. <https://Matrix.R-forge.R-project.org>.
- Blake, Catherine L., and Christopher J. Merz. 1998. *UCI Repository of Machine Learning Databases*. Irvine, CA: University of California, Irvine, Department of Information; Computer Sciences.
- Breiman, Leo, Adele Cutler, Andy Liaw, and Matthew Wiener. 2024. *randomForest: Breiman and Cutlers Random Forests for Classification and Regression*. <https://www.stat.berkeley.edu/~breiman/RandomForests/>.
- Buchta, Christian, and Michael Hahsler. 2024. *arulesSequences: Mining Frequent Sequences*. <https://doi.org/10.32614/CRAN.package.arulesSequences>.
- Carr, Dan, Nicholas Lewin-Koh, Martin Maechler, and Deepayan Sarkar. 2024. *Hexbin: Hexagonal Binning Routines*. <https://github.com/edzer/hexbin>.
- Chen, Tianqi, Tong He, Michael Benesty, Vadim Khotilovich, Yuan Tang, Hyunsu Cho, Kailong Chen, et al. 2025. *Xgboost: Extreme Gradient Boosting*. <https://github.com/dmlc/xgboost>.
- Chen, Ying-Ju, Fadel M. Megahed, L. Allison Jones-Farmer, and Steven E. Rigdon. 2023. *Basemodels: Baseline Models for Classification and Regression*. <https://github.com/Ying-Ju/basemodels>.
- Fräley, Chris, Adrian E. Raftery, and Luca Scrucca. 2024. *Mclust: Gaussian Mixture Modelling for Model-Based Clustering, Classification, and Density Estimation*. <https://mclust-org.github.io/mclust/>.
- Friedman, Jerome, Trevor Hastie, and Robert Tibshirani. 2010. "Regularization Paths for Generalized Linear Models via Coordinate Descent." *Journal of Statistical Software* 33 (1): 1–22. <https://doi.org/10.18637/jss.v033.i01>.
- Friedman, Jerome, Trevor Hastie, Rob Tibshirani, Balasubramanian Narasimhan, Kenneth Tay, Noah Simon, and James Yang. 2025. *Glmmnet: Lasso and Elastic-Net Regularized Generalized Linear Models*. <https://glmnet.stanford.edu>.
- Grolemund, Garrett, and Hadley Wickham. 2011. "Dates and Times Made

- Easy with lubridate.” *Journal of Statistical Software* 40 (3): 1–25. <https://www.jstatsoft.org/v40/i03/>.
- Hahsler, Michael. 2017a. “An Experimental Comparison of Seriation Methods for One-Mode Two-Way Data.” *European Journal of Operational Research* 257 (1): 133–43. <https://doi.org/10.1016/j.ejor.2016.08.066>.
- . 2017b. “ArulesViz: Interactive Visualization of Association Rules with R.” *R Journal* 9 (2): 163–75. <https://doi.org/10.32614/RJ-2017-047>.
- . 2021. *An R Companion for Introduction to Data Mining*. Online Book. https://mhahsler.github.io/Introduction_to_Data_Mining_R_Examples/book.
- . 2025. *arulesViz: Visualizing Association Rules and Frequent Itemsets*. <https://github.com/mhahsler/arulesViz>.
- Hahsler, Michael, Christian Buchta, Bettina Gruen, and Kurt Hornik. 2025. *Arules: Mining Association Rules and Frequent Itemsets*. <https://github.com/mhahsler/arules>.
- Hahsler, Michael, Christian Buchta, and Kurt Hornik. 2025. *Seriation: Infrastructure for Ordering Objects Using Seriation*. <https://github.com/mhahsler/seriation>.
- Hahsler, Michael, Sudheer Chelluboina, Kurt Hornik, and Christian Buchta. 2011. “The Arules r-Package Ecosystem: Analyzing Interesting Patterns from Large Transaction Datasets.” *Journal of Machine Learning Research* 12: 1977–81. <https://jmlr.csail.mit.edu/papers/v12/hahsler11a.html>.
- Hahsler, Michael, Bettina Gruen, and Kurt Hornik. 2005. “Arules – A Computational Environment for Mining Association Rules and Frequent Item Sets.” *Journal of Statistical Software* 14 (15): 1–25. <https://doi.org/10.18637/jss.v014.i15>.
- Hahsler, Michael, Bettina Grün, and Kurt Hornik. 2005. “Arules – A Computational Environment for Mining Association Rules and Frequent Item Sets.” *Journal of Statistical Software* 14 (15): 1–25. <http://www.jstatsoft.org/v14/i15/>.
- Hahsler, Michael, Kurt Hornik, and Christian Buchta. 2008. “Getting Things in Order: An Introduction to the r Package Seriation.” *Journal of Statistical Software* 25 (3): 1–34. <https://doi.org/10.18637/jss.v025.i03>.
- Hahsler, Michael, and Matthew Piekenbrock. 2025. *Dbscan: Density-Based Spatial Clustering of Applications with Noise (DBSCAN) and Related Algorithms*. <https://github.com/mhahsler/dbscan>.
- Hahsler, Michael, Matthew Piekenbrock, and Derek Doran. 2019. “dbscan: Fast Density-Based Clustering with R.” *Journal of Statistical Software* 91 (1): 1–30. <https://doi.org/10.18637/jss.v091.i01>.
- Hastie, Trevor, and Brad Efron. 2022. *Lars: Least Angle Regression, Lasso and Forward Stagewise*. <https://doi.org/10.1214/0090536040000000067>.
- Hennig, Christian. 2024. *Fpc: Flexible Procedures for Clustering*. <https://www.unibo.it/sitoweb/christian.hennig/en/>.
- Hornik, Kurt. 2023. *RWeka: R/Weka Interface*. <https://doi.org/10.32614/CRAN.package.RWeka>.
- Hornik, Kurt, Christian Buchta, and Achim Zeileis. 2009. “Open-Source Ma-

- chine Learning: R Meets Weka.” *Computational Statistics* 24 (2): 225–32. <https://doi.org/10.1007/s00180-008-0119-7>.
- Horst, Allison, Alison Hill, and Kristen Gorman. 2022. *Palmerpenguins: Palmer Archipelago (Antarctica) Penguin Data*. <https://allisonhorst.github.io/palmerpenguins/>.
- Hothorn, Torsten, Peter Buehlmann, Sandrine Dudoit, Annette Molinaro, and Mark Van Der Laan. 2006. “Survival Ensembles.” *Biostatistics* 7 (3): 355–73. <https://doi.org/10.1093/biostatistics/kxj011>.
- Hothorn, Torsten, Kurt Hornik, Carolin Strobl, and Achim Zeileis. 2025. *Party: A Laboratory for Recursive Partytioning*. <http://party.R-forge.R-project.org>.
- Hothorn, Torsten, Kurt Hornik, and Achim Zeileis. 2006. “Unbiased Recursive Partitioning: A Conditional Inference Framework.” *Journal of Computational and Graphical Statistics* 15 (3): 651–74. <https://doi.org/10.1198/106186006X133933>.
- Karatzoglou, Alexandros, Alex Smola, and Kurt Hornik. 2024. *Kernlab: Kernel-Based Machine Learning Lab*. <https://doi.org/10.32614/CRAN.package.kernlab>.
- Karatzoglou, Alexandros, Alex Smola, Kurt Hornik, and Achim Zeileis. 2004. “Kernlab – an S4 Package for Kernel Methods in R.” *Journal of Statistical Software* 11 (9): 1–20. <https://doi.org/10.18637/jss.v011.i09>.
- Kassambara, Alboukadel. 2023. *Ggcorrplot: Visualization of a Correlation Matrix Using Ggplot2*. <http://www.sthda.com/english/wiki/ggcorrplot-visualization-of-a-correlation-matrix-using-ggplot2>.
- Kassambara, Alboukadel, and Fabian Mundt. 2020. *Factoextra: Extract and Visualize the Results of Multivariate Data Analyses*. <http://www.sthda.com/english/rpkgs/factoextra>.
- Krijthe, Jesse. 2023. *Rtsne: T-Distributed Stochastic Neighbor Embedding Using a Barnes-Hut Implementation*. <https://github.com/jkrijthe/Rtsne>.
- Kuhn, Max. 2024. *Caret: Classification and Regression Training*. <https://github.com/topepo/caret/>.
- Kuhn, Max, and Ross Quinlan. 2025. *C50: C5.0 Decision Trees and Rule-Based Models*. <https://topepo.github.io/C5.0/>.
- Kuhn, and Max. 2008. “Building Predictive Models in r Using the Caret Package.” *Journal of Statistical Software* 28 (5): 1–26. <https://doi.org/10.18637/jss.v028.i05>.
- Leisch, Friedrich, and Evgenia Dimitriadou. 2024. *Mlbench: Machine Learning Benchmark Problems*. <https://doi.org/10.32614/CRAN.package.mlbench>.
- Liaw, Andy, and Matthew Wiener. 2002. “Classification and Regression by randomForest.” *R News* 2 (3): 18–22. <https://CRAN.R-project.org/doc/Rnews/>.
- Maechler, Martin, Peter Rousseeuw, Anja Struyf, and Mia Hubert. 2025. *Cluster: “Finding Groups in Data”: Cluster Analysis Extended Rousseeuw Et Al*. <https://svn.r-project.org/R-packages/trunk/cluster/>.
- Meyer, David, and Christian Buchta. 2022. *Proxy: Distance and Similarity Measures*. <https://doi.org/10.32614/CRAN.package.proxy>.

- Meyer, David, Evgenia Dimitriadou, Kurt Hornik, Andreas Weingessel, and Friedrich Leisch. 2024. *E1071: Misc Functions of the Department of Statistics, Probability Theory Group (Formerly: E1071), TU Wien*. <https://doi.org/10.32614/CRAN.package.e1071>.
- Milborrow, Stephen. 2025. *Rpart.plot: Plot Rpart Models: An Enhanced Version of Plot.rpart*. <http://www.milbo.org/rpart-plot/index.html>.
- Müller, Kirill, and Hadley Wickham. 2025. *Tibble: Simple Data Frames*. <https://tibble.tidyverse.org/>.
- R Core Team. 2025. *R: A Language and Environment for Statistical Computing*. Vienna, Austria: R Foundation for Statistical Computing. <https://www.R-project.org/>.
- Ripley, Brian. 2025. *Nnet: Feed-Forward Neural Networks and Multinomial Log-Linear Models*. <http://www.stats.ox.ac.uk/pub/MASS4/>.
- Ripley, Brian, and Bill Venables. 2025. *MASS: Support Functions and Datasets for Venables and Ripley's MASS*. <http://www.stats.ox.ac.uk/pub/MASS4/>.
- Robin, Xavier, Natacha Turck, Alexandre Hainard, Natalia Tiberti, Frédérique Lisacek, Jean-Charles Sanchez, and Markus Müller. 2011. “pROC: An Open-Source Package for r and s+ to Analyze and Compare ROC Curves.” *BMC Bioinformatics* 12: 77.
- . 2025. *pROC: Display and Analyze ROC Curves*. <https://xrobin.github.io/pROC/>.
- Roeveer, Christian, Nils Raabe, Karsten Luebke, Uwe Ligges, Gero Szepannek, Marc Zentgraf, and David Meyer. 2023. *klaR: Classification and Visualization*. <https://statistik.tu-dortmund.de>.
- Romanski, Piotr, Lars Kotthoff, and Patrick Schratz. 2023. *FSelector: Selecting Attributes*. <https://github.com/larskotthoff/fselector>.
- Sarkar, Deepayan. 2008. *Lattice: Multivariate Data Visualization with r*. New York: Springer. <http://lmdvr.r-forge.r-project.org>.
- . 2025. *Lattice: Trellis Graphics for r*. <https://lattice.r-forge.r-project.org/>.
- Schloerke, Barret, Di Cook, Joseph Larmarange, Francois Briatte, Moritz Marbach, Edwin Thoen, Amos Elberg, and Jason Crowley. 2025. *GGally: Extension to Ggplot2*. <https://ggobi.github.io/ggally/>.
- Scrucca, Luca, Chris Fraley, T. Brendan Murphy, and Adrian E. Raftery. 2023. *Model-Based Clustering, Classification, and Density Estimation Using mclust in R*. Chapman; Hall/CRC. <https://doi.org/10.1201/9781003277965>.
- Sievert, Carson. 2020. *Interactive Web-Based Data Visualization with r, Plotly, and Shiny*. Chapman; Hall/CRC. <https://plotly-r.com>.
- Sievert, Carson, Chris Parmer, Toby Hocking, Scott Chamberlain, Karthik Ram, Marianne Corvellec, and Pedro Despouy. 2025. *Plotly: Create Interactive Web Graphics via Plotly.js*. <https://plotly-r.com>.
- Simon, Noah, Jerome Friedman, Trevor Hastie, and Robert Tibshirani. 2011. “Regularization Paths for Cox’s Proportional Hazards Model via Coordinate Descent.” *Journal of Statistical Software* 39 (5): 1–13. <https://doi.org/10.18637/jss.v039.i05>.
- Spinu, Vitalie, Garrett Grolemond, and Hadley Wickham. 2024. *Lubridate:*

- Make Dealing with Dates a Little Easier.* <https://lubridate.tidyverse.org>.
- Strobl, Carolin, Anne-Laure Boulesteix, Thomas Kneib, Thomas Augustin, and Achim Zeileis. 2008. “Conditional Variable Importance for Random Forests.” *BMC Bioinformatics* 9 (307). <https://doi.org/10.1186/1471-2105-9-307>.
- Strobl, Carolin, Anne-Laure Boulesteix, Achim Zeileis, and Torsten Hothorn. 2007. “Bias in Random Forest Variable Importance Measures: Illustrations, Sources and a Solution.” *BMC Bioinformatics* 8 (25). <https://doi.org/10.1186/1471-2105-8-25>.
- Tan, Pang-Ning, Michael S. Steinbach, Anuj Karpatne, and Vipin Kumar. 2017. *Introduction to Data Mining*. 2nd Edition. Pearson. <https://www-users.cs.umn.edu/~kumar001/dmbook>.
- Tan, Pang-Ning, Michael S. Steinbach, and Vipin Kumar. 2005. *Introduction to Data Mining*. 1st Edition. Addison-Wesley. <https://www-users.cs.umn.edu/~kumar001/dmbook/firsted.php>.
- Tay, J. Kenneth, Balasubramanian Narasimhan, and Trevor Hastie. 2023. “Elastic Net Regularization Paths for All Generalized Linear Models.” *Journal of Statistical Software* 106 (1): 1–31. <https://doi.org/10.18637/jss.v106.i01>.
- Therneau, Terry, and Beth Atkinson. 2025. *Rpart: Recursive Partitioning and Regression Trees*. <https://github.com/bethatkinson/rpart>.
- Tillé, Yves, and Alina Matei. 2025. *Sampling: Survey Sampling*. <https://doi.org/10.32614/CRAN.package.sampling>.
- van der Maaten, L. J. P. 2014. “Accelerating t-SNE Using Tree-Based Algorithms.” *Journal of Machine Learning Research* 15: 3221–45.
- van der Maaten, L. J. P., and G. E. Hinton. 2008. “Visualizing High-Dimensional Data Using t-SNE.” *Journal of Machine Learning Research* 9: 2579–2605.
- Venables, W. N., and B. D. Ripley. 2002a. *Modern Applied Statistics with s*. Fourth. New York: Springer. <https://www.stats.ox.ac.uk/pub/MASS4/>.
- . 2002b. *Modern Applied Statistics with s*. Fourth. New York: Springer. <https://www.stats.ox.ac.uk/pub/MASS4/>.
- Venables, W. N., D. M. Smith, and the R Core Team. 2021. *An Introduction to R*.
- Weihs, Claus, Uwe Ligges, Karsten Luebke, and Nils Raabe. 2005. “klaR Analyzing German Business Cycles.” In *Data Analysis and Decision Support*, edited by D. Baier, R. Decker, and L. Schmidt-Thieme, 335–43. Berlin: Springer-Verlag.
- Wickham, Hadley. 2016. *Ggplot2: Elegant Graphics for Data Analysis*. Springer-Verlag New York. <https://ggplot2.tidyverse.org>.
- . 2023a. *Forcats: Tools for Working with Categorical Variables (Factors)*. <https://forcats.tidyverse.org/>.
- . 2023b. *Tidyverse: Easily Install and Load the Tidyverse*. <https://tidyverse.tidyverse.org>.
- . 2025. *Stringr: Simple, Consistent Wrappers for Common String Operations*. <https://stringr.tidyverse.org>.

- Wickham, Hadley, Mara Averick, Jennifer Bryan, Winston Chang, Lucy D'Agostino McGowan, Romain François, Garrett Grolmund, et al. 2019. "Welcome to the tidyverse." *Journal of Open Source Software* 4 (43): 1686. <https://doi.org/10.21105/joss.01686>.
- Wickham, Hadley, Mine Çetinkaya-Rundel, and Garrett Grolmund. 2023. *R for Data Science: Import, Tidy, Transform, Visualize, and Model Data*. 2nd ed. O'Reilly Media, Inc. <https://r4ds.hadley.nz/>.
- Wickham, Hadley, Winston Chang, Lionel Henry, Thomas Lin Pedersen, Kohske Takahashi, Claus Wilke, Kara Woo, Hiroaki Yutani, Dewey Dunnington, and Teun van den Brand. 2025. *Ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. <https://ggplot2.tidyverse.org>.
- Wickham, Hadley, Romain François, Lionel Henry, Kirill Müller, and Davis Vaughan. 2023. *Dplyr: A Grammar of Data Manipulation*. <https://dplyr.tidyverse.org>.
- Wickham, Hadley, and Lionel Henry. 2025. *Purrr: Functional Programming Tools*. <https://purrr.tidyverse.org/>.
- Wickham, Hadley, Jim Hester, and Jennifer Bryan. 2024. *Readr: Read Rectangular Text Data*. <https://readr.tidyverse.org>.
- Wickham, Hadley, Thomas Lin Pedersen, and Dana Seidel. 2025. *Scales: Scale Functions for Visualization*. <https://scales.r-lib.org>.
- Wickham, Hadley, Davis Vaughan, and Maximilian Girlich. 2024. *Tidyr: Tidy Messy Data*. <https://tidyr.tidyverse.org>.
- Wilkinson, Leland. 2005. *The Grammar of Graphics (Statistics and Computing)*. Berlin, Heidelberg: Springer-Verlag. <https://doi.org/10.1007/0-387-28695-0>.
- Witten, Ian H., and Eibe Frank. 2005. *Data Mining: Practical Machine Learning Tools and Techniques*. 2nd ed. San Francisco: Morgan Kaufmann.
- Yu, Guangchuang. 2025. *Scatterpie: Scatter Pie Plot*. <https://doi.org/10.32614/CRAN.package.scatterpie>.
- Zaki, Mohammed J. 2000. "Sequence Mining in Categorical Domains: Incorporating Constraints." In *Proceedings of the Ninth International Conference on Information and Knowledge Management*, 422–29. CIKM '00. New York, NY, USA: Association for Computing Machinery. <https://doi.org/10.1145/354756.354849>.
- Zeileis, Achim, Torsten Hothorn, and Kurt Hornik. 2008. "Model-Based Recursive Partitioning." *Journal of Computational and Graphical Statistics* 17 (2): 492–514. <https://doi.org/10.1198/106186008X319331>.