dd

- ALL NOTES
- STATS 101
- DATA PREPROCESSING

CHEATSHEET - DATA SCIENCE - TOO LONG; DIDN'T DO

tl;dd: common data science concepts and code

Why?: - Adding AI is easy - Deeply understanding AI is hard - These summarise: - data science - machine learning - big data - Artificial Intelligence - Chatbots and - Voice (think Alexa and Google Home)

Note: This is a work in progress. It's literally currently footnotes.

Learning Objectives Assessed:

The final examination assesses the following Course Learning Objectives:

Critically reflect upon different data sources, types, formats and structures. Apply data integration techniques to import and combine different sources of data. Apply different data manipulation techniques to recode, filter, select, split, aggregate, and reshape the data into a format suitable for statistical analysis. Justify data by detecting and handling missing values, outliers, inconsistencies and errors. Demonstrate practical experience by having been exposed to real data problems. Effectively use leading open source software for reproducible, automated data preprocessing.

DATA PREPROCESSING

---

0 Useful Libraries

```
library(dplyr)
library(readr)
library(tidyr)
library(knitr)
```

1. Weird Syntax

`<-` assigns, like `=` normally does `%>%` piping is the same as the . operator in OO languages

Inspect functions

```
str(something)
class(something)
typeof()
dim() # get dimensions
attributes()
length()
# or
something %>% class()
# using pipes, which is like .operator
```

2.

VECTORS

```
vector_name <- c(1:5) #
```

FACTORS

```
vector_fact2 <- factor(
  c("very low", "low", "medium", "high", "very high"),
  levels = c("very low", "low", "medium", "high", "very high")
    )
```

LISTS

```
vector_list <- list(
  vector_int, vector_dbl, vector_char, vector_fact, vector_fact2, vector_fact3)

# name elements in a list
names(vect_list2) <- c ("comp1", "comp2", "comp3", "comp4", "comp5", "comp6", "comp7")
```

MATRIX

```
mat1 <- matrix(seq(0,36,by=2), nrow = 5, ncol = 4)
```

DATA.FRAME

```
OurDataframe <- data.frame(row, col)
```

DATA.TABLE (NOT COVERED)

TIBBLE (NOT COVERED)

Combining

```
vector_combined <- c(vector_int, vector_fact3)

# Appending
vector_list2 <- append(vector_list, list(states))

# Bind rows
bindRows <- rbind(vector_int, vector_fact3)
# Bind columns
bindColumns <- cbind(vector_int, vector_fact3)

# cbind data.frames requires stringsAsFactors set
df <- cbind(OurDataframe, vect_fact3, stringsAsFactors = TRUE)
df
```

Add column and row names

```
colnames(df3) <- c("numbers", "colours", "scale")
rownames(df3) <- c("r1", "r2", "r3", "r4", "r5")
```

Selecting / subsetting

df[4:5,] # subset a dataframe by row numbers

df[,(1,3)] # subset a dataframe by col numbers

df$colName[3] # Select the third element of dataframe from the column named colName

---

Saving

```
library(xlsx)

write.xlsx(df, file = "cars.xlsx")

multiple_df <- createWorkbook()

# Create worksheets within workbooks
cars_df <- createSheet(wb = multiple_df, sheetName = "Cars")
iris_df <- createSheet(wb = multiple_df, sheetName = "Iris")

# Add dataframes to worksheets
```

```
addDataFrame(x = mtcars, sheet = car_df)
addDataFrame(x = iris, sheet = iris_df)

saveWorkbook(multiple_df, file = "combined.xlsx")

save(foo, bar, file = "foobar.RData")

save.image()

saveRDS(input, "something.rds")
output <- readRDS("something.rds")
identical(input, output)
# [1] TRUE
```

3. Types of variables

TODO: summarise Nominal variable: They have a scale in which the numbers or letters assigned to objects serve as labels for identification or classification. Examples of this variable include binary variables (e.g., yes/no, male/female) and multinomial variables (e.g. religious affiliation, eye colour, ethnicity, suburb).

Ordinal variable: They have a scale that arranges objects or alternatives according to their ranking. Examples include the exam grades (i.e., HD, DI, Credit, Pass, Fail etc.) and the disease severity (i.e., severe, moderate, mild).

The second type of variable is called the quantitative variable. These variables are the numerical data that we can either measure or count. The quantitative variables can be either discrete or continuous.

Continuous quantitative variable: They arise from a measurement process. Continuous variables are measured on a continuum or scale. They can have almost any numeric value and can be meaningfully subdivided into finer and finer increments, depending upon the precision of the measurement system. For example: time, temperature, wind speed may be considered as continuous quantitative variables.

Discrete quantitative variable: They arise from a counting process. Examples include the number of text messages you sent this past week and the number of faults in a manufacturing process.

Type
logical
double
integer
character
factor
vector

factor? vectors

Data structure

Dimensions	Homogeneous		Heterogeneous	
1	Atomic vectors	one dimensional array of same type	List	combined types of elements
2	Matrix	must also have same length()	data.frame	

Dimensions	Homogeneous	Heterogeneous
n	Array	-

nrow() ncol()

These 3 print: Cat print Noquote

Filter works on columns Select works on single instance variables

Quote

Tidy and manipulate data pt2

Functions

`Gather()` `spread()` `separate()` `unite()`

	What it does	Eg
Columns from `tidyr::`	Reshape a dataframe	
`gather()`	Wide data -> longer	`dataframe %>% gather(Year, newColumnName, 2:4)`
`spread()`	long data -> wider	`dataframe %>% spread(Year, ColumnWithValuesForNewColumn)`
`separate()`	single column -> multiple columns	`dataframe %>% separate(date, c("year", "month", "day"), sep = "-")`
`unite()`	multiple columns -> single	`unite(date, year, month, day, sep = "-")`
Get specific values		
`filter()`	pick observations based on values	
`>, >=, !=, ==, etc`	comparison operators	
`arrange()`	reorder rows	
	`desc()`	reorder rows in descending order
`select()`	select variables	
	- `starts_with()`	select variables based on patterns
	- `ends_with()`	
	- `contains()`	
**Create new variables **		
`mutate()`	create new variables	
`transmute()`	create new variables	

	What it does	Eg
`summarise()`	summarise data	
`group_by()`	group based on categorical values	
`%>%`	piping operator from `magrittr::`	

5 Scan: Missing Values

Identifying Missing Data

```
df <- data.frame(col1 = c(1:3, NA),
                 col2 = c("this", NA,"is", "text"),
                 col3 = c(TRUE, FALSE, TRUE, TRUE),
                 col4 = c(2.5, 4.2, 3.2, NA),
                 stringsAsFactors = FALSE)

# identify NAs in full data frame

is.na(df)
##       col1  col2  col3  col4
## [1,] FALSE FALSE FALSE FALSE
## [2,] FALSE  TRUE FALSE FALSE
## [3,] FALSE FALSE FALSE FALSE
## [4,]  TRUE FALSE FALSE  TRUE
# identify NAs in specific data frame column

# identify location of NAs in vector

which(is.na(x))
## [1] 5 8
# identify count of NAs in data frame

sum(is.na(df))
## [1] 3
More convenient way to compute the total missing values in each column is to use colSums():

colSums(is.na(df))
## col1 col2 col3 col4
##    1    1    0    1
```

Recode Missing Data

```
# create vector with missing data

x <- c(1:4, NA, 6:7, NA)
x
## [1]  1  2  3  4 NA  6  7 NA
# recode missing values with the mean (also see "Missing Value Imputation Techniques" section)

x[is.na(x)] <- mean(x, na.rm = TRUE)
```

```
x
## [1] 1.000000 2.000000 3.000000 4.000000 3.833333 6.000000 7.000000 3.833333
```

Similarly, if missing values are represented by another value (i.e. ..) we can simply subset the data for the elements that contain that value and then assign a desired value to those elements. Remember that population_NA data frame has missing values represented by ".." in the X.2016 column. Now let's change ".." values to NA's.

```
# population_NA data frame has missing values represented by ".." in the X.2016 column.

population_NA$X.2016
##  [1] "3.32   " "3.99   " ".."     " ".."     " ..."     " "46.11  " "62.41  "
##  [8] ".."     " "504.01 " "510.77 "
# Note the white spaces after ..'s and  change  ".."     " values to NAs


population_NA[population_NA == ".."     " ] <- NA

population_NA
##                                Region X.2013 X.2014 X.2015  X.2016
## 1  ISL                                  3.21   3.25   3.28 3.32
## 2  CAN                                  3.87   3.91   3.94 3.99
## 3  RUS                                  7.83   7.85   7.87    <NA>
## 4  COL                                 41.27  41.74     NA    <NA>
## 5  ZAF                                 43.53  44.22     NA    <NA>
## 6  LTU                                 47.42  46.96  46.63 46.11
## 7  MEX                                 60.43  61.10  61.76 62.41
## 8  IND                                394.85     NA     NA    <NA>
## 9  NLD                                497.64 499.59 501.68 504.01
## 10 KOR                                504.92 506.97 508.91 510.77
```

If we want to recode missing values in a single data frame variable, we can subset for the missing value in that specific variable of interest and then assign it the replacement value. For example, in the following example, we will recode the missing value in col4 with the mean value of col4.

```
# data frame with missing data

df <- data.frame(col1 = c(1:3, NA),
                 col2 = c("this", NA,"is", "text"),
                 col3 = c(TRUE, FALSE, TRUE, TRUE),
                 col4 = c(2.5, 4.2, 3.2, NA),
                 stringsAsFactors = FALSE)

# recode the missing value in col4 with the mean value of col4

df$col4[is.na(df$col4)] <- mean(df$col4, na.rm = TRUE)

df
##   col1 col2  col3 col4
## 1    1 this  TRUE  2.5
## 2    2 <NA> FALSE  4.2
## 3    3   is  TRUE  3.2
## 4   NA text  TRUE  3.3
```

Excluding Missing Data

A common method of handling missing values is simply to omit the records or fields with missing values from the analysis. However, this may be dangerous, since the pattern of missing values may in fact be systematic, and simply deleting records with missing values would lead to a biased subset of the data. Some authors recommend that if the amount of missing data is very small relatively to the size of the data set (up to 5%), then leaving out the few values with missing features would be the best strategy in order not to bias the analysis. When this is the case, we can exclude missing values in a couple different ways.

If we want to exclude missing values from mathematical operations, we can use the na.rm = TRUE argument. If you do not exclude these values, most functions will return an NA. Here are some examples:

```
# create a vector with missing values

x <- c(1:4, NA, 6:7, NA)

# including NA values will produce an NA output when used with mathematical operations

mean(x)
## [1] NA
# excluding NA values will calculate the mathematical operation for all non-missing values

mean(x, na.rm = TRUE)
## [1] 3.833333
```

We may also want to subset our data to obtain complete observations (those observations in our data that contain no missing data). We can do this a few different ways.

```
# data frame with missing values

df <- data.frame(col1 = c(1:3, NA),
                 col2 = c("this", NA,"is", "text"),
                 col3 = c(TRUE, FALSE, TRUE, TRUE),
                 col4 = c(2.5, 4.2, 3.2, NA),
                 stringsAsFactors = FALSE)

df
##   col1 col2  col3 col4
## 1    1 this  TRUE  2.5
## 2    2 <NA> FALSE  4.2
## 3    3   is  TRUE  3.2
## 4   NA text  TRUE   NA
```

First, to find complete cases we can leverage the complete.cases() function which returns a logical vector identifying rows which are complete cases. So in the following case rows 1 and 3 are complete cases. We can use this information to subset our data frame which will return the rows which complete.cases() found to be TRUE.

```
complete.cases(df)
## [1]  TRUE FALSE  TRUE FALSE
# subset data frame with complete.cases to get only complete cases

df[complete.cases(df), ]
##   col1 col2 col3 col4
## 1    1 this TRUE  2.5
## 3    3   is TRUE  3.2
# or subset with `!` operator to get incomplete cases

df[!complete.cases(df), ]
##   col1 col2  col3 col4
## 2    2 <NA> FALSE  4.2
## 4   NA text  TRUE   NA
```

A shorthand alternative approach is to simply use na.omit() to omit all rows containing missing values.

```
# or use na.omit() to get same as above

na.omit(df)
##   col1 col2 col3 col4
## 1    1 this TRUE  2.5
## 3    3   is TRUE  3.2
```

However, it seems like a waste to omit the information in all the other fields just because one field value is missing. Therefore, data analysts should carefully approach to excluding missing values especially when the amount of missing data is very large.

Another recommended approach is to replace the missing value with a value substituted according to various criteria. These approaches will be given in the next section.

Basic Missing Value Imputation Techniques

Imputation is the process of estimating or deriving values for fields where data is missing. There is a vast body of literature on imputation methods and it goes beyond the scope of this course to discuss all of them. In this section I will provide basic missing value imputation techniques only. Replace the missing value(s) with some constant, specified by the analyst In some cases, a missing value can be determined because the observed values combined with their constraints force a unique solution. As an example, consider the following data frame listing the costs for staff, cleaning, housing and the total total for three months.

```
df <- data.frame(month = c(1:3),
                 staff = c(15000 , 20000, 23000),
                 cleaning = c(100, NA, 500),
                 housing = c(300, 200, NA),
                 total = c(NA, 20500, 24000)
                 )

df
##   month staff cleaning housing total
## 1     1 15000      100     300    NA
## 2     2 20000       NA     200 20500
## 3     3 23000      500      NA 24000
```

Now, assume that we have the following rules for the calculation of total cost: staff + cleaning + housing = total and all costs > 0. Therefore, if one of the variables is missing we can clearly derive the missing values by solving the rule. For this example, first month's total cost can be found as 15000 + 100 + 300 = 15400. Other missing values can be found in a similar way. The deducorrect and validate packages have a number of functions available that can impute (and correct) the values according to the given rules automatically for a given data frame.

```
install.packages("deductive")

install.packages("validate")

library(deductive)
library(validate)
# Define the rules as an validator expression

Rules <- validator( staff + cleaning + housing == total,
                            staff >= 0,
                            housing >= 0,
                            cleaning >= 0
             )
## Found more than one class "rule" in cache; using the first, from namespace 'cli'
## Also defined by 'validate'
# Use impute_lr function

imputed_df <- impute_lr(df,Rules)

imputed_df
##   month staff cleaning housing total
## 1     1 15000      100     300 15400
## 2     2 20000      300     200 20500
## 3     3 23000      500     500 24000
```

The deducorrect package together with validate provide a collection of powerful methods for automated data cleaning and imputing. For more information on these packages please refer to "Correction of Obvious Inconsistencies and Errors" section of the module notes and the deducorrect package manual and validate package manual. Replace the missing value(s) with the mean, median or mode Replacing the missing value with the mean, median (for numerical variables) or the mode (for

categorical variables) is a crude way of treating missing values. The Hmisc package has a convenient wrapper function allowing you to specify what function is used to compute imputed values from the non-missing. Consider the following data frame with missing values:

```
x <- data.frame( no = c(1:6),
                 x1 = c(15000 , 20000, 23000, NA, 18000, 21000),
                 x2 = c(4, NA, 4, 5, 7, 8),
                 x3 = factor(c(NA, "False", "False", "False", "True", "True"))
                 )
x
##   no    x1 x2    x3
## 1  1 15000  4  <NA>
## 2  2 20000 NA False
## 3  3 23000  4 False
## 4  4    NA  5 False
## 5  5 18000  7  True
## 6  6 21000  8  True
```

For this data frame, imputation of the mean, median and mode can be done using Hmisc package as follows:

```
install.packages("Hmisc")
library(Hmisc)

# mean imputation (for numerical variables)

x1 <- impute(x$x1, fun = mean)

x1
##      1      2      3      4      5      6
##  15000  20000  23000 19400*  18000  21000
# median imputation (for numerical variables)

x2 <- impute(x$x2, fun = median)

x2
##  1  2  3  4  5  6
##  4 5*  4  5  7  8
# mode imputation (for categorical/factor variables)

x3 <- impute(x$x3, fun= mode)

x3
##      1      2      3      4      5      6
## False*  False  False  False   True   True
```

An nice feature of the impute function is that the resulting vector remembers what values were imputed. This information may be requested with is.imputed function as in the example below.

```
# check which values are imputed

is.imputed(x1)
## [1] FALSE FALSE FALSE  TRUE FALSE FALSE
is.imputed(x2)
## [1] FALSE  TRUE FALSE FALSE FALSE FALSE
is.imputed(x3)
## [1]  TRUE FALSE FALSE FALSE FALSE FALSE
```

More Complex Approaches to Missing Value Imputation

Another strategy is to use predictive models to impute the missing data. There are many different predictive models and algorithms to predict and impute the missing values. Regression analysis, multiple imputation methods, random forests, k nearest neighbours, last observation carried forward / next observation carried backward, etc. are only some of these techniques. In R, there are many different packages (e.g., mice, missForest, impute etc.) that can be used to predict and impute the missing data.

For the detailed information on the missing value imputation please refer to the “Statistical analysis with missing data (Little and Rubin (2014))” for the theory behind the missing value mechanism and analysis. For multiple imputation techniques and case studies using R, please refer to “Flexible imputation of missing data (Van Buuren (2012))”.

Special values In addition to missing values, there are a few special values that are used in R. These are -Inf, Inf and NaN. If a computation results in a number that is too big, R will return Inf (meaning positive infinity) for a positive number and -Inf for a negative number (meaning negative infinity). Here are some examples: ```r 3 ^ 1024

---

[1] Inf

-3 ^ 1024

---

[1] -Inf

This is also the value returned when you divide by 0: 12 / 0

---

[1] Inf

```
Sometimes, a computation will produce a result that makes little sense. In these cases, R will often return NaN (meaning “not a number”):
```r
Inf - Inf
## [1] NaN
0/0
## [1] NaN
```

Identifying Special Values

Calculations involving special values often result in special values, thus it is important to handle special values prior to analysis. The is.finite, is.infinite, or is.nan functions will generate logical values (TRUE or FALSE) and they can be used to identify the special values in a data set.

```r
# create a vector with special values
m <- c( 2, 0/0, NA, 1/0, -Inf, Inf, (Inf*2) )
m
## [1]    2  NaN   NA  Inf -Inf  Inf  Inf
# check finite values

is.finite(m)
## [1]  TRUE FALSE FALSE FALSE FALSE FALSE FALSE
# check infinite values

is.infinite(m)
```
```

```
## [1] FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE
# check not a number values

is.nan(m)
## [1] FALSE  TRUE FALSE FALSE FALSE FALSE FALSE
# create a data frame containing special values

df <- data.frame(col1 = c( 2, 0/0, NA, 1/0, -Inf, Inf),
                 col2 = c( NA, Inf/0, 2/0, NaN, -Inf, 4)
                )

df
##   col1 col2
## 1    2   NA
## 2  NaN  Inf
## 3   NA  Inf
## 4  Inf  NaN
## 5 -Inf -Inf
## 6  Inf    4
is.infinite(df)

# Error in is.infinite(df) : default method not implemented for type 'list'
```

These functions accept vectorial input, this is why you will receive an error when you try to use it with a data frame. Hopefully, we can write a simple function that may be used to check every numerical column in a data frame for infinite values or NA's.

Check inputs whether they are not finite or NA using a function called is.special

```
is.special <- function(x){
if (is.numeric(x)) !is.finite(x) else is.na(x)
}


is.special <- function(x){
if (is.numeric(x)) !is.finite(x)
}

# apply this function to the data frame.

sapply(df, is.special)
##       col1  col2
## [1,] FALSE  TRUE
## [2,]  TRUE  TRUE
## [3,]  TRUE  TRUE
## [4,]  TRUE  TRUE
## [5,]  TRUE  TRUE
## [6,]  TRUE FALSE
```

Here, the is.special function is applied to each column of df using sapply. is.special checks the data frame for numerical special values if the type is numeric, otherwise it only checks for NA.

Checking for Obvious Inconsistencies or Errors

An obvious inconsistency occurs when a data record contains a value or combination of values that cannot correspond to a real-world situation. For example, a person's age cannot be negative, a man cannot be pregnant and an under-aged person cannot possess a drivers license. Such knowledge can be expressed as rules or constraints. In data preprocessing literature

these rules are referred to as edit rules or edits, in short. Checking for obvious inconsistencies can be done straightforwardly in R using logical indices. For example, to check which elements of x obey the rule: "x must be non negative" one can simply use the following.

```
# create a vector called x

x <- c( 0, -2, 1, 5)

# check the non negative elements

x_nonnegative <- (x >= 0)

x_nonnegative
## [1]  TRUE FALSE  TRUE  TRUE
```

However, as the number of variables increases, the number of rules may increase and it may be a good idea to manage the rules separate from the data. For such cases, the editrules package allows us to define rules on categorical, numerical or mixed-type data sets which each record must obey. Furthermore, editrules can check which rules are obeyed or not and allows one to find the minimal set of variables to adapt so that all rules can be obeyed. This package also implements a number of basic rule operations allowing users to test rule sets for contradictions and certain redundancies.

To illustrate I will use a small data set (datawitherrors.csv) given below:

```
datawitherrors <- read.csv("data/datawitherrors.csv")

datawitherrors
##   no age agegroup height  status yearsmarried
## 1  1  21    adult    178  single           -1
## 2  2   2    child    147 married            0
## 3  3  18    adult    167 married           20
## 4  4 221  elderly    154 widowed            2
## 5  5  34    child   -174 married            3
```

As you noticed, there are many inconsistencies/errors in this small data set (i.e., age = 221, height = -174 , years married = -1, etc.) . To begin with a simple case, let's define a restriction on the age variable using editset functions. In order to use editset functions, we need to install and load the editrules package.

```
install.packages("editrules")
library(editrules)
In the first rule, we will define the restriction on the age variable as $ 0 age 150 $ using editset function.

(Rule1 <- editset(c("age >= 0", "age <= 150")))
##
## Edit set:
## num1 : 0 <= age
## num2 : age <= 150
```

The editset function parses the textual rules and stores them in an editset object. Each rule is assigned a name according to it's type (numeric, categorical, or mixed) and a number. The data set can be checked against these rules using the violatedEdits function.

```
violatedEdits(Rule1, datawitherrors)
##       edit
## record  num1  num2
##      1 FALSE FALSE
##      2 FALSE FALSE
##      3 FALSE FALSE
##      4 FALSE  TRUE
##      5 FALSE FALSE
```

violatedEdits returns a logical array indicating for each row of the data, which rules are violated. From the output, it can be understood that the 4th record violates the second rule (age <= 150). One can also read rules, directly from a text file using the editfile function. As an example consider the contents of the following text file (also available here):

```
# numerical rules
age >= 0
height > 0
age <= 150
age > yearsmarried

# categorical rules
status %in% c("married","single","widowed")
agegroup %in% c("child","adult","elderly")
if ( status == "married" ) agegroup %in% c("adult","elderly")

# mixed rules
if ( status %in% c("married","widowed")) age - yearsmarried >= 17
if ( age < 18 ) agegroup == "child"
if ( age >= 18 && age <65 ) agegroup == "adult"
if ( age >= 65 ) agegroup == "elderly"
```

These rules are numerical, categorical and mixed (both data types). Comments are written behind the usual # character. The rule set can be read using editfile function as follows:

```
Rules <- editfile("data/editrules.txt", type = "all")

Rules
##
## Data model:
## dat6 : agegroup %in% c('adult', 'child', 'elderly')
## dat7 : status %in% c('married', 'single', 'widowed')
##
## Edit set:
## num1 : 0 <= age
## num2 : 0 < height
## num3 : age <= 150
## num4 : yearsmarried < age
## cat5 : if( agegroup == 'child' ) status != 'married'
## mix6 : if( age < yearsmarried + 17 ) !( status %in% c('married', 'widowed') )
## mix7 : if( age < 18 ) !( agegroup %in% c('adult', 'elderly') )
## mix8 : if( 18 <= age & age < 65 ) !( agegroup %in% c('child', 'elderly') )
## mix9 : if( 65 <= age ) !( agegroup %in% c('adult', 'child') )
violatedEdits(Rules, datawitherrors)
##       edit
## record  num1  num2  num3  num4  dat6  dat7  cat5  mix6  mix7  mix8  mix9
##      1 FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##      2 FALSE FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE FALSE FALSE FALSE
##      3 FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE
##      4 FALSE FALSE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
##      5 FALSE  TRUE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE  TRUE FALSE
```

As the number of rules grows, looking at the full array produced by violatedEdits becomes complicated. For this reason, editrules offers methods to summarise or visualise the result as follows:

```
Violated <- violatedEdits(Rules, datawitherrors)

# summary of violated rules

summary(Violated)
## Edit violations, 5 observations, 0 completely missing (0%):
##
##  editname freq rel
##      cat5    2 40%
##      mix6    2 40%
##      num2    1 20%
##      num3    1 20%
##      num4    1 20%
##      mix8    1 20%
##
## Edit violations per record:
##
##  errors freq rel
##       0    1 20%
```

```
##       1    1 20%
##       2    2 40%
##       3    1 20%
# plot of violated rules

plot(Violated)
```

Using the functions available in editrules package, users can detect the obvious errors and/or inconsistencies in the data set, and define edit rules to identify the inconsistent records. Moreover, analysts may need to correct the obvious errors and/or inconsistencies in a data set. In the next section, I will introduce the deducorrect package (actually it is a former version of deductive package) functions to correct the obvious errors and inconsistencies.

Correction of Obvious Inconsistencies or Errors When the data you are analysing is generated by people rather than machines or measurement devices, certain typical human-generated errors are likely to occur. Given that data has to obey certain edit rules, the occurrence of such errors can sometimes be detected from raw data with (almost) certainty. Examples of errors that can be detected are typing errors in numbers, rounding errors in numbers, and sign errors.

The deducorrect package has a number of functions available that can correct such errors. Consider the following data frame (datawitherrors2.csv):

```
datawitherrors2 <- read.csv("data/datawitherrors2.csv")

datawitherrors2
##   no height unit
## 1  1 178.00   cm
## 2  2   1.47    m
## 3  3  70.00 inch
## 4  4 154.00   cm
## 5  5   5.92   ft
```

The task here is to standardise the lengths and express all of them in meters. The deducorrect package can correct this inconsistency using correctionRules function. For example, to perform the above task, one first specifies a file with correction rules as follows (also available here).

```
1 # convert centimeters
2 if ( unit == “cm” ){
3 height <- height/100
4 }
5 # convert inches
6 if (unit == “inch” ){
7 height <- height/39.37
8 }
9 # convert feet
10 if (unit == “ft” ){
11 height <- height/3.28
12 }
13 # set all units to meter
14 unit <- “m”
```

With correctionRules we can read these rules from the txt file using .file argument. ```r install.packages("deducorrect")

library(deducorrect)

---

read rules from txt file using validate

Rules2 <- correctionRules("data/editrules2.txt")

Rules2

Object of class 'correctionRules'

1-------

if (unit == "cm") height <- height/100

2-------

if (unit == "inch") height <- height/39.37

3-------

if (unit == "ft") height <- height/3.28

4-------

unit <- "m"

Now, we can apply them to the data frame and obtain a log of all actual changes as follows:

```r
cor <- correctWithRules(Rules2, datawitherrors2)

cor
## $corrected
##   no   height unit
## 1  1 1.780000    m
## 2  2 1.470000    m
## 3  3 1.778004    m
## 4  4 1.540000    m
## 5  5 1.804878    m
##
## $corrections
##   row variable  old      new                                      how
## 1   1   height  178     1.78     if (unit == "cm") height <- height/100
## 2   1     unit   cm        m                                unit <- "m"
## 3   3   height   70 1.778004 if (unit == "inch") height <- height/39.37
```

```
## 4   3     unit inch        m                              unit <- "m"
## 5   4   height  154     1.54     if (unit == "cm") height <- height/100
## 6   4     unit   cm        m                              unit <- "m"
## 7   5   height 5.92 1.804878    if (unit == "ft") height <- height/3.28
## 8   5     unit   ft        m                              unit <- "m"
```

The returned value, cor$corrected will give a list containing the corrected data as follows: ```r cor$corrected

---

no height unit

---

1 1 1.780000 m

---

2 2 1.470000 m

---

3 3 1.778004 m

---

4 4 1.540000 m

---

5 5 1.804878 m

---

7 - Transform TODO

DATA TRANSFORMATION, STANDARDISATION, AND REDUCTION

DR. ANIL DOLGUN

LAST UPDATED: 15 JUNE, 2018

TODO: add histograms TODO: add math images to box cox

tl;dr

- To reduce right skewness in the distribution, taking roots or logarithms or reciprocals work well.
- To reduce left skewness, taking squares or cubes or higher powers work well.

TODO: The most useful data transformation functions are: - the logarithm (`base 10` and `base e`) reciprocal - cube root - square root - square ## Learning Objectives - Apply well-known transformations - Apply normalisation and standardisation - Learn commonly used approaches for data discretisation - Apply different variable selection, ranking and feature extraction techniques for data reduction

Data transformation is perhaps the most important step in the data preprocessing for the development and deployment of statistical analysis and machine learning models. In almost all statistical and machine learning analyses, it is necessary to perform some data transformations (i.e., data transformation, scaling, centering, standardisation and normalisation) on the raw (but tidy and clean!) data before it can be used for modelling.

In this module, we focus on the most common and useful data transformations that can be easily implemented in R. The number of possible data transformations is indeed large and the selection of proper and useful transformations would definitely depend on the context of the data and the type of the statistical analysis. This is why specific types of analyses may require specific types of transformations. As you move forward in your master's program, you will learn the details of these specific transformations used in different data analysis subjects (i.e., Introduction to Statistics, Machine Learning, Analysis of Categorical Data, Time Series Analysis, Forecasting and Applied Bayesian Analysis courses, etc.). Our aim in this course is not to give technical details of these transformations, but you may refer to the "Additional Resources and Further Reading" section to find out more on the topic.

Data Transformation

Data transformation is often a requisite to further proceed with statistical analysis. Below are the situations where we might need transformations:

WHY?

- **Understanding** - We may need to change the **scale** of a variable or **standardise the values** of a variable for better understanding.
- Get **Linear relationships** - We may need to transform complex non-linear relationships into linear relationships. Transformation helps us to convert a non-linear relation into linear one.

WHY? NORMAL DISTRIBUTION

- **Normal distribution** In statistical inference, symmetric (normal) distribution is preferred over skewed distribution.
- For **Statistical Analysis** Also, some statistical analysis techniques (i.e., `parametric tests`, `linear regression`, etc) requires normal distribution of variables and homogeneity of variances.
- Reduce the **skewness**
- Reduce **heterogeneity of variances**

So, whenever we have a skewed distribution and/or heterogeneous of variances, we can use transformations which can reduce skewness and/or heterogeneity of variances.

DATA TRANSFORMATION VIA MATHEMATICAL OPERATIONS

TODO: turn this into a summary card with links

Often, mathematical operations are applied to the data to achieve `normality` and/or `variance homogeneity`. Such transformations through mathematical operations like:

- logarithmic (i.e., ln and log),
- square root,
- power transformations
- etc.

can easily be done in R using arithmetic functions.

The most useful transformations in introductory data analysis are:

- the logarithm (`base 10` and `base e`) reciprocal
- cube root
- square root

- square

Log transformation

The Log transformation (`base 10 - log10` or `base e - loge`) compresses high values and spreads low values by expressing the values as orders of magnitude. This transformation is commonly used for reducing right skewness. It can not be applied to zero or negative values directly. In order to apply the log transformation to a zero or negative value, we can add a non-negative constant to all observations and then take the logarithm.

Let's have a look at the hypothetical data on the salaries data (`salary.csv`).

TODO: Add histogram of salary$salary

From the histogram, we observe that salaries have a right-skewed distribution. By applying a logarithmic transformation, the salary distribution would be more symmetrical. We can apply the logarithmic transformation (`base 10`) using the `log10()` function in R as follows:

```
log_salary <- log10(salary$salary)

hist(log_salary)
```

TODO: Add histogram of log_salary

`LOG()` AND `LN()` NATURAL LOGARITHM TRANSFORMATION

Another logarithmic transformation is the natural logarithm (often called ln or $\log_e$). This transformation can be easily done using the `log()` function in R.

For the salaries data, we can also apply the `ln` transformation as follows:

```
ln_salary <- log10(salary$salary)

hist(ln_salary)
```

TODO: Add histogram of ln_salary

As seen from the histograms, the `log10` transformation worked slightly better than the `ln` transformation for this example.

Usually, analysts apply different transformations on the same data and then select the one that works well or is useful.

`sqrt()` - square root transformation

Another transformation is the square root transformation. It is also used for reducing right skewness, and also has the advantage that it can be applied to zero values. In R, the function `sqrt()` will apply the square root transformation. Let's apply the square root transformation and see the effect of this transformation on the salary distribution:

```
sqrt_salary <- sqrt(salary$salary)
hist(sqrt_salary)
```

TODO: Add histogram of sqrt_salary

The square root transformation has reduced the skewness in the salary distribution a bit, however it didn't

`x^2` square transformation

The **square transformation** has a moderate effect on distribution shape and it can be used to reduce left skewness. It spreads out the high values relative to the smaller values. In R, the mathematical operation x^2 would apply the square transformation. To illustrate the square transformation, let's read in `left_skewed1.csv` and have a look at the shape of the distribution using histogram:

```
x1<- read.csv("data/left_skewed1.csv")

x1<-x1[,-1] # dropping the first column

hist(x1)
```

This distribution is left skewed, we can try and see the effect of square transformation on the distribution using the following:

```
xsquare1 <- x1^2

hist(xsquare1)
```

TODO: Add histogram of xsquare1

As seen above, the square transformation was not helpful at all to get a symmetrical distribution for this data set. Now, let's look at another example data left_skewed2.csv having less skewed distribution:

```
x2<- read.csv("data/left_skewed2.csv")

x2<-x2[,-1] # dropping the first column

hist(x2)
```

TODO: Add histogram of x2

Let's apply square transformation to x2 and see the effect:

```
x2square <- x2^2

hist(x2square)
```

TODO: Add histogram of x2square

As seen in the last histogram, square transformation was able to transform the shape of the distribution into a more symmetric one when the distribution is mildly left skewed.

---

reciprocal transformation

The **reciprocal transformation** is a very strong transformation with a drastic effect on the distribution shape. It will compress large values to smaller values. The mathematical operation `1/x` or `x^(-1)` would apply the reciprocal transformation.

Let's apply the reciprocal transformation to the x3 variable in the right_skewed1.csv data and see its drastic effect on the shape of the distribution:

```
x3<- read.csv("data/right_skewed1.csv")

x3<-x3[,-1] # dropping the first column
```

```
hist(x3)

x3recip <- 1/x3

hist(x3recip)
```

TODO: Add histogram of x3recip

As seen above, reciprocal transformation worked very well in this case.

Power transformation

The main transformations mentioned previously, with the exception of the logarithm, are all powers. Therefore, these transformations are also called as power transformations. Here is the list of power transformations:

TODO:

Transformation	Power
reciprocal	square -2
reciprocal	-1
(yields one)	0
cube	root 1/3
square	root 1/2
identity	1
square	2
cube	3
fourth	power 4

There are also some recommendations on useful transformations for specific types of distributions. For example:

tl;dr

- To reduce right skewness in the distribution, taking roots or logarithms or reciprocals work well.
- To reduce left skewness, taking squares or cubes or higher powers work well.

Note that these are general recommendations on mathematical transformations and may not work for every data set. Usually, the best approach is to apply different transformations on the same data and select the one that works best.

Box-Cox Transformation

Normal distribution assumption is very crucial for many statistical hypothesis tests especially for the parametric hypothesis testing, linear regression, time series analysis, etc. The Box-Cox transformation is a type of power transformation to transform non-normal data into a normal distribution. This transformation is named after statisticians George Box and Sir David Cox who

collaborated on a 1964 paper and developed the technique (Box and Cox (1964)). Let y denote the variable at the original scale and y′ the transformed variable. The Box-Cox transformation is defined as: $y′={yλ−1λ,if λ≠0log(y),if λ=0$

If the data include any negative observations, a shifting parameter λ2 can be included in the Box-Cox transformation as given by: $y′={(y+λ2)λ−1λ,if λ≠0log(y+λ2),if λ=0$

As seen in the equations, the λ parameter is very important for applying this transformation. Essentially, finding a good λ parameter satisfying the normality assumption is also a hard task and can be done by a search algorithm or the maximum likelihood estimation. There are many powerful packages that can be used to apply the Box-Cox transformation. Among them the

- `caret`
- `MASS`
- `forecast`
- `geoR`
- `EnvStats`
- `AIS`

packages are only some of them. In this module, I will introduce the forecast package, as this package has very useful functions to find the best `λ` parameter for the Box-Cox transformation.

```
library(forecast)
BoxCox(salary$salary,lambda = "auto")
##   [1] 0.9854516 0.9843098 0.9880776 0.9836914 0.9839247 0.9884059 0.9851253
## and soforth
##  [99] 0.9843304 0.9868072
## attr(,"lambda")
## [1] -0.9999242
```

This function returns the transformed data values. From this output the optimum λ value is found as `-0.9999242`. We can also see the distribution of transformed values using histogram. `hist(boxcox_salary)`

```
x3<- read.csv("data/right_skewed1.csv")

x3<-x3[,-1] # dropping the first column

hist(x3)

boxcox_x3<- BoxCox(x3,lambda = "auto")

hist(boxcox_x3)
```

Transforms skewed distributions into a symmetric distribution.

Data Normalisation

Some statistical analysis methods are sensitive to the scale of the variables and there can be instances found in data sets where values for one variable could range between 1-10 and values for other variable could range from 1-10000000. In scenarios like these, the impact on response variables by the variables having greater numeric range (i.e., 1-10000000), could be more than the one having less numeric range (i.e. 1-10).

Especially for the distance based methods in machine learning, this could in turn impact the prediction accuracy. For such cases, we may need to normalise or scale the values under different variables such that they fall under common range.

There are different normalisation techniques used in machine learning. These are centering using mean, scaling using

- `standard deviation`
- `z-score transformation`
    - (i.e. **centering and scaling** using both `mean` and `standard deviation`)

- `min-max`
- `range`
- `(0-1) transformation`

Mean-centering and scaling

Centering (a.k.a. mean-centering) involves the subtraction of the variable average from the data. Let y denote the variable at the original scale and the $\bar{y}$ is the average. The centered variable y′ is defined as:

TODO: $y' = y - \bar{y}$

If we have more than one variable to center, we can calculate the average value of each variable and then subtract it from the data. This implies that each column will be transformed in such a way that the resulting variable will have a **zero mean**.

`scale()`

We can use simple user-defined functions or built-in functions available in R to center variables. One of the functions to apply mean-centering is the `scale()` function under Base R.

The `scale()` function has the following arguments:

if center =	TRUE	FALSE
	column values - column means	no centering
Scaling = TRUE	Divided by columns `standard deviation`	Divided by `root-mean-squared`
Scaling = FALSE	no scaling	no scaling

- `x:` a numeric object
- `center:` if TRUE, the objects' column means are subtracted from the values in those columns (ignoring NAs); if FALSE, centering is not performed.
- `scale:`
  - if `TRUE`, the centered column values are divided by the column's standard deviation (when center is also TRUE) or
  - divided by the root-mean-square (when center is FALSE). If scale = FALSE, scaling is not performed.

```
df <- data.frame(x1 = c(10, 20, 40, 50, 10),
                 x2 = c(1000, 5000, 3000, 2000, 1500),
                 x3 = c(0.1, 0.12, 0.11, 0.14, 0.16),
                 x4 = c(2.5, 4.2, 3.2, 4.5, 3.8) )
```

`scale()` - Mean Centering => standard deviation

To apply **mean-centering**:

```
center_x <-scale(df, center = TRUE, scale = FALSE)

center_x
##       x1    x2     x3    x4
## [1,] -16 -1500 -0.026 -1.14
## [2,]  -6  2500 -0.006  0.56
```

```
## [3,]  14   500 -0.016 -0.44
## [4,]  24  -500  0.014  0.86
## [5,] -16 -1000  0.034  0.16
## attr(,"scaled:center")
##       x1       x2       x3       x4
##   26.000 2500.000    0.126    3.640
```

In the output, the new centered values for each column are given along with the column (variables') averages.

Scaling involves the division of the values to its **standard deviation** (or **root-mean-square** value). Let y denote the variable at the original scale and the SDy is the standard deviation. The scaled variable y' is defined as: `y´=ySDy`

`Scale()` => column root-mean-square values

HOW: WITHOUT CENTERING => ROOT-MEAN-SQUARE

In order to apply just scaling (without centering) to the data frame we can use `center = FALSE` and `scale = TRUE` arguments as follows:

scaled with the **column** `root-mean-square` values

```
  scale(df,
    center = FALSE,
    scale = TRUE)

##           x1        x2        x3        x4
## [1,] 0.29173 0.3113996 0.6997114 0.6027159
## [2,] 0.58346 1.5569979 0.8396537 1.0125628
## [3,] 1.16692 0.9341987 0.7696826 0.7714764
## [4,] 1.45865 0.6227992 0.9795960 1.0848887
## [5,] 0.29173 0.4670994 1.1195383 0.9161282
## attr(,"scaled:scale")
##           x1           x2           x3           x4
##   34.2782730 3211.3081447    0.1429161    4.1478910
```

Note that, when we scale values without centering, the `scale()` function divides the values to the root-mean-square value instead of standard deviation. Therefore, in this output the new scaled variables are actually **scaled with the column root-mean-square values**.

`scale()` by standard deviation without centering (and gets columns standard deviation)

If we want to scale by the standard deviations without centering, we can use the following:

```
scale(
  df,
  center = FALSE,
  scale =
    apply(
      df,
      2,
      sd,
      na.rm = TRUE))
##             x1        x2       x3       x4
## [1,] 0.5504819 0.6324555 4.152274 3.117701
## [2,] 1.1009638 3.1622777 4.982729 5.237738
## [3,] 2.2019275 1.8973666 4.567501 3.990658
## [4,] 2.7524094 1.2649111 5.813184 5.611863
## [5,] 0.5504819 0.9486833 6.643638 4.738906
```

```
## attr(,"scaled:scale")
##           x1           x2           x3           x4
## 1.816590e+01 1.581139e+03 2.408319e-02 8.018728e-01
```

The output above now reports the scaled values (by standard deviation) along with the column standard deviations. Usually, scaling is not used alone, instead, it is used together with mean-centering and then it is called as the `z-score` standardisation.

---

z score standardisation

- transformed data values have a zero mean and one standard deviation.

`z=y−y¯SDy` The mean of observations are first subtracted from each individual data point, then divided by the standard deviation of all points. In the equation below, y denotes the values of observations, $\bar{y}$ and SDy are the sample mean and standard deviation, respectively.

```
scale(df, center = TRUE, scale = TRUE)

##              x1         x2         x3         x4
## [1,] -0.8807710 -0.9486833 -1.0795912 -1.4216719
## [2,] -0.3302891  1.5811388 -0.2491364  0.6983651
## [3,]  0.7706746  0.3162278 -0.6643638 -0.5487155
## [4,]  1.3211565 -0.3162278  0.5813184  1.0724893
## [5,] -0.8807710 -0.6324555  1.4117732  0.1995329
## attr(,"scaled:center")
##       x1       x2       x3       x4
##   26.000 2500.000    0.126    3.640
## attr(,"scaled:scale")
##           x1           x2           x3           x4
## 1.816590e+01 1.581139e+03 2.408319e-02 8.018728e-01
```

Note that we can also use other functions (i.e., `scores()`) from other packages to get the same result.

Min- Max Normalisation (a.k.a. range or (0-1) normalisation)

`y′=y−yminymax−ymin`

- In contrast to z-score standardisation this normalisation can suppress the effect of outliers.

```
minmaxnormalise <- function(x){(x- min(x)) /(max(x)-min(x))}

# `lapply()` one can apply this function to a data frame.
lapply(df, minmaxnormalise)
## $x1
## [1] 0.00 0.25 0.75 1.00 0.00
##
## $x2
## [1] 0.000 1.000 0.500 0.250 0.125
##
## $x3
## [1] 0.0000000 0.3333333 0.1666667 0.6666667 1.0000000
##
## $x4
## [1] 0.00 0.85 0.35 1.00 0.65
If you would like to store the normalised values as a data frame you may also use as.data.frame() function:
as.data.frame(lapply(df, minmaxnormalise))
##     x1    x2        x3   x4
## 1 0.00 0.000 0.0000000 0.00
## 2 0.25 1.000 0.3333333 0.85
## 3 0.75 0.500 0.1666667 0.35
```

```
## 4 1.00 0.250 0.6666667 1.00
## 5 0.00 0.125 1.0000000 0.65
```

Binning (a.k.a. Discretisation)

- Good for outliers
    - by placing them to the first or last category
- to get discrete values as an input or output variable
- i.e., most versions of Naive Bayes and CHAID analysis
- transform numerical variables into categorical counterparts

2 kinds:

- equal-width binning
- equal-frequency binning.

Equal width (distance) binning

variable is divided into n intervals of equal size

ymax and ymin are the maximum and the minimum values in the variable, the width of the intervals will be:

TODO: `w=ymax-yminn`

Thus, we need to define the number of intervals n prior to binning. However, this is not an easy task for the analysts and constitutes one of the disadvantages of this method.

```
library(infotheo)
iris <- read.csv("data/iris.csv")

iris %>%
  filter( Species == "versicolor" ) %>%
  dplyr::select(Sepal.Length) %>%
  head()
##   Sepal.Length
## 1          7.0
## 2          6.4
## 3          6.9
## 4          5.5
## 5          6.5
## 6          5.7
```

Let's apply equal-width binning to the `Sepal.Length` variable.

TODO:

```
ew_binned<-
  discretize(
    versicolor,
    disc = "equalwidth")
versicolor %>%
bind_cols(ew_binned) %>%
head(5)
##    Sepal.Length Sepal.Length1
## 1           7.0             3
## 2           6.4             3
## 3           6.9             3
## 4           5.5             1
## 5           6.5             3
```

```
## 6          5.7            2
## 7          6.3            3
## 8          4.9            1
## 9          6.6            3
## 10         5.2            1
## 11         5.0            1
## 12         5.9            2
## 13         6.0            2
## 14         6.1            2
## 15         5.6            1
```

Equal depth (frequency) binning `disc = "equalfreq"`

`discretize(disc = "equalfreq")`

- divided into n intervals,
- each containing approximately the same number of observations (frequencies).

```
ed_binned<-discretize(versicolor, disc = "equalfreq")
versicolor %>% bind_cols(ed_binned) %>% head(5)
##    Sepal.Length Sepal.Length1
## 1          7.0             3
## 2          6.4             3
## 3          6.9             3
## 4          5.5             1
## 5          6.5             3
```

Data (dimension) reduction

- For too many variables
    - reduces risk of overfitting
    - because machine learning algorithms and regression are sensitive
- feature selection
- feature extraction

`mlr` A quick look to the mlr package

- missing values imputation
- normalising
- centering
- standardising
- transforming
- feature extraction and
- selection)

```
library(mlr)
```

mlr steps:

1. Create a task ->
2. Create a learner ->
3. Fit a model ->
4. Make predictions ->
5. Evaluate the learner

---

1. Create a task

- specifying the type of analysis
- providing the data and response variable
- `makeClassifTask()` = define the task as a classification, similarly
- `makeRegrTask()` = specify the task as a regression analysis.

Eg. To predict the Sepal.Length using other variables.

1. Define the task. Ie. regression

```
iris.task =
  makeRegrTask(
    data = iris,
    target = "Sepal.Length")
```

---

2. `makeLearner()` Create a learners

- Choosing a specific algorithm (learner) which learns from task (or data).

With `makeLearner()` function we can easily specify the learner.

1. Define the learner

e.g generalised linear model (a.k.a classical regression)

```
## Choose a specific algorithm (e.g. generalised linear model - classical regression)
lrn = makeLearner("regr.glm")
```

---

3. Fit the model - `train()` and `test`

Fitting the model = get a random subset of data (i.e. training set).

1. divide the data set into train and test sets.

- training data `sample()` to get train
- testing set `setdiff()` from `dplyr` package:

```
n = nrow(iris)
train.set = sample(n, size = 2/3*n)
test.set = setdiff(1:n, train.set)

## 3) Fit the model
## Train the learner on the task using a random subset of the data as training set
```

```
model = mlr::train(lrn, iris.task, subset = train.set)
```

---

4. `predict()` Make predictions

Apply the model in the test set to predict the **response variable** for new observations.

```
## 4) Make predictions
prediction = predict(model, task = iris.task, subset = test.set)
```

---

5. Evaluate the learner with `performance()`

- Calculate the performance metrics for learners
- e.g. mean misclassification error and accuracy

```
## 5) Evaluate the learner
## Calculate the mean squared error and mean absolute error
performance(prediction, measures = list(mse, mae))
##       mse       mae
## 0.1007054 0.2718763
```

So far, I've introduced the main workflow of mlr package in a general sense. In the next sections we will see the details of how to use these functions for feature selection and feature extraction.

Feature selection

In feature selection, we try to find a subset of the original set of variables, or features which are best representatives of the data. There are different strategies to select features depending on the problem that you are dealing with. The most basic ones are given in the following subsections.

Feature filtering

In feature filtering, redundant features are filtered out and the ones that are most useful or most relevant for the problem are selected. Feature filtering methods include removing features with zero and near zero-variance and removing highly correlated variables (i.e., greater than 0.8). To illustrate feature filtering, we will continue with the iris.task (given above). In the iris data frame, we have 5 features (one of them Sepal.Length is defined as response/target variable).

```
iris.task
## Supervised task: iris
## Type: regr
## Target: Sepal.Length
## Observations: 150
## Features:
##    numerics     factors     ordered functionals
##           4           1           0           0
## Missings: FALSE
## Has weights: FALSE
```

```
## Has blocking: FALSE
## Has coordinates: FALSE
# Print the feature names in the task

getTaskFeatureNames(iris.task)
## [1] "X"            "Sepal.Width"  "Petal.Length" "Petal.Width"
## [5] "Species"
```

To drop specific features, let's say Species, we can use dropFeatures(task, features) function.

```
# Drop the Species and write it into the iris.task1

iris.task1 <-dropFeatures(iris.task, features = "Species")
iris.task1
## Supervised task: iris
## Type: regr
## Target: Sepal.Length
## Observations: 150
## Features:
##    numerics     factors     ordered functionals
##           4           0           0           0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
## Has coordinates: FALSE
# Print the feature names

getTaskFeatureNames(iris.task1)
## [1] "X"            "Sepal.Width"  "Petal.Length" "Petal.Width"
```

To remove/keep the features using a specific rule (i.e., remove/keep features with zero or near zero variance, remove/keep highly correlated features, etc) we can use the filterFeatures function. This function again requires the task and a character string specifying the filter method.

```
filterFeatures(task, method, perc, abs, threshold)
```

```
## Filter out all features with zero variance
filtered.task1 = filterFeatures(iris.task1, method = "variance", threshold = 0)

filtered.task1
## Supervised task: iris
## Type: regr
## Target: Sepal.Length
## Observations: 150
## Features:
##    numerics     factors     ordered functionals
##           4           0           0           0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
## Has coordinates: FALSE
getTaskFeatureNames(filtered.task1)
## [1] "X"            "Sepal.Width"  "Petal.Length" "Petal.Width"
```

From the output, no features have zero variance, that's why the filterFeatures function kept all features.

```
## Filter all features with high correlation (i.e r>0.7)
filtered.task2 = filterFeatures(iris.task1, method = "linear.correlation", threshold = 0.7)

filtered.task2
## Supervised task: iris
## Type: regr
## Target: Sepal.Length
## Observations: 150
## Features:
##    numerics     factors     ordered functionals
```

```
##           3           0           0           0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
## Has coordinates: FALSE
getTaskFeatureNames(filtered.task2)
## [1] "X"            "Petal.Length" "Petal.Width"
```

check manually which features have high correlation with the target variable (`Sepal.Length`) using

> **bivariate correlations**.

```
# Check correlations in the data
cor(iris$Sepal.Length, iris$Petal.Length)
## [1] 0.8717538
```

The correlations indicate that the correlation between Sepal.Length and Petal.Length is r=0.87 and the correlation between Sepal.Length and Petal.Width is r=0.82. Therefore, Sepal.Length and Petal.Width were being the filtered features. More in filter features

---

Feature ranking `filterFeatures()`

- Ranked by an `importance criteria`, ie.
  - chi-square test
  - correlation test
  - entropy based tests
  - random forest
- selecting those which are `above a defined threshold`
- to be kept or removed from the data set

```
## Rank all features using chi square test importance and select the two most important ones

filterFeatures(iris.task1, method = "chi.squared", abs = 2)

## Supervised task: iris
## Type: regr
## Target: Sepal.Length
## Observations: 150
## Features:
##    numerics     factors     ordered functionals
##           2           0           0           0
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
## Has coordinates: FALSE
getTaskFeatureNames(filtered.task3)
## [1] "Petal.Length" "Petal.Width"
```

> Petal.Length and Petal.Width are the two most important features according to the chi-square criteria, so we keep them.

```
## Rank all features using correlation test importance and select the most important one

filterFeatures(iris.task1, method = "linear.correlation", abs = 1)

## Supervised task: iris
## Type: regr
## Target: Sepal.Length
## Observations: 150
## Features:
##    numerics     factors     ordered functionals
##           1           0           0           0
```

```
## Missings: FALSE
## Has weights: FALSE
## Has blocking: FALSE
## Has coordinates: FALSE
getTaskFeatureNames(filtered.task4)
## [1] "Petal.Length"
```

The output suggest that the Petal.Length is the most important feature according to the correlation criteria.

Feature extraction - reduces the dimensions of data

Feature extraction reduces the data in a high dimensional space to a lower dimension space, - Creating a new combinations of attributes,

Feature selection - include and exclude attributes present in the data without changing them. Eg. with PCA

Principal Component Analysis (PCA)

- an unsupervised algorithm that creates `linear combinations of the original features`.
- Orthoganal and uncorrelated
- Ranked by “explained variance”
- Eg. the first principal component (PC1) explains the most variance in the data, and so on.
- Down to 90% a cumulative explained variance of 90%
- Pros: fast and simple to impliment
- Cons: PCA's aren't interpretable

Packages include: `prcomp()` `caret` - any caret functions can also be converted into the mlr functions.

`preProcess()` function as follows:

```
library(caret)
library(mlbench) # to load the sonar data
Note that this data has 208 observations and 60 features.
data(Sonar)

head(Sonar)
##       V1     V2     V3     V4     V5     V6     V7     V8     V9    V10
## 1 0.0200 0.0371 0.0428 0.0207 0.0954 0.0986 0.1539 0.1601 0.3109 0.2111
## 2 0.0453 0.0523 0.0843 0.0689 0.1183 0.2583 0.2156 0.3481 0.3337 0.2872
## 3 0.0262 0.0582 0.1099 0.1083 0.0974 0.2280 0.2431 0.3771 0.5598 0.6194
## 4 0.0100 0.0171 0.0623 0.0205 0.0205 0.0368 0.1098 0.1276 0.0598 0.1264
## 5 0.0762 0.0666 0.0481 0.0394 0.0590 0.0649 0.1209 0.2467 0.3564 0.4459
## 6 0.0286 0.0453 0.0277 0.0174 0.0384 0.0990 0.1201 0.1833 0.2105 0.3039

## ... more here

##   Class
## 1     R
## 2     R
## 3     R
## 4     R
## 5     R
## 6     R
```

reach a cumulative explained variance of 90% by specifying thresh = 0.90 argument.

```
preProcess(Sonar, method = "pca", thresh = 0.90)

## Created from 208 samples and 61 variables
```

```
##
## Pre-processing:
##   - centered (60)
##   - ignored (1)
##   - principal component signal extraction (60)
##   - scaled (60)
##
## PCA needed 22 components to capture 90 percent of the variance
```

According to the summary output, the PCA extracted 22 components to reach a cumulative explained variance of 90%. Note that the variables were centered and scaled in default for this analysis.

Inspect the extracted components using `pca1$rotation`.

```
head(pca1$rotation)
##           PC1        PC2          PC3         PC4        PC5        PC6
## V1 0.13637827 -0.1223047  0.015992208 -0.01398332 0.13559873 -0.1102138
## V2 0.14605308 -0.1310784  0.016703474 -0.06095574 0.16977542 -0.1432289
## V3 0.11572088 -0.1424144  0.008359428 -0.11166025 0.18842191 -0.2173942
## V4 0.09390192 -0.1544207 -0.023779899 -0.10136333 0.17819174 -0.2512730
## V5 0.05534548 -0.1604564  0.025419982 -0.07332698 0.07256160 -0.2126554
## V6 0.05175506 -0.1458544  0.068286250  0.10835565 0.06708443 -0.1715077

## and more up to PC22
```

When we have a prior knowledge on the number of dimensions we can also set the specific number of PCA components to keep using the pcaComp argument.

```
preProcess(Sonar, method = "pca", pcaComp = 3)
## Created from 208 samples and 61 variables
##
## Pre-processing:
##   - centered (60)
##   - ignored (1)
##   - principal component signal extraction (60)
##   - scaled (60)
##
## PCA used 3 components as specified
```

In this example, I applied a PCA analysis with 3 components by specifying pcaComp = 3 argument. We can further inspect the extracted components using pca2$rotation.

```
pca2$rotation
##              PC1          PC2          PC3
## V1   0.136378273 -0.122304688  0.015992208
## V2   0.146053080 -0.131078408  0.016703474
## V3   0.115720878 -0.142414435  0.008359428

## and soforth

## V59  0.143756315 -0.090282629 -0.072818633
## V60  0.117841717 -0.075271297 -0.069013391
```

`prcomp()`

- centers the variable to have mean equals to zero. With
- parameter `scale. = T`, variables are normalised to have standard deviation of 1.

```
prcomp(Sonar, scale. = T)

# Error in colMeans(x, na.rm = TRUE) : 'x' must be numeric
```

because PCA does not work on categorical variables. `prcomp()` we need to manually filter out this factor variable before conducting the analysis.

```
# Exclude Class variable and then apply prcomp

pca3 <- prcomp(Sonar[,-61], scale. = T)
We can see the output of this analysis using the names():
names(pca3)
## [1] "sdev"     "rotation" "center"   "scale"    "x"
```

- The rotation gives the principal component loading, and this is another useful measure used to understand the contribution of each variable to the extracted components/dimensions.
- `head(pca3$x)` - x gives the extracted principal components (equal to rotation in preProcess).

Module 8 - long version

- The learning objectives of this module are as follows:
- Apply basic date-time manipulations using Base R functions
- Apply basic date-time manipulations using lubridate functions
- Learn basic string manipulations using Base R functions
- Learn basic string manipulations using stringr functions

8.0 BaseR

Getting current date and time

Base R has functions to get the current date and time. Also the lubridate package offers fast and user friendly parsing of date-time data. In this section I will use both Base R and lubridate functions to demonstrate date-time manipulations.

In order to get the current date and time information you can use:

```
# get time zone information
Sys.timezone()
## [1] "Australia/Melbourne"
# get date information
Sys.Date()
Sys.time()
## [1] "2018-06-15 22:21:02 AEST"
```

8.1 Lubridate

```
install.packages("lubridate")
library(lubridate)
```

get current time using `lubridate`

```
now()
## [1] "2018-06-15 22:21:02 AEST"
```

You may also get the same information using the lubridate functions:

```
candy <- read.csv("data/candy_production.csv", stringsAsFactors = FALSE)

head(candy)
##   observation_date IPG3113N
## 1       1972-01-01  85.6945
## 2       1972-02-01  71.8200
## 3       1972-03-01  66.0229
## 4       1972-04-01  64.5645
## 5       1972-05-01  65.0100
## 6       1972-06-01  67.6467
# check the structure

str(candy$observation_date)
##  chr [1:548] "1972-01-01" "1972-02-01" "1972-03-01" "1972-04-01" ...
```

BaseR - Converting strings to dates

When date and time data are imported into R they will often default to a character string (or factors if you are using `stringsAs-Factors = FALSE` option). If this is the case, we need to convert strings to proper date format.

To illustrate, let's read in the candy production data which is available here `candy_production.csv`

```
candy$observation_date <- as.Date(candy$observation_date)
# check the structure

str(candy$observation_date)
##  Date[1:548], format: "1972-01-01" "1972-02-01" "1972-03-01" "1972-04-01" "1972-05-01" ...
```

The observation_date variable was read in as a character. In order to convert this to a date format, we can use different strategies. First one is to convert using as.Date() function under Base R.

Note that the default date format is `YYYY-MM-DD`; therefore, if your string is of different format you must incorporate the format argument. There are multiple formats that dates can be in; for a complete list of formatting code options in R type `?strftime` in your console.

For example:

```
x <- c("08/03/2018", "23/03/2016", "30/01/2018")
y <- c("08.03.2018", "23.03.2016", "30.01.2018")
```

This time the string format is `DD/MM/YYYY` for x and `DD.MM.YYYY` for y; therefore, we need to specify the format argument explicitly.

```
x_date <- as.Date(x, format = "%d/%m/%Y")
x_date
## [1] "2018-03-08" "2016-03-23" "2018-01-30"
y_date <- as.Date(y, format = "%d.%m.%Y")
y_date
## [1] "2018-03-08" "2016-03-23" "2018-01-30"
```

Lubridate package

The lubridate package on the other hand can automatically recognise the common separators used when recording dates (-, /, ., and). As a result, you only need to focus on specifying the order of the date elements to determine the parsing function applied. Here is the list of lubridate functions used for this purpose:

Function vs Order of elements in date-time

```
ymd() # year, month, day
ydm() # year, day, month
mdy() # month, day, year
dmy() # day, month, year
hm() #  hour, minute
hms() # hour, minute, second
ymd_hms() # year, month, day, hour, minute, second
```

If the strings are in different formats like the following, the lubridate functions can easily handle these.

```
z <- c("08.03.2018", "29062017", "23/03/2016", "30-01-2018")
dmy(z)
## [1] "2018-03-08" "2017-06-29" "2016-03-23" "2018-01-30"
```

Even with different separators within the same vector, `dmy()` function was able to fetch this information easily.

Extract & manipulate parts of dates

Sometimes, instead of a single string, we will have the individual components of the date-time spread across multiple columns.

selecting just date and time

```
library(nycflights13)
flights_new <- flights %>%
dplyr::select(year, month, day, hour, minute)

head(flights_new, 1)
## # A tibble: 6 x 5
##    year month   day  hour minute
##   <int> <int> <int> <dbl>  <dbl>
## 1  2013     1     1     5     15
```

The date is given in multiple columns

To create a date/time from this sort of input, we can use `make_date()` for dates and `make_datetime()` for date-times.

```
flights_new <- flights_new %>%
  mutate(departure =
    make_datetime(year, month, day, hour, minute))

head(flights_new, 1)
## # A tibble: 6 x 6
##    year month   day  hour minute departure
##   <int> <int> <int> <dbl>  <dbl> <dttm>
## 1  2013     1     1     5     15 2013-01-01 05:15:00
```

Now, let's explore functions that let us get and set individual components of date and time. We can extract individual parts of the date with the accessor functions in lubridate. Here is the list of available functions:

Accessor Function Extracts

`year()`	year
`month()`	month
`mday()`	day of the month
`yday()`	day of the year
`wday()`	day of the week
`hour()`	hour
`minute()`	minute
`second()`	second

Eg. to extract the year information of the `flights_new$departure` column we can use:

```
flights_new$departure %>% year() %>% head()
## [1] 2013 2013 2013 2013 2013 2013
```

For `month()` and `wday()` we can set `label = TRUE` argument to return the abbreviated name of the month or day of the week.

We can also set `abbr = FALSE` to return the full name:

```
flights_new$departure %>% month(label = TRUE, abbr = TRUE) %>% head()
## [1] Jan Jan Jan Jan Jan Jan
## 12 Levels: Jan < Feb < Mar < Apr < May < Jun < Jul < Aug < Sep < ... < Dec
flights_new$departure %>% month(label = TRUE, abbr = FALSE) %>% head()
## [1] January January January January January January
## 12 Levels: January < February < March < April < May < June < ... < December
```

Set date/time

We can also use each accessor function to set the components of a date/time:

create a date

```
datetime <- ymd_hms("2016-07-08 12:34:56")
```

replace the year component with 2020

```
year(datetime) <- 2020
datetime
## [1] "2020-07-08 12:34:56 UTC"
```

replace the month component with Jan

```
month(datetime) <- 01
datetime
## [1] "2020-01-08 12:34:56 UTC"
```

add one hour

```
hour(datetime) <- hour(datetime) + 1
datetime
## [1] "2020-01-08 13:34:56 UTC"
```

Date arithmetic

Often we may require to compute a new variable from the date - time information. In this section, you will learn to create a sequence of dates and how arithmetic with dates works (including subtraction, addition, and division)

For example, to create a sequence of dates we can use the seq() function with specifying the four arguments seq(from, to, by, and length.out).

`seq()`

create a sequence of years from 1980 to 2018 by 2

```
even_years <- seq(from = 1980, to=2018, by = 2)
even_years
##  [1] 1980 1982 1984 1986 1988 1990 1992 1994 1996 1998 2000 2002 2004 2006
## [15] 2008 2010 2012 2014 2016 2018
```

This can be applied for days, months, minutes, seconds, etc.

```
hour_list <- seq (ymd_hm("2018-1-1 9:00"), ymd_hm("2018-1-1 12:00"), by = "hour")

hour_list
## [1] "2018-01-01 09:00:00 UTC" "2018-01-01 10:00:00 UTC"
## [3] "2018-01-01 11:00:00 UTC" "2018-01-01 12:00:00 UTC"
month_list <- seq (ymd_hm("2018-1-1 9:00"), ymd_hm("2018-12-1 9:00"), by = "month")

month_list
##  [1] "2018-01-01 09:00:00 UTC" "2018-02-01 09:00:00 UTC"
##  [3] "2018-03-01 09:00:00 UTC" "2018-04-01 09:00:00 UTC"
##  [5] "2018-05-01 09:00:00 UTC" "2018-06-01 09:00:00 UTC"
##  [7] "2018-07-01 09:00:00 UTC" "2018-08-01 09:00:00 UTC"
##  [9] "2018-09-01 09:00:00 UTC" "2018-10-01 09:00:00 UTC"
## [11] "2018-11-01 09:00:00 UTC" "2018-12-01 09:00:00 UTC"
```

`difftime()`

In R, when you subtract two dates, you get a time intervals/differences object (a.k.a `difftime()` in R) . To illustrate let's calculate my age using:

```
my_age <- today() - ymd(19810529)
my_age
## Time difference of 13531 days
Or, equivalently we can use:
```

```
difftime(today(), ymd(19810529))
## Time difference of 13531 days
```

As seen in the output, subtraction of two date-time objects gives an object of time difference class. In order to change the time difference to another unit we can use units argument:

```
difftime(today(), ymd(19810529), units = "weeks")
## Time difference of 1933 weeks
Logical comparisons are also available for date-time variables.

your_age <- today() - ymd(19890101)
your_age
## Time difference of 10757 days
your_age == my_age
## [1] FALSE
your_age < my_age
## [1] TRUE
```

`duration()`

We can also deal with time intervals/differences by using the duration functions in lubridate. Durations simply measure the time span between start and end dates. lubridate provides simplistic syntax to calculate durations with the desired measurement (seconds, minutes, hours, etc.).

It should be noted that the lubridate package uses seconds as the unit of calculation. Therefore, durations always record the time span in seconds. Larger units are created by converting minutes, hours, days, weeks, and years to seconds at the standard rate **(60 seconds in a minute, 60 minutes in an hour, 24 hours in day, 7 days in a week, 365 days in a year)**.

```
# create a new duration (represented in seconds)
duration(1)
## [1] "1s"
# create durations for minutes
dminutes(1)
## [1] "60s (~1 minutes)"
# create durations for hours
dhours(1)
## [1] "3600s (~1 hours)"
# create durations for years
dyears(1)
## [1] "31536000s (~52.14 weeks)"
# add/subtract durations from date/time object
x <- ymd_hms("2015-09-22 12:00:00")
x + dhours(10)
## [1] "2015-09-22 22:00:00 UTC"
x + dhours(10) + dminutes(33) + dseconds(54)
## [1] "2015-09-22 22:33:54 UTC"
```

9.1 Strings and Characters

String/character manipulations are often overlooked in data analysis because the focus typically remains on numeric values. However, the growth in text mining resulted in greater emphasis on handling, cleaning and processing character strings.

This section includes:

- how to create, convert and print character strings
- how to count the number of elements and characters in a string.

Creating Strings

The most basic way to create strings is to use quotation marks and assign a string to an object similar to creating number sequences like this:

```
a <- "MATH2349"    # create string a
b <- "is awesome"     # create string b
```

`paste()` creating and building strings.

It takes one or more R objects, converts them to character, and then it concatenates (pastes) them to form one or several character strings.

```


# paste multiple strings with a separating character
c <- "Preprocessing"
paste("I", "love", "Data", c, sep = "-")
## [1] "I-love-Data-Preprocessing"

# use paste0() to paste without spaces between characters
paste0("I", "love",  "Data", "Preprocessing")
## [1] "IloveDataPreprocessing"
# paste objects with different lengths

paste("R", 1:5, sep = " v1.")
## [1] "R v1.1" "R v1.2" "R v1.3" "R v1.4" "R v1.5"
## [1] "R v1.1" "R v1.2" "R v1.3" "R v1.4" "R v1.5"
```

`sort()` Sorting character strings:

```
a <- c("MATH2349", "MATH1324")

sort(a)
## [1] "MATH1324" "MATH2349"
```

`toString()` Converting to Strings

Similar to the numerics, strings and characters can be tested with is.character() and any other data format can be converted into strings/characters with as.character() or with toString().

```
a <- "The life of"
b <- pi

is.character(a)
## [1] TRUE
is.character(b)
## [1] FALSE
c <- as.character(b)
is.character(c)
## [1] TRUE
```

```
toString(c("Jul", 25, 2017))
## [1] "Jul, 25, 2017"
```

`print()` - Printing Strings

Printing strings/characters can be done with the following functions:

Function	Usage
`print()`	generic printing
`noquote()`	print with no quotes
`cat()`	concatenate and print with no quotes

The primary printing function in R is `print()`.

```
# basic printing

a <- "MATH2349 is awesome"

print(a)
## [1] "MATH2349 is awesome"
# print without quotes

print(a, quote = FALSE)
## [1] MATH2349 is awesome
# alternative to print without quotes

noquote(a)
## [1] MATH2349 is awesome
```

`cat()` Concatenating strings

The `cat()` function allows us to concatenate objects and print them either on screen or to a file. The output result is very similar to `noquote()`; however, cat() does not print the numeric line indicator. As a result, `cat()` can be useful for printing nicely formatted responses to users.

```
# basic printing (similar to noquote)
cat(a)
## MATH2349 is awesome
# combining character strings

cat(a, "and I love R")
## MATH2349 is awesome and I love R
# basic printing of alphabet

cat(letters)
## a b c d e f g h i j k l m n o p q r s t u v w x y z
# specify a seperator between the combined characters

cat(letters, sep = "-")
## a-b-c-d-e-f-g-h-i-j-k-l-m-n-o-p-q-r-s-t-u-v-w-x-y-z
# collapse the space between the combine characters

cat(letters, sep = "")
## abcdefghijklmnopqrstuvwxyz
```

You can also format the line width for printing long strings using the fill argument:

```
x <- "Today I am learning how to manipulate strings."
y <- "Tomorrow I plan to work on my assignment."
z <- "The day after I will take a break and drink a beer :)"

# No breaks between lines

cat(x, y, z, fill = FALSE)
## Today I am learning how to manipulate strings. Tomorrow I plan to work on my assignment. The day after I will take a break and drink a beer :)
# Breaks between lines

cat(x, y, z, fill = TRUE)
## Today I am learning how to manipulate strings.
## Tomorrow I plan to work on my assignment.
## The day after I will take a break and drink a beer :)
```

`length()` Counting string elements and characters

To count the number of elements in a string use `length()`:

```
length("How many elements are in this string?")
## [1] 1
length(c("How", "many", "elements", "are", "in", "this", "string?"))
## [1] 7
To count the number of characters in a string use nchar():

nchar("How many characters are in this string?")
## [1] 39
nchar(c("How", "many", "characters", "are", "in", "this", "string?"))
## [1]  3  4 10  3  2  4  7
```

9.2 String manipulation with Base R

Basic string manipulation typically includes:

BASE:

- case conversion
- simple character replacement
- abbreviating
- substring replacement

STRINGR:

- adding/removing whitespace
- and performing set operations to compare similarities and differences between two character vectors.

These operations can all be performed with base R functions; however, some operations are greatly simplified with the `stringr` package.

`toupper()` and `tolower()` case conversion

`tolower()` - To convert all upper case characters to lower case:

```r
a <- "MATH2349 is AWesomE"

tolower(a)
## [1] "math2349 is awesome"
```

`toupper()` - To convert all lower case characters to upper case:

```r
toupper(x)
## [1] "TODAY I AM LEARNING HOW TO MANIPULATE STRINGS."
```

`chartr()` Simple Character Replacement

To replace a character (or multiple characters) in a string we can use chartr():

```r
# replace 'A' with 'a'
x <- "This is A string."
chartr(old = "A", new = "a", x)
## [1] "This is a string."
# multiple character replacements
# replace any 'd' with 't' and any 'z' with 'a'

y <- "Tomorrow I plzn do lezrn zbout dexduzl znzlysis."
chartr(old = "dz", new = "ta", y)
## [1] "Tomorrow I plan to learn about textual analysis."
```

Note that chartr() replaces every identified letter for replacement so you need to use it when you are certain that you want to change every possible occurence of that letter(s).

`abbreviate()` String Abbreviations

To abbreviate strings we can use abbreviate():

```r
streets <- c("Victoria", "Yarra", "Russell", "Williams", "Swanston")

# default abbreviations
abbreviate(streets)
## Victoria    Yarra  Russell Williams Swanston
##   "Vctr"   "Yarr"   "Rssl"   "Wllm"   "Swns"
# set minimum length of abbreviation
abbreviate(streets, minlength = 2)
## Victoria    Yarra  Russell Williams Swanston
##     "Vc"     "Yr"     "Rs"     "Wl"     "Sw"
```

`substr()` Extract/Replace Substrings

To extract or replace substrings in a character vector there are two primary base R functions to use: `substr()` and `strsplit()`.

The purpose of `substr()` is to extract and replace substrings with specified starting and stopping characters. Here are some examples on `substr()` usage:

```r
alphabet <- paste(LETTERS, collapse = "")

alphabet
## [1] "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
# extract 18th character in alphabet
substr(alphabet, start = 18, stop = 18)
## [1] "R"
# extract 18-24th characters in alphabet
substr(alphabet, start = 18, stop = 24)
## [1] "RSTUVWX"
# replace 19-24th characters with `R`

substr(alphabet, start = 19, stop = 24) <- "RRRRRR"
alphabet
## [1] "ABCDEFGHIJKLMNOPQRRRRRRRYZ"
```

`strsplit()`

To split the elements of a character string we can use `strsplit()`. Here are some examples:

```r
z <- "The day after I will take a break and drink a beer :)"
strsplit(z, split = " ")
## [[1]]
##  [1] "The"   "day"   "after" "I"     "will"  "take"  "a"     "break"
##  [9] "and"   "drink" "a"     "beer"  ":)"
a <- "Victoria-Yarra-Russell-Williams-Swanston"
strsplit(a, split = "-")
## [[1]]
## [1] "Victoria" "Yarra"    "Russell"  "Williams" "Swanston"
```

Note that the output of `strsplit()` is a list.

`unlist()`

To convert the output to a simple atomic vector simply wrap in `unlist()`:

```r
unlist(strsplit(a, split = "-"))
## [1] "Victoria" "Yarra"    "Russell"  "Williams" "Swanston"
```

9.3 `Vector` operations Set operations for character strings

There are also base R functions that allows for assessing the set union, intersection, difference, equality, and membership of two vectors.

`union()`

To obtain the elements of the union between two character vectors we can use `union()`:

```
set_1 <- c("lagunitas", "bells", "dogfish", "summit", "odell")
set_2 <- c("sierra", "bells", "harpoon", "lagunitas", "founders")

union(set_1, set_2)
## [1] "lagunitas" "bells"     "dogfish"   "summit"    "odell"     "sierra"
## [7] "harpoon"   "founders"
```

`intersect()`

To obtain the common elements of two character vectors we can use intersect().

```
intersect(set_1, set_2)
## [1] "lagunitas" "bells"
```

`setdiff()`

In order to obtain the non-common elements, or the difference, of two character vectors we can use setdiff().

```
# returns elements in set_1 not in set_2
setdiff(set_1, set_2)
## [1] "dogfish" "summit"  "odell"
# returns elements in set_2 not in set_1
setdiff(set_2, set_1)
## [1] "sierra"   "harpoon"  "founders"
```

`setequal()`

In order to test if two vectors contain the same elements regardless of order we can use setequal()

```
set_3 <- c("VIC", "NSW", "TAS")
set_4 <- c("WA", "SA", "NSW")
set_5 <- c("NSW", "SA", "WA")

setequal(set_3, set_4)
## [1] FALSE
setequal(set_4, set_5)
## [1] TRUE
```

`identical()`

We can use identical() to test if two character vectors are equal in content and order.

```
set_6 <- c("VIC", "NSW", "TAS")
set_7 <- c("NSW", "VIC", "TAS")
```

```
set_8 <- c("VIC", "NSW", "TAS")

identical(set_6, set_7)
## [1] FALSE
identical(set_6, set_8)
## [1] TRUE
```

`is.element()`

In order to test if an element is contained within a character vector use `is.element()` or `%in%`. Here are some examples:

```
set_6 <- c("VIC", "NSW", "TAS")
set_7 <- c("NSW", "VIC", "TAS")
set_8 <- c("VIC", "NSW", "TAS")

is.element("VIC", set_8)
## [1] TRUE
"VIC" %in% set_8
## [1] TRUE
"WA" %in% set_8
## [1] FALSE
```

9.4 `stringr` String manipulation

The stringr package was developed by Hadley Wickham to provide a consistent and simple wrappers to common string operations. Before using these functions, we need to install and load the stringr package.

```
install.packages("stringr")
library(stringr)
```

There are three string functions that are closely related to their base R equivalents, but with a few enhancements.

They are:

- Concatenate with `str_c()`
- Number of characters with `str_length()`
- Substring with `str_sub()`

`str_c()`

```
str_c() is equivalent to the paste() function in Base R.

# same as paste0()

str_c("Learning", "to", "use", "the", "stringr", "package")
## [1] "Learningtousethestringrpackage"
# same as paste()

str_c("Learning", "to", "use", "the", "stringr", "package", sep = " ")
## [1] "Learning to use the stringr package"
```

`str_length()`

```
str_length() is similiar to the nchar() function; however, str_length() behaves more appropriately with missing  NA values:

# some text with NA
text = c("Learning", "to", NA, "use", "the", NA, "stringr", "package")

# compare `str_length()` with `nchar()`
nchar(text)
## [1]  8  2 NA  3  3 NA  7  7
str_length(text)
## [1]  8  2 NA  3  3 NA  7  7
```

As seen above, str_length() function returns NA for the missing values, where else, nchar() counts the number of characters in NA and returns 2 as a value.

`str_dup()` Duplicate Characters within a String

The stringr provides a new functionality using str_dup() in which base R does not have a specific function for is character duplication.

```
str_dup("apples", times = 4)
## [1] "applesapplesapplesapples"
str_dup("apples", times = 1:4)
## [1] "apples"                   "applesapples"
## [3] "applesapplesapples"       "applesapplesapplesapples"
```

`str_trim()` Remove Leading and Trailing Whitespace

In string processing, a common task is parsing text into individual words. Often, this results in words having blank spaces (whitespaces) on either end of the word. The `str_trim()` can be used to remove these spaces. Here are some examples:

```
text <- c("Text ", "  with", " whitespace ", " on", "both ", " sides ")
text
## [1] "Text "         "  with"        " whitespace " " on"
## [5] "both "         " sides "
# remove whitespaces on the left side
str_trim(text, side = "left")
## [1] "Text "       "with"        "whitespace " "on"          "both "
## [6] "sides "
# remove whitespaces on the right side
str_trim(text, side = "right")
## [1] "Text"         "  with"      " whitespace" " on"         "both"
## [6] " sides"
# remove whitespaces on both sides
str_trim(text, side = "both")
## [1] "Text"       "with"       "whitespace" "on"         "both"
## [6] "sides"
```

`str_pad()` - Pad a String with Whitespace

To add whitespace, or to pad a string, we will use `str_pad()`. We can also use `str_pad()` to pad a string with specified characters. The width argument will give width of padded strings and the pad argument will specify the padding characters. Here are some examples:

```
str_pad("apples", width = 10, side = "left")
## [1] "    apples"
str_pad("apples", width = 10, side = "both")
## [1] "  apples  "
str_pad("apples", width = 10, side = "right", pad = "!")
## [1] "apples!!!!"
```