R Cheatsheet | ISYS 3340: Data Analytics in Practice

Marios Kokkodis, kokkodis@bc.edu

Good practices to knit

- Save your R Markdown file first in your Rmd directory.
- Load all the needed packages in the first R chunk with the library() function.
- Load any necessary file with the read_csv() function and not manually.

```
library(tidyverse)
read_csv("../data/fileName.csv")
```

R operators

Operator	Description
+, −	Addition, subtraction
*, /	Multiplication, division
%/%, %in%	Integer division, membership operator
&, |, !	Logical AND, OR, NOT
>, >=	Greater than, greater than or equal to
<, <=	Less than, less than or equal to
==, ! =	Exactly equal to, not equal to

Built-in functions

Aggregations and numeric transformations:

```
#na.rm=T ignores any missing values
#estimates the mean of x:
mean(x, na.rm=T)
#estimates standard deviation of x:
sd(x, na.rm=T)
#sums x:
sum(x, na.rm=T)
#log-transforms vector x:
log(x)
#min value of x:
min(x,na.rm=T)
#max value of x:
max(x,na.rm=T)
#concatenate inputs, the following returns:
# data_analytics
paste("data","analytics", sep="_")
```

Logical:

```
#are there missing values in x?
is.na(x)
#returns a vector  c(F,F,T,NA):
ifelse(x > 1,T,F)
```

Basic tibble (and data frame) functionality:

```
#get an overview of the tibble:
summary(t)
#return only the first 2 rows of t:
head(t,2)
#return only the last 2 rows of t:
tail(t,2)
#access a tibble column:
t$numbers
t$words
#create a new column:
t$colors = c("red","yellow",'green')
#each tibble column is a vector:
mean(t$numbers)
#count the number of characters of each word:
nchar(t$words)
```

From file to tibble

```
library(jsonlite)
library(readxl)
#load the songs.csv file to tibble d:
d = read_csv("../data/songs.csv")
#load the example.json file to a tibble:
d <- as_tibble(fromJSON("../data/example.json"))
#load the retail.xlsx file into tibble:
d = read_excel("../data/retail.xlsx")
```

Web scraping

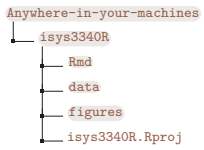
```
library(rvest)
j = read_html("yahoo_finance_URL")) #any website
#identify the html tag (with SelectorGadget) and
#extract information, create a one-column tibble:
j = j %>% html_nodes(".Pend\\(8px\\)") %>%
 html_text() %>% as_tibble_col("comment")
#extract tables from html:
r%>%html_nodes("table")%>%pluck(2)%>%html_table()
```

Dates with lubridate

```
library(lubridate)
#get a weekday from a date:
wday(as_date("2020-09-01", format=...), label=T)
```

Basic directory structure

To run the posted code from a saved R markdown file in your Rmd directory you need to have the following directory structure:



When running code from a saved file inside Rmd, your working directory is:
Anywhere-in-your-machines/isys3340R/Rmd
Hence, to access the data folder from this working directory, we need to go one level up into the isys3340R directory. We can do this with the special keyword ..; within this R markdown file, we can access a file inside the data or figures directories as follows:

../data/filename

dplyr

dplyr is a grammar of data manipulation that provides a set of functions (AKA verbs) that solve some of the most common data manipulation challenges:

```
library(tidyverse)
#select only column "numbers"
select(t,numbers) #selects columns
#keep only rows with the word "data":
filter(t,words =="data") #selects rows
#create two new columns and update tibble t:
t = mutate(t,colors = c("red","yellow",'green'),
           shapes = c(0,1,2))
#keep only a new column:
transmute(t,colorsAndShapes = paste(colors,
                                    shapes, sep=">"))
#rank t in decreasing order of numbers:
arrange(t,desc(numbers))
#rename column shapes to newShape:
rename(t, newShape = shapes)
#group by words
g = group_by(t,words)
#summarize based on groups:
summarize(g,meanNumbers = mean(numbers))
#or summarize without grouping,
#across the tibble:
summarize(t,meanNumbers = mean(numbers),
          stdNumbers = sd(numbers))
```

Instead of giving a tibble as input to each dplyr verb, we can use pipes %>%. The way to read a pipe is "and then". Here we connect some of the abover operations through pipes (**note that with pipes, we do not include the tibble t as input inside the verbs**):

```
t %>%  mutate(colors = c("red","yellow",'green'),
           shapes = c(0,1,2)) %>%
  group_by(words) %>%
  summarize(meanNumbers = mean(numbers))
```

API calls

```
library(httr)
#Access the API endpoint with GET()
r = GET("API_endpoint_URL")
#Binary to char and from json to tibble:
d = as_tibble(fromJSON(rawToChar(r$content)))
```

Twitter wrapper rtweet:

```
library(rtweet)
library(tidyquant)
#authenticate with your credentials:
myToken = create_token(app = app_name,
  consumer_key = key, consumer_secret = secret,
  access_token = access,access_secret = a_secret)
#search tweets that include ``search phrase'':
a = search_tweets("search phrase", n=500,
  token = myToken, include_rts = F)
#get stock market info:
r = c('VTI') %>% tq_get(get ="stock.prices",
    from = "2020-01-01",
    to = "2020-09-25")
```

Custom functions

```
#below I identify a function that takes as input
#one parameter: stockSymbol
getYahooFinanceComments = function(stockSymbol){
  [..define functionality..]
  #return an R object (number, tibble, etc.):
  return(j)
}
```

Basic terminology

- variable = object: nicknames that store values or other structures.
- vector c(): a sequence of values.
- data types: character (string), numeric (or double), logical (T or F), date
- tibble = data frame (for all practical purposes): a spreadsheet object with columns and rows.
- function(*input*): a predefined set of commands that transform the *input* and return an *output*.

Cheatsheet variables

Across the cheatsheet, I will assume that:

```
#x is a numeric vector
x = c(0,1,2,NA)
# t is a tibble:
t = tibble(numbers = c(2,4,7),
           words = c("hello","data",'happy'))
```

ggplot2

To create graphics ggplot2 uses the following syntax:
ggplot(<DATA>, aes(<MAPPINGS>)) + <GEOM_FUNCTION>()
where:

- <DATA> represents your tibble dataset.
- <MAPPINGS> are the aesthetic mappings of your plots which describe how variables in the data are mapped to visual properties.
- <GEOM_FUNCTION> is a geometrical object that a plot uses to represent data such as a bar plot, a histogram, a line, and so on.

```
library(ggplot2)
library(ggthemes)
maroon = "#8a100b"
gold = "#b29d6c"
gray = "#726158"
legendTitle=""
#put column numbers on the x axis
#put column shapes on the y axis
#map color and shape to the column words
ggplot(t,aes(x=numbers, y=shapes,
             color=words, shape=words))+
  geom_point()+ #add points
  geom_smooth(method="lm")+ #add smoothed line
  geom_line()+ #add line over points
  #now customize the color of each point
  scale_color_manual(values=c(maroon,gold),
                     name=legendTitle)+
  #customize the shape of each point
  scale_shape_manual(values=c(3,4),
                     name=legendTitle)+
  #rename x and y axis:
  xlab("Numbers")+ylab("Shapes")+
  #change theme:
  theme_tufte()
```

For barplots:

```
#create a barplot where on the x-axis are the
#different words in column "words".
#Color each bar based on the words.
ggplot(t, aes(x=words, color=factor(words),
              fill = factor(words))) +
  geom_bar(size=2,alpha=0.7,position = "dodge")+
  #the outline of the bars:
  scale_color_manual(values = c(maroon,gold),
                     name=legendTitle)+
  #the fill color of the bars:
  scale_fill_manual(values = c(maroon,gold),
                    name=legendTitle)+
  #legend in the top, plots next to each other:
  theme(legend.position="top")+facet_wrap(~words)
```

Distributions:

```
#on the y axis, show the density
#(alternatively, the y axis can be ..count..)
ggplot(t,aes(x=numbers, y = ..density..))+
  #create a histogram:
  geom_histogram(alpha=0.7,fill=gold)+
  #add the smoothed distribution:
  geom_density(color=maroon, size=1.5)
```

Repetitive operations with map

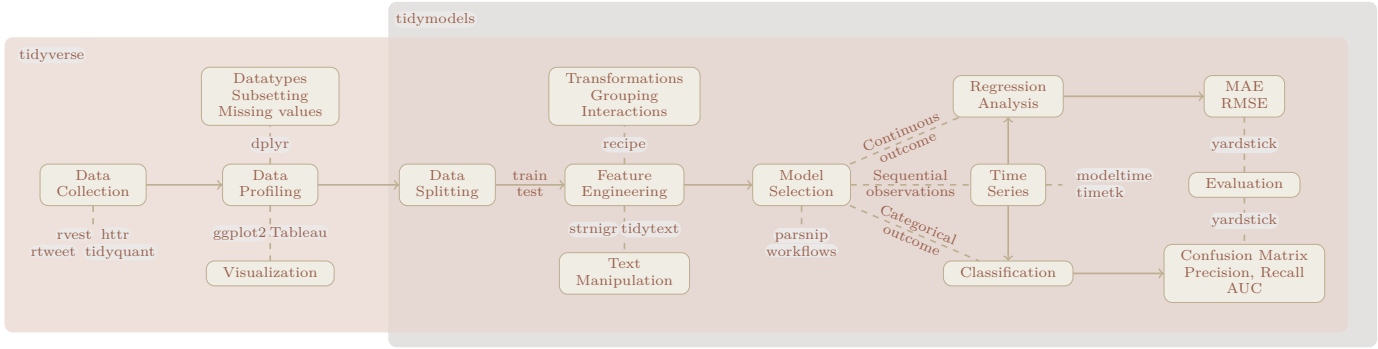
The map() function transforms the input object by applying a given function to each element of the input:

```
#apply the function `nchar` to all words:
t %>% select(words) %>% map(nchar)
#get the average word size with map_dbl:
select(t,words) %>% map(nchar) %>% map_dbl(mean)
#map on custom functions:
c('TSLA','ZM')%>%map_dfr(getYahooFinanceComments)
```

R Cheatsheet | ISYS 3340: Data Analytics in Practice

Marios Kokkodis, kokkodis@bc.edu

The process of building models



Regular expressions with `stringr`

```
#is "L" in "HELLO"?
str_detect("HELLO","L")
#transform to lowercase the column "words"
t = t %>% mutate(words = str_to_lower(words))
#keep only words that have letters b or n
t %>% filter(str_detect(words, "b|n"))
#replace all "w" with "LL":
t = t %>% mutate(words=str_replace_all(words,"w","LL"))
#create a column numbers that extracts digits from words:
t = t %>% mutate(t = str_extract(words,"[0-9]+"))
```

Joins

A join is an operation that in its simplest form, it takes a left tibble and matches information for each row from a right tibble based on a common column between the two tibbles.

```
#assume the following two tibbles:
join1 = tibble(id=c(1,2), info=c("some","info"))
join2 = tibble(id=c(1,2,3),
    some_add_info = c("some","add","info"))
#an inner join retains rows found in both tibbles:
join1 %>% inner_join(join2, by="id")
#a left join keeps all info from the left tibble:
join2 %>% left_join(join1, by="id")
#an anti join keeps only elements
#that are not found in the right tibble:
join2 %>% anti_join(join1, by="id")
```

Regression models with `parsnip`

If `DV` is continuous, then we have a regression problem. We can define regression models as follows:

```
#linear regression
lm_reg = linear_reg() %>% set_engine("lm") %>%
                          set_mode("regression")
#gradient boosting
xg_reg = boost_tree() %>% set_engine("xgboost") %>%
                          set_mode("regression")
#decision trees regression
dt_reg = decision_tree() %>% set_engine("rpart") %>%
                          set_mode("regression")
#svm regression
svm_reg = svm_poly() %>% set_engine("kernlab") %>%
                          set_mode("regression")
#random forest classification
rf_reg = rand_forest() %>% set_engine("ranger") %>%
                          set_mode("regression")
#multilayer perceptron regression
mlp_reg = mlp() %>% set_engine("keras") %>%
                          set_mode("regression")
```

Time series models with `modeltime`

If `DV` is continues and is affected by the time component `date_time`, then we have a time series regression problem. We can define time series models as follows:

```
#Arima regression
arima_reg = arima_reg() %>%
                  set_engine("auto_arima")
#Prophet regression
prophet_reg = prophet_reg() %>%
                  set_engine("prophet")
#Arima with XGBoost
arima_reg_b = arima_boost() %>%
                  set_engine("arima_xgboost")
#Prophet with XGBoost
prophet_reg_b = prophet_boost() %>%
                  set_engine("prophet_xgboost")
#build and evaluate a time series model:
m1 = bw %>% update_model(prophet_reg) %>% fit(trs)
m2 = bw %>% update_model(arima_reg) %>% fit(trs)
mt = modeltime_table(m1, m2) ct = mt %>%
    modeltime_calibrate(ts)
#print out the accuracies:
ct %>% modeltime_accuracy() %>% select(.model_id,
  .model_desc , mae, rmse)
```

Text mining with `tidytext`

Bag of words approach:

```
#Let's unnest the column "words" of tibble t:
t1 = t %>% unnest_tokens(word, words)
#remove stop words:
t2 = t1 %>% anti_join(stop_words, by="word")
#create a document-term matrix
#(recall tibble t has a column numbers as well)
t2 %>% count(word,numbers) %>%
pivot_wider(names_from=word,values_from=n,values_fill=0)
# tf-idf:
t2 %>% count(word,numbers) %>%
bind_tf_idf(word,numbers,n) %>%
select(word,numbers,tf_idf) %>%
pivot_wider(names_from = word, values_from = tf_idf,
                                        values_fill=0)
```

Beyond bag of words, we can estimate the sentiment of each document:

```
#sentiment analysis based on lexicons:
nrc_lexicon = get_sentiments("nrc")
#first, let's keep only sentiment words:
t2 = t1 %>% inner_join(nrc_lexicon, by="word")
#now we can count the sentiment of each document;
t2 %>% count(numbers,sentiment) %>%
pivot_wider(names_from = sentiment, values_from=n,
                                        values_fill=0)
```

Classification models with `parsnip`

If `DV` is binary or categorical (a factor), then we have a classification problem. We can define classification models as follows:

```
#logistic regression
lg_class = logistic_reg() %>% set_engine("glm") %>%
                              set_mode("classification")
#gradient boosting
xg_class = boost_tree() %>% set_engine("xgboost") %>%
                              set_mode("classification")
#decision trees regression
dt_class = decision_tree() %>% set_engine("rpart") %>%
                              set_mode("classification")
#svm regression
svm_class = svm_poly() %>% set_engine("kernlab") %>%
                              set_mode("classification")
#random forest regression
rf_reg = rand_forest() %>% set_engine("ranger") %>%
                              set_mode("classification")
#multilayer perceptron classification
mlp_reg = mlp() %>% set_engine("keras") %>%
                              set_mode("classification")
```

Training and testing

```
#to train a model, we call the function fit:
learnedModel = bw %>% fit(data=train_set)
#we can get predictions by calling the function
#predict on the testing set:
predict(learnedModel, test_set) %>%
        bind_cols(test_set %>% select(DV))
#if we have a classification problem, we can get
#probability predictions by including the term "prob":
predict(learnedModel, test_set, "prob") %>%
        bind_cols(test_set %>% select(DV))
```

Threshold analysis

```
# pr is a tibble w probabilities (see evaluation)
thresholdData = pr %>% threshold_perf(DV, .pred_Label,
    thresholds = seq(0.2,0.8, by=0.0025))
# get the threshold that maximizes the j-index
thresholdData %>% filter(.metric == "j_index") %>%
    filter(.estimate == max(.estimate)) %>% head
# plot thresholds:
thresholdData %>% ggplot(aes(x=.threshold, y =.estimate,
                        color =.metric))+ geom_line()
```

Feature engineering with `recipes`

The following assumes a tibble `d` with a column `DV` that will play the role of a dependent variable, columns `X1,X2,X3` that will play the role of independent variables, and a column `date_time` that stores a time component.
For both regression and classification, our first task is to split the data into training and test sets:

```
#assumes 4/5 as training, 1/5 as testing
data_split = initial_split(d,prop = 4/5)
train_set = training(data_split)
test_set = testing(data_split)
```

In time series problems, we need to make sure that our test set comes after the train set. Hence, we use a slightly different approach:

```
splits = d %>% time_series_split(date_var = date_time,
                                 assess = "6 months",
                                 cumulative = T)
trs = training(splits) ts = testing(splits)
```

Once we split into training and test set, we can use the package recipes to define models and do feature engineering:

```
#a very basic recipe:
br = recipe(DV ~ X1 + X2 + X3,train_set) %>%
#assumes X1 has NAs, which we impute to X1's median:
    step_medianimpute(X1)  %>%
    step_dummy(all_nominal(), -all_outcomes())
#we can extend this recipe:
er = br %>% step_interact(terms = ~X1:X2) %>%
    step_log(X2)
```

For our time series problem, we can define a recipe as follows:

```
time_rec = recipe(DV ~date_time+X1+X2+X3,trs) %>%
step_dummy(all_nominal())
```

For any recipe, we can see the result by using `prep` and `juice`:

```
br %>% prep %>% juice
```

Additional step functions can be found here: https://recipes.tidymodels.org/reference/index.html

Workflows

Workflows allow us to combine recipes and models:

```
#here is a workflow that uses a regression model
#and our basic recipe br
bw = workflow() %>% add_model(lm_reg) %>%
                    add_recipe(basic_recipe)
#we can update the workflow's model:
bw %>% update_model(xgboost_reg)
#we can update the workflow's recipe:
bw %>% update_recipe(fe_recipe)
```

Evaluation

```
#If we have a regression problem:
p = tpredict(learnedModel, test_set) %>%
        bind_cols(test_set %>% select(DV))
p %>% rmse(.pred, DV)
p %>% mae(.pred, DV)
#If we have a classification problem:
#confusion matrix:
p %>%  conf_mat(DV,.pred_class)
#accuracy:
p %>% accuracy(DV,.pred_class)
#AUC
pr = tpredict(learnedModel, test_set, "prob") %>%
        bind_cols(test_set %>% select(DV))
#to find whether your event is first or second run:
levels(DV)
roc_auc(truth = DV, .pred_Label, event_level = "first")
```