PUBLIC HEALTH AWARENESS ANALYSIS

Project Submission Part 3: Development Part 1

Project title: DAC_Phase3

Batch member: P.Pavithra

Phase 3 submission document : Public Health Awareness analysis load and preprocessing dataset

Data Pre-processing in R

There are many tools for doing data pre-processing available, such as R, STATA, SAS, and Python; each differs in the level of programming background required. R is a free tool that is supported by a range of statistical and data manipulation packages. In this section of the chapter, we will go through some examples demonstrating various steps of data pre-processing in R, using data from various MIMIC dataset (SQL extraction codes included). Due to the significant content involved with the data cleaning step of pre-processing, this step will be separately addressed in Chaps. 13 and 14. The examples in this section will deal with some R basics as well as data integration, transformation, and reduction.

R—The Basics

The most common data output from a MIMIC database query is in the form of 'comma separated values' files, with filenames ending in '.csv'. This output file format can be selected when exporting the SQL query results from MIMIC database. Besides '.csv' files, R is also able to read in other file formats, such as Excel, SAS, etc., but we will not go into the detail here.

Understanding 'Data Types' in R

For many who have used other data analysis software or who have a programming background, you will be familiar with the concept of 'data types'.

R strictly stores data in several different data types, called 'classes':

Numeric – e.g. 3.1415, 1.618

Integer – e.g. -1, 0, 1, 2, 3

Character – e.g. “vancomycin”, “metronidazole”

Logical – TRUE, FALSE

Factors/categorical – e.g. male or female under variable, gender

R also usually does not allow mixing of data types for a variable, except in a:

List – as a one dimensional vector, e.g. c(“vancomycin”, 1.618, “red”)

Data-frame – as a two dimensional table with rows (observations) and columns (variables)

Lists and data-frames are treated as their own ‘class’ in R.

Query output from MIMIC commonly will be in the form of data tables with different data types in different columns. Therefore, R usually stores these tables as ‘data-frames’ when they are read into R.

Special Values in R

NA – ‘not available’, usually a default placeholder for missing values.

NAN – ‘not a number’, only applying to numeric vectors.

NULL – ‘empty’ value or set. Often returned by expressions where the value is undefined.

Inf – value for ‘infinity’ and only applies to numeric vectors.

Setting Working Directory

This step tells R where to read in the source files.

command: setwd(“directory_path”)

Example: (If all data files are saved in directory “MIMIC_data_files” on the Desktop)

Reading in .csv Files from MIMIC Query Results

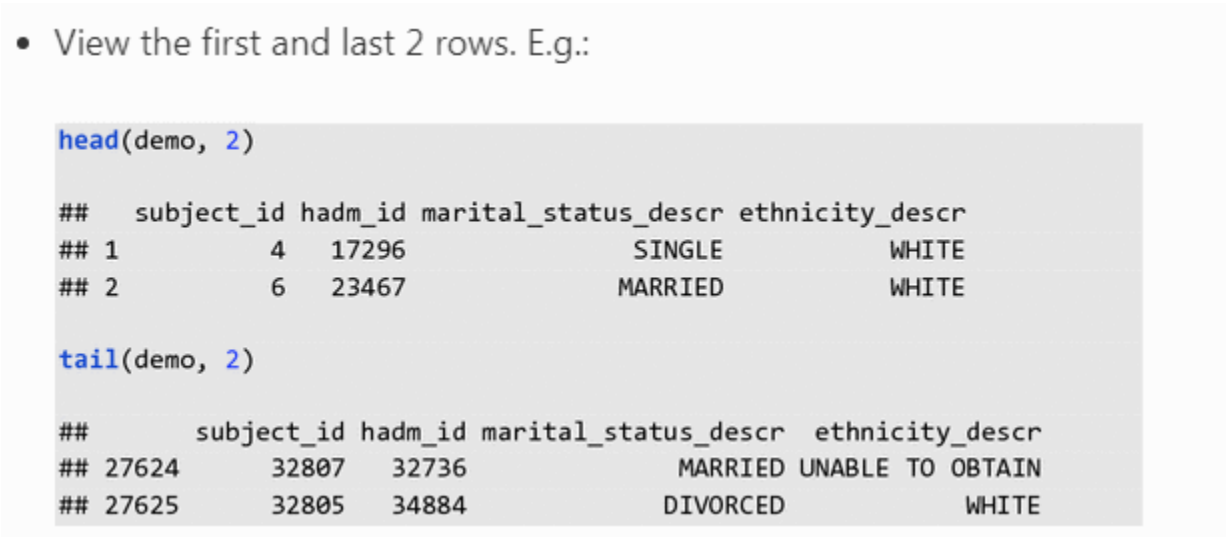
The data read into R is assigned a ‘name’ for reference later on.

Command: set_var_name <- read.csv(“filename.csv”)

Viewing the Dataset

There are several commands in R that are very useful for getting a ‘feel’ of your datasets and see what they look like before you start manipulating them.

- View the first and last 2 rows. E.g.:



- View summary statistics. E.g.:

```
summary(demo)

##    subject_id       hadm_id      marital_status_descr
##  Min.   :    3   Min.   :    1     MARRIED  :13447
##  1st Qu.: 8063   1st Qu.: 9204   SINGLE   : 6412
##  Median :16060   Median :18278   WIDOWED  : 4029
##  Mean   :16112   Mean   :18035   DIVORCED : 1623
##  3rd Qu.:24119   3rd Qu.:26762            : 1552
##  Max.   :32809   Max.   :36118   SEPARATED:  320
##                                    (Other)  :  242
##                ethnicity_descr
##  WHITE                  :19360
##  UNKNOWN/NOT SPECIFIED : 3446
##  BLACK/AFRICAN AMERICAN: 2251
##  …
```

- View structure of data set (obs = number of rows). E.g.:

```
str(demo)

## 'data.frame':    27625 obs. of  4 variables:
##  $ subject_id          : int  4 6 3 9 15 14 11 18 18 19 ...
##  $ hadm_id             : int  17296 23467 2075 8253 4819 23919 28128
24759 33481 25788 ...
##  $ marital_status_descr: Factor w/ 8 levels "","DIVORCED",..: 6 4 4
1 6 4 4 4 4 1 ...
##  $ ethnicity_descr     : Factor w/ 39 levels "AMERICAN INDIAN/ALASKA
NATIVE",..: 35 35 35 34 12 35 35 35 35 35 ...
```

- Find out the 'class' of a variable or dataset. E.g.:

```
class(demo)

## [1] "data.frame"
```

- View number of rows and column, or alternatively, the dimension of the da

```
nrow(demo)

## [1] 27625

ncol(demo)

## [1] 4

dim(demo)

## [1] 27625     4
```

Subsetting a Dataset and Adding New Variables/Columns

Aim: Sometimes, it may be useful to look at only some columns or some rows in a dataset/data-frame—this is called subsetting.

Let's create a simple data-frame to demonstrate basic subsetting and other command functions in R. One simple way to do this is to create each column of the data-frame separately then combine them into a dataframe later. Note the different kinds of data types for the columns/variables created, and beware that R is case-sensitive.

```
subject_id <- c(1:6)                                    #integer
gender <- as.factor(c("F", "F", "M", "F", "M", "M"))#factor/categorical
height <- c(1.52, 1.65, 1.75, 1.72, 1.85, 1.78)     #numeric
weight <- c(56.7, 99.6, 90.4, 85.3, 71.4, 130.5)    #numeric
data <- data.frame(subject_id, gender, height, weight)

head(data, 4)                                  # View only the first 4 rows

##   subject_id gender height weight
## 1          1      F   1.52   56.7
## 2          2      F   1.65   99.6
## 3          3      M   1.75   90.4
## …

str(data)                  # Note the class of each variable/column

## 'data.frame':    6 obs. of  4 variables:
##  $ subject_id: int  1 2 3 4 5 6
##  $ gender    : Factor w/ 2 levels "F","M": 1 1 2 1 2 2
##  $ height    : num  1.52 1.65 1.75 1.72 1.85 1.78
##  $ weight    : num  56.7 99.6 90.4 85.3 71.4 ...
```

To subset or extract only e.g., weight, we can use either the dollar sign ($) after the dataset, data, or use the square brackets, []. The $ selects column with the column name (without quotation mark in this case). The square brackets [] here selected the column weight by its column number:

```
w1 <- data$weight; w1

## [1]  56.7  99.6  90.4  85.3  71.4 130.5

w2 <- data[, 4]; w2

## [1]  56.7  99.6  90.4  85.3  71.4 130.5
```

Generally one can subset a dataset by specifying the rows and column desired like this: data[row number, column number]. For example:

```
dat_sub <- data[2:4, 1:3]; dat_sub

##   subject_id gender height
## 2          2      F   1.65
## 3          3      M   1.75
## 4          4      F   1.72
```

The square brackets are useful for subsetting multiple columns or rows. Note that it is important to 'concatenate', c(), if selecting multiple variables/columns and to use quotation marks when selecting with columns names

```
h_w1 <- data[, c(3, 4)]; h_w1

##   height weight
## 1   1.52   56.7
## 2   1.65   99.6
## 3   1.75   90.4
## …

h_w2 <- data[, c("height", "weight")]; h_w2

##   height weight
## 1   1.52   56.7
## 2   1.65   99.6
## 3   1.75   90.4
## …
```

To calculate the BMI (weight/height^2) in a new column—there are different ways to do this but here is a simple method:

```
data$BMI <- data$weight/data$height^2
head(data, 4)

##   subject_id gender height weight      BMI
## 1          1      F   1.52   56.7 24.54120
## 2          2      F   1.65   99.6 36.58402
## 3          3      M   1.75   90.4 29.51837
## 4          4      F   1.72   85.3 28.83315
```

Let's create a new column, obese, for BMI > 30, as TRUE or FALSE. This also demonstrates the use of 'logicals' in R.

One can also use logical vectors to subset datasets in R. A logical vector, named "ob" here, is created and then we pass it through the square brackets [] to tell R to select only the rows where the condition BMI > 30 is TRUE:

```
ob <- data$BMI > 30
data_ob <- data[ob, ];data_ob

##   subject_id gender height weight      BMI obese
## 2          2      F   1.65   99.6 36.58402  TRUE
## 6          6      M   1.78  130.5 41.18798  TRUE
```

Combining Datasets (Called Data Frames in R)

Aim: Often different variables (columns) of interest in a research question may come from separate MIMIC tables and could have been exported as separate.csv files if they were not merged via SQL queries. For ease of analysis and visualization, it is often desirable to merge these separate data frames in R on their shared ID column(s).

Occasionally, one may also want to attach rows from one data frame after rows from another. In this case, the column names and the number of columns of the two different datasets must be the same.

Examples: In general, there are a couple ways of combining columns and rows from different datasets in R:

- merge()—This function merges columns on shared ID column(s) between the data frames so the associated rows match up correctly.

Command: merging on one ID column, e.g.:

```
df_merged <- merge(df1, df2, by = "column_ID_name")
```

Command: merging on two ID columns, e.g.:

```
df_merged <- merge(df1, df2, by = c("column1", "column2"))
```

- cbind()—This function simply 'add' together the columns from two data frames (must have equal number of rows). It does not match up the rows by any identifier.

Command: joining columns. E.g.:

```
df_total <- cbind(df1, df2)
```

- • rbind()—The function ‘row binds’ the two data frames vertically (must have the same column names).

Command: joining rows. E.g.:

```
df_total <- rbind(df1, df2)
```

Using Packages in R

There are many packages that make life so much easier when manipulating data in R. They need to be installed on your computer and loaded at the start of your R script before you can call the functions in them. We will introduce examples of of a couple of useful packages later in this chapter.

For now, the command for installing packages is:

```
install.packages("name_of_package_case_sensitive")
```

The command for loading the package into the R working environment:

```
library(name_of_package_case_sensitive)
```

Note—there are no quotation marks when loading packages as compared to installing; you will get an error message otherwise.

Getting Help in R

There are various online tutorials and Q&A forums for getting help in R. Stackoverflow, Cran and Quick-R are some good examples. Within the R console, a question mark, ?, followed by the name of the function of interest will bring up the help menu for the function, e.g.

```
?head
```

3.2 Data Integration

Aim: This involves combining the separate output datasets exported from separate MIMIC queries into a consistent larger dataset table.

To ensure that the associated observations or rows from the two different datasets match up, the right column ID must be used. In MIMIC, the ID columns could be subject_id, hadm_id, icustay_id, itemid, etc. Hence, knowing the context of what each column ID is used to identify and how they are related to each other is important. For example, subject_id is used to identify each individual patient, so includes their date of birth (DOB), date of death (DOD) and various other clinical detail and laboratory values in MIMIC. Likewise, the hospital admission ID, hadm_id, is used to specifically identify various events and outcomes from an unique hospital admission; and is also in turn associated with the subject_id of the patient who was involved in that particular hospital admission. Tables pulled from MIMIC can have one or more ID columns. The different tables exported from MIMIC may share some ID columns, which allows us to 'merge' them together, matching up the rows correctly using the unique ID values in their shared ID columns.

Examples: To demonstrate this with MIMIC data, a simple SQL query is constructed to extract some data, saved as: "population.csv" and "demographics.csv".

We will these extracted files to show how to merge datasets in R.

1. 1.

SQL query:

```
WITH
population AS(
SELECT subject_id, hadm_id, gender, dob, icustay_admit_age,
icustay_intime, icustay_outtime, dod, expire_flg
FROM mimic2v26.icustay_detail
  WHERE subject_icustay_seq = 1
  AND icustay_age_group = 'adult'
  AND hadm_id IS NOT NULL
)
, demo AS(
SELECT subject_id, hadm_id, marital_status_descr, ethnicity_descr
FROM mimic2v26.demographic_detail
WHERE subject_id IN (SELECT subject_id FROM population)
)

--# Extract the the datasets with each one of the following line of
codes in turn:
--SELECT * FROM population
--SELECT * FROM demo
```

Note: Remove the -- in front of the SELECT command to run the query.

1. 2.

R code: Demonstrating data integration

Set working directory and read data files into R::

```
setwd("~/Desktop/MIMIC_data_files")
demo <- read.csv("demographics.csv", sep = ",")
pop <- read.csv("population.csv", sep = ",")
head(demo)

##   subject_id hadm_id marital_status_descr        ethnicity_descr
## 1          4   17296                SINGLE                  WHITE
## 2          6   23467               MARRIED                  WHITE
## 3          3    2075               MARRIED                  WHITE
## …
head(pop)

##   subject_id hadm_id gender                 dob icustay_admit_age
## 1          4   17296      F 3351-05-30 00:00:00          47.84414
## 2          6   23467      F 3323-07-30 00:00:00          65.94048
## 3          3    2075      M 2606-02-28 00:00:00          76.52892
## …


##        icustay_intime     icustay_outtime                 dod
expire_flg
## 1 3399-04-03 00:29:00 3399-04-04 16:46:00
N
## 2 3389-07-07 20:38:00 3389-07-11 12:47:00
N
## 3 2682-09-07 18:12:00 2682-09-13 19:45:00 2683-05-02 00:00:00
Y
## …
```

Merging pop and demo: Note to get the rows to match up correctly, we need to merge on both the subject_id and hadm_id in this case. This is because each subject/patient could have multiple hadm_id from different hospital admissions during the EHR course of MIMIC database.

```
demopop <- merge(pop, demo, by = c("subject_id", "hadm_id"))
head(demopop)

##   subject_id hadm_id gender                 dob icustay_admit_age
## 1        100     445      F 3048-09-22 00:00:00          71.94482
## 2       1000   15170      M 2442-05-11 00:00:00          69.70579
## 3      10000   10444      M 3149-12-07 00:00:00          49.67315
## …


##        icustay_intime     icustay_outtime                 dod
expire_flg
## 1 3120-09-01 11:19:00 3120-09-03 14:06:00
N
## 2 2512-01-25 13:16:00 2512-03-02 06:05:00 2512-03-02 00:00:00
Y
## 3 3199-08-09 09:53:00 3199-08-10 17:43:00
N
## …

##   marital_status_descr        ethnicity_descr
## 1              WIDOWED  UNKNOWN/NOT SPECIFIED
## 2              MARRIED  UNKNOWN/NOT SPECIFIED
## 3                         HISPANIC OR LATINO
## 4              MARRIED BLACK/AFRICAN AMERICAN
## 5              MARRIED                  WHITE
## 6            SEPARATED BLACK/AFRICAN AMERICAN
```

As you can see, there are still multiple problems with this merged database, for example, the missing values for 'marital_status_descr' column. Dealing with missing data is explored in Chap. 13.

Data Transformation

Aim: To transform the presentation of data values in some ways so that the new format is more suitable for the subsequent statistical analysis. The main processes involved are normalization, aggregation and generalization (See part 1 for explanation).

Examples: To demonstrate this with a MIMIC database example, let us look at a table generated from the following simple SQL query, which we exported as "comorbidity_scores.csv".

The SQL query selects all the patient comorbidity information from the mimic2v26.comorbidity_scores table on the condition of (1) being an adult, (2) in his/her first ICU

admission, and (3) where the hadm_id is not missing according to the mimic2v26.icustay_detail table.

1. 1.

SQL query:

```
SELECT *
FROM mimic2v26.comorbidity_scores
WHERE subject_id IN (SELECT subject_id
        FROM mimic2v26.icustay_detail
        WHERE subject_icustay_seq = 1
                AND icustay_age_group = 'adult'
                AND hadm_id IS NOT null)
```

2. 2.

R code: Demonstrating data transformation:

```
setwd("~/Desktop/MIMIC_data_files")
c_scores <- read.csv("comorbidity_scores.csv", sep = ",")
```

Note the ‘class’ or data type of each column/variable and the total number of rows (obs) and columns (variables) in c_scores:

```
str(c_scores)

## 'data.frame':    27525 obs. of  33 variables:
##  $ subject_id              : int  2848 21370 2026 11890 27223 27520
17928 31252 32083 9545 ...
##  $ hadm_id                 : int  16272 17542 11351 12730 32530
32724 20353 30062 32216 10809 ...
##  $ category                : Factor w/ 1 level "ELIXHAUSER": 1 1 1 1
1 1 1 1 1 1 ...
##  $ congestive_heart_failure: int  0 0 0 0 1 0 0 0 1 1 ...
##  $ cardiac_arrhythmias     : int  0 1 1 0 1 0 0 0 0 1 ...
##  $ valvular_disease        : int  0 0 0 0 1 0 0 0 0 1 ...
##  $ …
```

Here we add a column in c_scores to save the overall ELIXHAUSER. The rep() function in this case repeats 0 for nrow(c_scores) times. Function, colnames(), rename the new or last column, [ncol(c_scores)], as “ELIXHAUSER_overall”.

```
c_scores <- cbind(c_scores, rep(0, nrow(c_scores)))
colnames(c_scores)[ncol(c_scores)] <- "ELIXHAUSER_overall"
```

Take a look at the result. Note the new “ELIXHAUSER_overall” column added at the end:

```
str(c_scores)

## 'data.frame':    27525 obs. of  34 variables:
##  $ subject_id              : int  2848 21370 2026 11890 27223 27520
17928 31252 32083 9545 ...
##  $ hadm_id                 : int  16272 17542 11351 12730 32530
32724 20353 30062 32216 10809 ...
##  $ category                : Factor w/ 1 level "ELIXHAUSER": 1 1 1 1
1 1 1 1 1 1 ...
##  $ congestive_heart_failure: int  0 0 0 0 1 0 0 0 1 1 ...
##  $ cardiac_arrhythmias     : int  0 1 1 0 1 0 0 0 0 1 ...
##  $ valvular_disease        : int  0 0 0 0 1 0 0 0 0 1 ...
##  $ …
```

Aggregation Step

Aim: To sum up the values of all the ELIXHAUSER comorbidities across each row. Using a ‘for loop’, for each i-th row entry in column “ELIXHAUSER_overall”, we sum up all the comorbidity scores in that row.

```
for (i in 1:nrow(c_scores)) {
  c_scores[i, "ELIXHAUSER_overall"] <- sum(c_scores[i,4:33])
}
```

Let’s take a look at the head of the resulting first and last column:

```
head(c_scores[, c(1, 34)])

##   subject_id ELIXHAUSER_overall
## 1       2848                  1
## 2      21370                  3
## 3       2026                  3
## …
```

Normalization Step

Aim: Scale values in column ELIXHAUSER_overall to between 0 and 1, i.e. in [0, 1]. Function, max(), finds out the maximum value in column ELIXHAUSER *overall. We then re-assign each*

*entry in column ELIXHAUSER*overall as a proportion of the max_score to normalize/scale the column.

```
max_score <- max(c_scores[,"ELIXHAUSER_overall"])
c_scores[,"ELIXHAUSER_overall"] <- c_scores[ ,
"ELIXHAUSER_overall"]/max_score
```

We subset and remove all the columns in c_score, except for “subject_id”, “hadm_id”, and “ELIXHAUSER_overall”:

```
c_scores <- c_scores[, c("subject_id", "hadm_id",
"ELIXHAUSER_overall")]
head(c_scores)

##   subject_id hadm_id ELIXHAUSER_overall
## 1       2848   16272         0.09090909
## 2      21370   17542         0.27272727
## 3       2026   11351         0.27272727
## …
```

Generalization Step

Aim: Consider only the patient sicker than the average Elixhauser score. The function, which(), return the row numbers (indices) of all the TRUE entries of the logical condition set on c_scores inside the round () brackets, where the condition being the column entry for ELIXHAUSER_overall ≥0.5. We store the row indices information in the vector, ‘sicker’. Then we can use ‘sicker’ to subset c_scores to select only the rows/patients who are ‘sicker’ and store this information in ‘c_score_sicker’.

```
sicker <- which(c_scores[,"ELIXHAUSER_overall"]>=0.5)
c_score_sicker <- c_scores[sicker, ]
head(c_score_sicker)

##    subject_id hadm_id ELIXHAUSER_overall
## 10       9545   10809          0.5454545
## 15      12049   27692          0.5454545
## 59      29801   33844          0.5454545
## …
```

Saving the results to file: There are several functions that will do this, e.g. write.table() and write.csv(). We will give an example here:

```
write.table(c_score_sicker, file = "c_score_sicker.csv", sep = ";",
row.names = F, col.names = F)
```

If you check in your working directory/folder, you should see the new “c_score_sicker.csv” file.

3.4 Data Reduction

Aim: To reduce or reshape the input data by means of a more effective representation of the dataset without compromising the integrity of the original data. One element of data reduction is eliminating redundant records while preserving needed data, which we will demonstrate in Example Part 1. The other element involves reshaping the dataset into a “tidy” format, which we will demonstrate in below sections.

Examples Part 1: Eliminating Redundant Records

To demonstrate this with a MIMIC database example, we will look at multiple records of non-invasive mean arterial pressure (MAP) for each patient. We will use the records from the following SQL query, which we exported as “mean_arterial_pressure.csv”.

The SQL query selects all the patient subject_id’s and noninvasive mean arterial pressure (MAP) measurements from the mimic2v26.chartevents table on the condition of (1) being an adult, (2) in his/her first ICU admission, and (3) where the hadm_id is not missing according to the mimic2v26.icustay_detail table.

1. 1.

SQL query:

```
SELECT subject_id, value1num
FROM mimic2v26.chartevents
WHERE subject_id IN (
SELECT subject_id
    FROM mimic2v26.icustay_detail
              WHERE subject_icustay_seq = 1
              AND icustay_age_group = 'adult'
              AND hadm_id IS NOT null)
AND itemid=456
AND value1num is not null

-- Export and save the query result as "mean_arterial_pressure.csv"
```

R code:

There are a variety of methods that can be chosen to aggregate records. In this case we will look at averaging multiple MAP records into a single average MAP for each patient. Other options which may be chosen include using the first recorded value, a minimum or maximum value, etc.

For a basic example, the following code demonstrates data reduction by averaging all of the multiple records of MAP into a single record per patient. The code uses the aggregate() function:

```
setwd("~/Desktop/MIMIC_data_files")
all_maps <- read.csv("mean_arterial_pressure.csv", sep = ",")

str(all_maps)

## 'data.frame':    790174 obs. of  2 variables:
##  $ subject_id: int  4 4 4 4 4 4 4 4 3 4 ...
##  $ value1num : num  80.7 71.7 74.3 69 75 ...
```

This step averages the MAP values for each distinct subject_id:

```
avg_maps <- aggregate(all_maps, by=list(all_maps[,1]), FUN=mean,
na.rm=TRUE)

head(avg_maps)

##   Group.1 subject_id value1num
## 1       3          3  75.10417
## 2       4          4  88.64102
## 3       6          6  91.37357
## …
```

Examples Part 2: Reshaping Dataset

Aim: Ideally, we want a “tidy” dataset reorganized in such a way so it follows these 3 rules [2, 3]:

1. 1.

Each variable forms a columnEach observation forms a row

2. 3.

Each value has its own cell

Datasets exported from MIMIC usually are fairly “tidy” already. Therefore, we will construct our own data frame here for ease of demonstration for rule 3. We will also demonstrate how to use some common data tidying packages.

R code: To mirror our own MIMIC dataframe, we construct a dataset with a column of subject_id and a column with a list of diagnoses for the admission.

```
diag <- data.frame(subject_id = 1:6,   diagnosis = c("PNA, CHF", "DKA",
"DKA, UTI", "AF, CHF", "AF", "CHF"))
diag
##   subject_id diagnosis
## 1          1  PNA, CHF
## 2          2       DKA
## 3          3  DKA, UTI
## …
```

Note that the dataset above is not “tidy”. There are multiple categorical variables in column “diagnosis”—breaks “tidy” data rule 1. There are multiple values in column “diagnosis”—breaks “tidy” data rule 3.

There are many ways to “tidy” and reshape this dataset. We will show one way to do this by making use of R packages “splitstackshape” [5] and “tidyr” [4] to make reshaping the dataset easier.

R package example 1—“splitstackshape”:

Installing and loading the package into R console.

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

The function, cSplit(), can split the multiple categorical values in each cell of column “diagnosis” into different columns, “diagnosis_1” and “diagnosis_2”. If the argument, direction, for cSplit() is not specified, then the function splits the original dataset “wide”.

```
diag2 <- cSplit(diag, "diagnosis", ",")
diag2

##    subject_id diagnosis_1 diagnosis_2
## 1:          1         PNA         CHF
## 2:          2         DKA          NA
## 3:          3         DKA         UTI
## …
```

One could possibly keep it as this if one is interested in primary and secondary diagnoses (though it is not strictly “tidy” yet).

Alternatively, if the direction argument is specified as “long”, then cSplit split the function “long” like so:

```
diag3 <- cSplit(diag, "diagnosis", ",", direction = "long")
diag3
##    subject_id diagnosis
## 1:          1       PNA
## 2:          1       CHF
## 3:          2       DKA
## …
```

Note diag3 is still not “tidy” as there are still multiple categorical variables under column diagnosis—but we no longer have multiple values per cell.

R package example 2—“tidyr”:

To further “tidy” the dataset, package “tidyr” is pretty useful.

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

The aim is to split each categorical variable under column, diagnosis, into their own columns with 1 = having the diagnosis and 0 = not having the diagnosis. To do this we first construct a third column, “yes”, that hold all the 1 values initially (because the function we are going use require a value column that correspond with the multiple categories column we want to ‘spread’ out).

```
diag3$yes <- rep(1, nrow(diag3))
diag3

##    subject_id diagnosis yes
## 1:          1       PNA   1
## 2:          1       CHF   1
## 3:          2       DKA   1
## …
```

Then we can use the spread function to split each categorical variables into their own columns. The argument, fill = 0, replaces the missing values.

```
diag4 <- spread(diag3, diagnosis, yes, fill = 0)
diag4

##    subject_id AF CHF DKA PNA UTI
## 1:          1  0   1   0   1   0
## 2:          2  0   0   1   0   0
## 3:          3  0   0   1   0   1
## …
```

One can see that this dataset is now “tidy”, as it follows all three “tidy” data rules.