6
Insights from THREE or More Variables

We all face multiple variables during the decision making process. We are also confident that the more information we have, the better the decision will be; so, we all crave for more rows and columns of data. If you have the good fortune of having "enough" and the "right" data, you need to decide whether you are going to use the data as they are, or you need to reduce their dimensionality.

If you plan to use the data as they are, you will face further challenges in trying to unravel the complexity. And, if you avoid using 3-D images, you will be making several univariate and bivariate plots. Of course, you can make use of different elements, as I will show later, but you need to be extremely careful in what you are trying to communicate.

On the other hand, if you believe that some reduction in dimensionality is necessary, you will produce better visuals. In that situation you will only need univariate and bivariate plots, and maybe some of the ones I will present in this chapter.

Whatever decision you make on this matter, you might need more than one plot to make your case.

6.1 Partitioning with Facets

Let me use the data frame on crimes that we were using at the chapter on bivariate plots. Remember that we subset the data frame `crimeDateDF` into the data frame `crimeDateDF_sub`, and with that data frame I prepared Figure 5.18.

Let me use the original data frame `crimeDateDF` to plot a time series per crime, using points and line patterns, in a descending order based on crime counts:

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

```
> library(ggplot2)
> #base
> basetF = ggplot(crimeDateDF,
+                 aes(x=date,
+                     y=count)) + theme_classic()
> #points
> tspF = basetF + geom_point(alpha=0.1,shape=4)
> tspF = tspF + geom_smooth(fill='grey90',
+                           method = 'loess',
+                           color='white')
> # format for horizontal text
>
> tspF = tspF + scale_x_date(date_labels = "%Y",
+                            date_breaks='2 years')
> tspF = tspF + theme(axis.text.x  = element_text(angle=90,
+                                                 size=7),
+                     axis.text.y  = element_text(size=7),
+                     strip.text = element_text(size = 8))
> # facetting
> tspF1 = tspF + facet_wrap(.~reorder(crime,-count),
+                           ncol=4,
+                           scales = "free_y")
>
```

We have used facets before for Figure 5.9 and Figure 5.19, in those cases we make use of **facet_grid**, but this time I am using **facet_wrap**. They do not work in exactly the same way:

- The ∼ can connect two or more variables, which are categorical. So you can get particular plots, each representing a combination of those variables.
- If you request a *wrap* you will get a subplot for every existent combination of those categorical variables, but if you request a *grid* you will get a subplot for every possible combination, even non-existent ones in the data. Then, a *grid* might produce more subplots, but some might be empty.
- You can use only one variable in a *grid*, but you will get all the subplots in one column (using . ∼) as in Figure 5.19; or in one row as in Figure 5.9 (using ∼ .).[1]
- You can use only one variable in a *wrap* by using . ∼ and *ggplot* will organize the result in several columns. In the code above, I am controlling the amount of columns.[2]
- In the code above, I have used `scales` with `"free_y"`. That will give you subplots each with their own y-axis value range (you can also use `"free_x"` for the same purpose on the horizontal). If you omit that argument, the subplots will share the same range values.

The faceted object `tspF1` is represented in Figure 6.1.

[1] You will notice in a while that *plotnine* in **Python** does not use the dot in either case.
[2] You will notice that *plotnine* in **Python** does not require any symbol before the variable name.

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

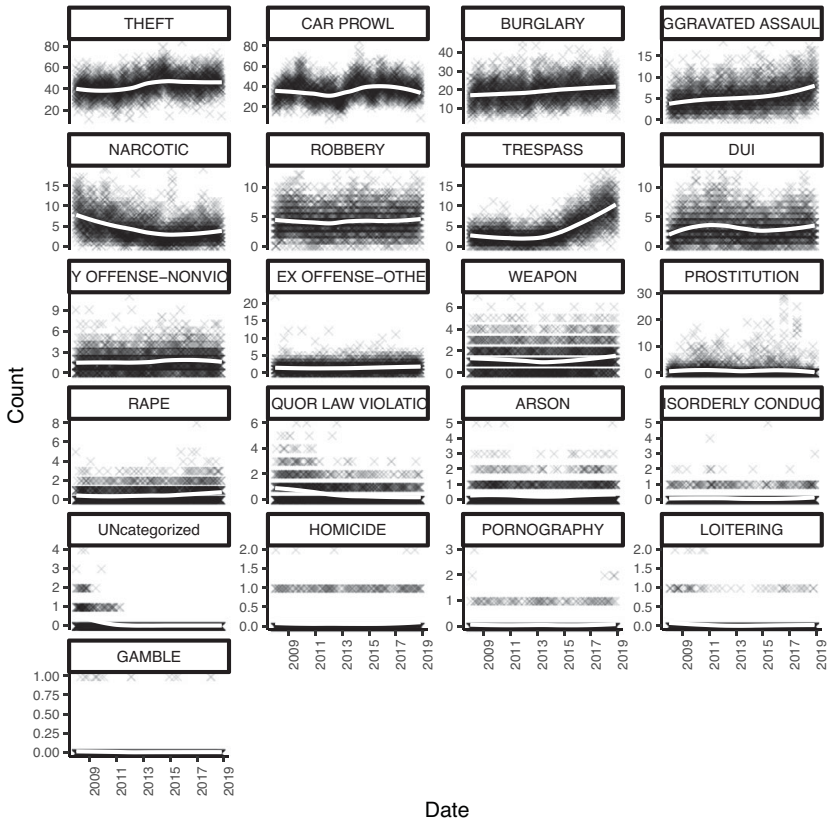

Figure 6.1 Faceted plot with independent axes

The vertical axes is free, so each plot has its own range of values on `y`. Each plot represents a particular crime. Data from Seattle Open Data Portal (City of Seattle, 2019).

The code in **Python** which outputs Figure 6.1 follows. Remember that we turned the variable `crime` into an ordinal. Notice that the code for formatting the dates axis is very similar; and in both languages you can even write '1 years' or '1 year' at `date_breaks`.

```
basetF = ggplot(crimeDateDF, aes(x='date',
                                 y='count')) +theme_classic()
basetF+= geom_point(alpha=0.1,shape='+')

tspF= basetF + geom_smooth(fill='silver',
                           method='loess',
                           alpha=1,
                           color='white')

tspF += scale_x_datetime(date_labels='%Y',
                         date_breaks='2 years')
```

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

```
tspF += theme(axis_text_x  = element_text(angle=90,size=7),
              axis_text_y  = element_text(size=7),
              strip_text = element_text(size = 6))

tspF1 = tspF + facet_wrap('crime',
                          ncol=4,
                          scales="free_y")
```

Also, pay attention to the value of `strip_text` (or `strip.text` in R), which controls the size of the title of each facet. You may need to change some long names in the original data if you plan to have a plot like the one in Figure 6.1. In general, that figure could help you to decide which crimes to focus on.

If you wanted a common vertical axis, as I mentioned before, just change the last line this way:

```
> # facetting
> tspF2 = tspF + facet_wrap(.~reorder(crime,-count),
+                           ncol=4)
```

Yo can do a similar change in **Python** like this:

```
tspF2 = tspF + facet_wrap('crime',
                          ncol=4)
```

6.2 Color Intensity

I have avoided the use of color as much as possible, but in this section I will make use of it. My main purpose is to play with color intensity or saturation. The choice to take advantage of color intensity is the *heatmap*.

Let me use a different data set this time. I will pay attention to the Safe Cities Index from the Economist (The Economist Intelligence Unit, 2017). I will call the data this way:

```
> linkRepo='https://github.com/resourcesbookvisual/data'
> linkCRI='/raw/master/safeCitiesIndexAll.csv'
> fullLink=paste0(linkRepo,linkCRI)
> safe=read.csv(fullLink,stringsAsFactors = FALSE)
```

Take a look at the variable names:

```
> names(safe)
```

```
 [1] "city"                          "D_In_PrivacyPolicy"
 [3] "D_In_AwarenessDigitalThreats" "D_In_PubPrivPartnerships"
 [5] "D_In_TechnologyEmployed"       "D_In_CyberSecurity"
 [7] "D_Out_IdentityTheft"           "D_Out_CompInfected"
 [9] "D_Out_InternetAccess"          "H_In_EnvironmentPolicies"
[11] "H_In_AccessHealthcare"         "H_In_Beds_1000"
[13] "H_In_Doctors_1000"             "H_In_AccessFood"
```

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

```
[15] "H_In_QualityHealthServ"       "H_Out_AirQuality"
[17] "H_Out_WaterQuality"           "H_Out_LifeExpectY"
[19] "H_Out_InfMortality"           "H_Out_CancerMortality"
[21] "H_Out_AttacksBioChemRad"      "I_In_EnforceTransportSafety"
[23] "I_In_PedestrianFriendliness"  "I_In_QualityRoad"
[25] "I_In_QualityElectricity"      "I_In_DisasterManagement"
[27] "I_Out_DeathsDisaster"         "I_Out_VehicularAccidents"
[29] "I_Out_PedestrianDeath"        "I_Out_LiveSlums"
[31] "I_Out_AttacksInfrastructure"  "P_In_PoliceEngage"
[33] "P_In_CommunityPatrol"         "P_In_StreetCrimeData"
[35] "P_In_TechForCrime"            "P_In_PrivateSecurity"
[37] "P_In_GunRegulation"           "P_In_PoliticalStability"
[39] "P_Out_PettyCrime"             "P_Out_ViolentCrime"
[41] "P_Out_OrganisedCrime"         "P_Out_Corruption"
[43] "P_Out_DrugUse"                "P_Out_TerroristAttacks"
[45] "P_Out_SeverityTerrorist"      "P_Out_GenderSafety"
[47] "P_Out_PerceptionSafety"       "P_Out_ThreaTerrorism"
[49] "P_Out_ThreatMilitaryConf"     "P_Out_ThreatCivUnrest"
```

These several variables are telling us information about the safety levels of some cities in the world, and are related to **D**_igital, **H**_ealth, **I**_nfrastructure, and **P**_ersonal_dimensions. For each of these dimensions, there are measures of actions taken (**In**), and results (**Out**). We have 49 variables. A great restriction for this many variables is that they share the same range of values (you will need to re scale the data in case it was not so); so, we are good as each variable ranges, at least theoretically, from zero to one hundred (0–100).

Let me use the heatmap to show you the whole data set, but first verify the *shape* of the data frame:

```
> #just four columns out of fifty
> head(safe[,c(1:4)])
```

```
       city D_In_PrivacyPolicy D_In_AwarenessDigitalThreats
1 Abu Dhabi                 50                         66.7
2 Amsterdam                100                        100.0
3    Athens                 75                        100.0
4   Bangkok                 25                         66.7
5 Barcelona                100                        100.0
6   Beijing                 75                         66.7
  D_In_PubPrivPartnerships
1                       50
2                       50
3                        0
4                        0
5                       50
6                        0
```

The current data frame is in *wide* format, but *ggplot* needs a long format; so let me reshape:

```
> library(reshape)
> safeAllLong=melt(safe, # all the data
+                  id.vars = 'city') #identifier
> head(safeAllLong)
```

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

```
       city           variable value
1 Abu Dhabi D_In_PrivacyPolicy    50
2 Amsterdam D_In_PrivacyPolicy   100
3    Athens D_In_PrivacyPolicy    75
4   Bangkok D_In_PrivacyPolicy    25
5 Barcelona D_In_PrivacyPolicy   100
6   Beijing D_In_PrivacyPolicy    75
```

The function `melt` from the *reshape* library (Wickham, 2018) helped us reshape the file. Notice that I only needed to tell what variables are the identifiers (in this case only `"city"`). Pay attention to the fact that the melting in **R** returned the `variable`, now in long format, as a categorical type:

```
> str(safeAllLong,width = 65,strict.width='cut')
```

```
'data.frame':   2940 obs. of  3 variables:
 $ city    : chr  "Abu Dhabi" "Amsterdam" "Athens" "Bangkok" ...
 $ variable: Factor w/ 49 levels "D_In_PrivacyPolicy",..: 1 1 1..
 $ value   : num  50 100 75 25 100 75 50 100 75 50 ...
```

Python can reshape in a very similar way but the resulting variable will not be categorical. Here is the code using `melt` from *Pandas*:

```
safeAllLong=pd.melt(safe, id_vars=['city'])
```

Let me keep just the variables that measure INPUT indexes. Fortunately, there is a pattern in the variable names, so you can apply some basic *regular expression* recovery technique:

```
> # "grep" will find coincidences and return positions
> positionsIN=grep("_In_", safeAllLong$variable)
> # using those positions to subset
> safeIN=safeAllLong[positionsIN,]
```

The use of `grep` avoided you having to look for the positions of these columns. You can do it manually in data frames with few columns, as we have done before, but with fifty columns I recommend you use regular expressions. You can achieve the same using the function `str.contains` from *Pandas* functions:

```
positionsIN=safeAllLong.variable.str.contains('_In_')
safeIN=safeAllLong[positionsIN]
```

The data frame `safeIN` is almost ready, the problem is that it still has unused categories in **R**:

```
> # the same amount of variables
> length(levels(safeIN$variable))
```

```
[1] 49
```

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

Since we had forty-nine variables in the data frame, the column `variable`, now a categorical, will have the same amount of levels, even though we have filtered half of them. You have two options:

```
> safeIN$variable=droplevels(safeIN$variable)
> # or
> safeIN$variable=as.character(safeIN$variable)
```

Notice that you do not need to apply both of the previous commands simultaneously. The first one will get rid of the unused levels but it will remain a category. The second one can work without the first one, and the variable will not be a category anymore. The purpose of the previous code was the introduction of the `droplevels` command . In **Python**, I do not need to do anything in this case (remember it did not create a category after melting).

Let me go ahead and prepare a heatmap with the next code. The result will be shown in Figure 6.2.

```
> library(ggplot2)
> base = ggplot(data = safeIN, aes(x = variable,
+                                  y = city))
> heat1 = base +  geom_tile(aes(fill = value))
```

The basic commands used to produce the object `heat1` may not be enough, as this kind of visual requires much more work to get a good result. Let me suggest some extra adjustments:

- Reorder columns and rows so that better and worse situations are easier to see. I am going to reorder the rows and columns without using the `reorder` function.
  - Here, I get the indexes (`variable`) sorted by median:

```
> library(magrittr)
> library(dplyr)
> #median per index (variable)
> medVar=safeIN %>%
+     group_by(variable) %>%
+       summarize(the50=median(value))
> # varSorted has the indexes sorted
> varSorted=medVar%>%
+     arrange(the50)%>%as.data.frame()%>%
+     .$variable%>%as.character()
```

  - Here, I get the cities (`city`) sorted by median:

```
> #median per city
> medCity=safeIN %>%
+     group_by(city) %>%
+       summarize(the50=median(value))
> # citySorted has the cities sorted
> citySorted=medCity%>%
+     arrange(the50)%>%as.data.frame()%>%
+     .$city%>%as.character()
```

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

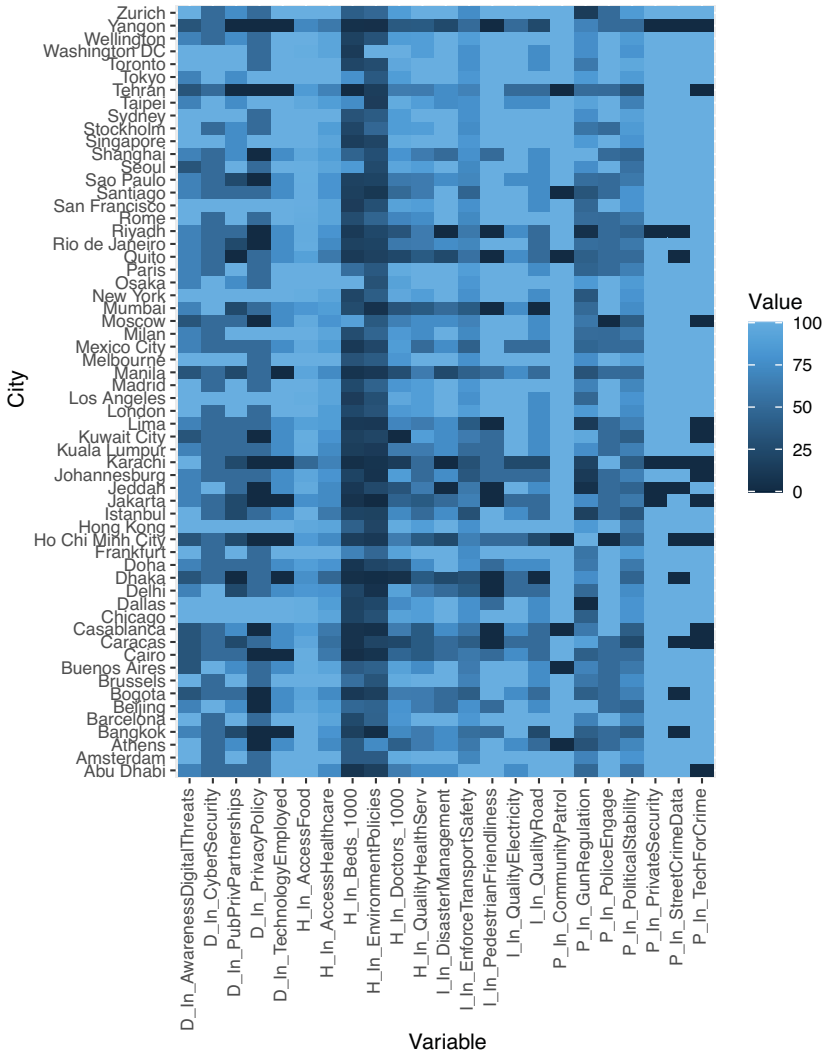

Figure 6.2 Basic heatmap

The legend indicates that the darker the lower the index in a particular city. Data from City Safety Index (The Economist Intelligence Unit, 2017).

Reordering the labels in **Python** can be done this way:

```
medVar=safeIN.groupby('variable').describe()['value'][['50%']]
varSorted=medVar.sort_values(by=['50%'],ascending=True).index

medCity=safeIN.groupby('city').describe()['value'][['50%']]
citySorted=medCity.sort_values(by=['50%'],ascending=True).index
```

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

- Change the color palette to identify central values. In this case, you will need to use the `scale_fill_gradient2` command.
- Make sure texts on the axes are readable. If it is important, you should highlight a particular case.

  - Remember that the current labels of `varSorted` have this text in each one `_In_`, I am going to keep just the text after that string:

```
> library(stringr)
> #splitting each text, keeping second part (right)
> varLabel=str_split(varSorted,pattern = 'In_',simplify = T)[,2]
```

  - If you are interested in highlighting one of the cities, you can prepare an ad hoc set of colors:[3]

```
> colorCity=ifelse(citySorted=='Lima','red','black')
```

  - The previous two steps can be done in **Python** this way:

```
#new labels for ticks
varLabel=[v[1] for v in varSorted.str.split('In_')]

# color for one city
colorCity=['r' if text=='Lima' else 'k' for text in citySorted]
```

Let me build the changes on top of object `heat1`:

```
> #reordering
> heat2 = heat1+ scale_x_discrete(limits=varSorted,labels=varLabel)
> heat2 = heat2+ scale_y_discrete(limits=citySorted)
> #change palette to highlight top, bottom and average
> heat2 = heat2+ scale_fill_gradient2(midpoint = 50,
+                                     mid= 'white',
+                                     low = 'red',
+                                     high = 'darkgreen')
> # Readable text
> heat2 = heat2 + labs(x="",y="")
> heat2 = heat2 + theme(axis.text.x = element_text(angle = 90,
+                                                  hjust = 1))
> # Highlighting one city (possible warning or error)
> heat2 = heat2 + theme(axis.text.y=element_text(colour=colorCity,
+                                                size=6))
```

Figure 6.3 is a good improvement from Figure 6.2.
You can reproduce Figure 6.3 using this code in **Python**:

```
#raw heatplot
base   = ggplot(safeIN, aes(x = 'variable',
                            y ='city'))
heat1  = base +  geom_tile(aes(fill = 'value'))

#reordering
heat2  = heat1 + scale_x_discrete(limits=varSorted,
```

[3] You may get a warning in **R** when you use the vector `colorCity`, if the plot crashes some time in the future verify if avoiding this helps.

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

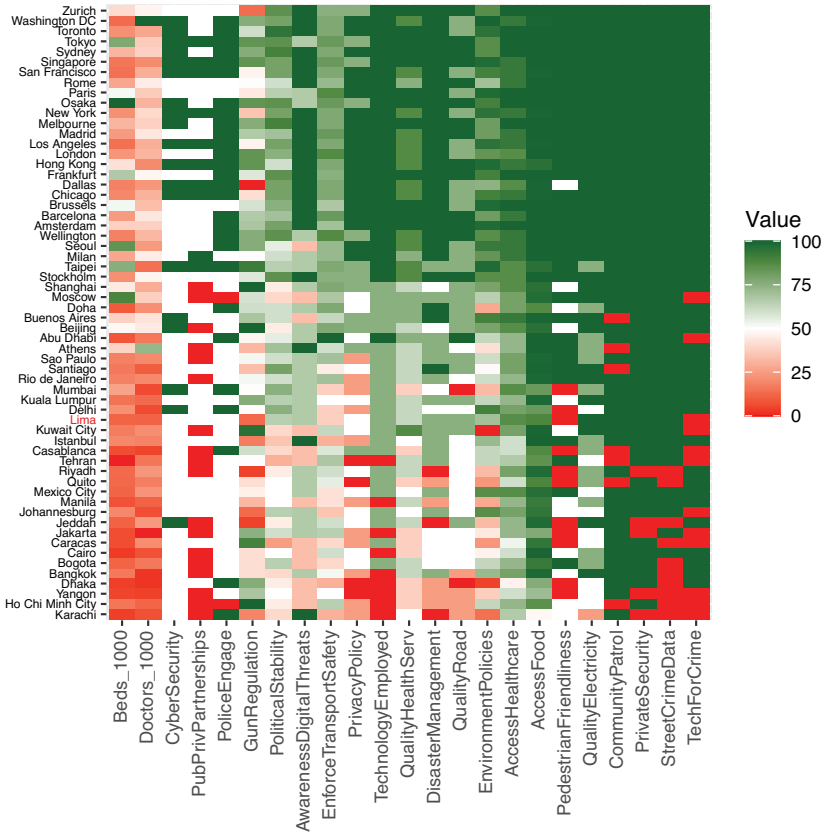

Figure 6.3 Improved heatmap

The palette helps identify different sectors (top, bottom, average). Rows and columns have been ordered; and names of variables (horizontal labels) have been shrunk. Data from City Safety Index (The Economist Intelligence Unit, 2017).

```
                                  labels=varLabel)
heat2 += scale_y_discrete(limits=citySorted)

#change palette to highlight top, bottom and average
heat2 += scale_fill_gradient2(midpoint = 50,
                              mid= 'white',
                              low = 'red',
                              high = 'darkgreen')
# Readable text
heat2 += labs(x="",y="")
heat2 += theme(axis_text_x = element_text(angle = 90,
                                          va = 'top',
                                          size = 6),
               axis_text_y = element_text(size = 6))
```

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

The previous code is not highlighting the city name, *plotnine* needs these last lines using matplotlib to accomplish that:

```
# Highlighting one city
import matplotlib.pyplot as plt

heat2FIG = heat2.draw() # plotnine to matplotlib
ax = heat2FIG.axes[0]   # the plotting area
labels = ax.get_yticklabels() #the city labels
# for every current city label:
for l, c in zip(labels, colorCity):
    l.set_color(c) # change color
```

You should by now realize that the heatmap is trying to use a two dimensional element to display multivariate patterns; so, we started from that basic idea and did our best to make patterns appear and highlight a situation of interest.

6.3 Simultaneity

I will use lines to see the whole behavior of the cases, expecting to find some patterns. The selected visual is the *parallel lines plot*. We used lines in time series data, but this time we do not have time on the horizontal axis but every column; and in the vertical axis the possible values of every column; so the same restriction we had for the heatmap remains here: the range of values for all the variables needs be the same.

Let me do some simple computations to reduce the number of variables (columns for the horizontal). I am going to compute an average (mean) of every city by type (Input/Output) and dimension (Digital, Health, Infrastructure, Personal):

```
> safe$meanDIN=rowMeans(safe[,c(grep("D_In", names(safe) ))])
> safe$meanDOUT=rowMeans(safe[,c(grep("D_Out", names(safe) ))])
> safe$meanHIN=rowMeans(safe[,c(grep("H_In", names(safe) ))])
> safe$meanHOUT=rowMeans(safe[,c(grep("H_Out", names(safe) ))])
> safe$meanIIN=rowMeans(safe[,c(grep("I_In", names(safe) ))])
> safe$meanIOUT=rowMeans(safe[,c(grep("I_Out", names(safe) ))])
> safe$meanPIN=rowMeans(safe[,c(grep("P_In", names(safe) ))])
> safe$meanPOUT=rowMeans(safe[,c(grep("P_Out", names(safe) ))])
```

You can get the row means in **Python** using this code:

```
safe['meanDIN']=safe.filter(regex='D_In').mean(axis=1)
safe['meanDOUT']=safe.filter(regex='D_Out').mean(axis=1)

safe['meanHIN']=safe.filter(regex='H_In').mean(axis=1)
safe['meanHOUT']=safe.filter(regex='H_Out').mean(axis=1)

safe['meanIIN']=safe.filter(regex='I_In').mean(axis=1)
```

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

```
safe['meanIOUT']=safe.filter(regex='I_Out').mean(axis=1)

safe['meanPIN']=safe.filter(regex='P_In').mean(axis=1)
safe['meanPOUT']=safe.filter(regex='P_Out').mean(axis=1)
```

We have increased the number of variables in the *safe* data frame. Let me just keep the columns with the averages of each of the "input" dimensions:

```
> safeINS=safe[,c(grep("IN$|^city", names(safe)))] # ends with
> NewNames=c("city",'DIGITAL','HEALTH','INFRA','PERSON')
> names(safeINS)=NewNames
```

Similarly, in **Python**:

```
safeINS=safe.filter(regex='IN$|^city')
safeINS.columns=["city",'DIGITAL','HEALTH','INFRA','PERSON']
```

As before, I would like to identify some cities from the rest. In this case, let me prepare another column for the `safeINS`; there, I will identify cities whose overall average is higher than 90 (100 is the most they can get):

```
> InValues=c('DIGITAL','HEALTH','INFRA','PERSON')
> safeINS$top=apply(safeINS[,InValues],1,mean)>90
```

The data frame is in wide format, so let me reshape it into a long one:

```
> safeINLongTop = melt(safeINS, id.vars = c('city','top'))
```

The last two steps can be achieved in **Python** like this:

```
InValues=['DIGITAL','HEALTH','INFRA','PERSON']
safeINS['top']=safeINS.loc[:,InValues].mean(axis=1)>90

# To long version
safeINLongTop = pd.melt(safeINS, id_vars = ['city','top'])
```

We can prepare the plot from here:

```
> library(ggrepel)
> #conditions
> conditionColor=ifelse(safeINLongTop$top,'black','grey90')
> conditionLabel=ifelse(safeINLongTop$top,safeINLongTop$city,"")
> # base
> basep1 = ggplot(safeINLongTop, aes(x = variable,
+                                    y = value,
+                                    group = city))
> basep1 = basep1 + theme_classic()
> # parallels using PATH
> para1 = basep1 + geom_path(color=conditionColor)
> # annotating
> para1 = para1 + geom_text_repel(aes(label=conditionLabel),
+                                      size=4)
```

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

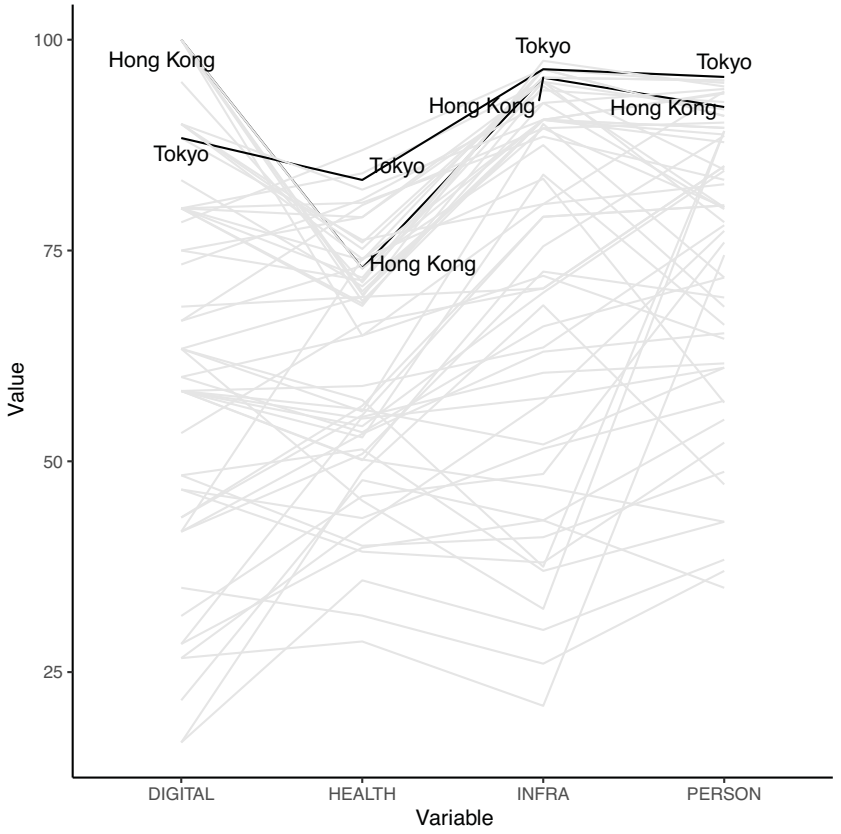

Figure 6.4 Parallel plot

Data from City Safety Index (The Economist Intelligence Unit, 2017).

Figure 6.4 can be easiliy replicated using **Python** like this:

```
import numpy as np

conditionColor=np.where(safeINLongTop['top'],
                    'black',"silver")
conditionLabel=np.where(safeINLongTop['top'],
                   safeINLongTop['city'],"")
basep1 = ggplot(safeINLongTop, aes(x = 'variable',
                                   y = 'value',
                                   group = 'city'))
basep1 += theme_classic()
para11  = basep1 +  geom_path(color=conditionColor)
para11+= geom_text(aes(label=conditionLabel))
```

Since the horizontal is not a date axis, and the column names do not represent a particular order, we can reorder those positions to make some patterns clearer. Let me do that next:

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

```
> #reordering
> newOrder=c("DIGITAL", "INFRA", "PERSON", "HEALTH")
> safeINLongTop$variable=ordered(safeINLongTop$variable,
+                                levels = newOrder)
> #sorting the long data frame
> safeINLongTop=safeINLongTop[order(safeINLongTop$variable),]
```

The same can be accomplished using **Python**:

```
NewOrder=["DIGITAL", "INFRA", "PERSON", "HEALTH"]
safeINLongTop.variable=pd.Categorical(safeINLongTop.variable,
                                      categories=NewOrder,
                                      ordered=True)
safeINLongTop=safeINLongTop.sort_values(by=['variable'])
```

At this point, you may need to reload the data frame and redo some previous steps:

```
> library(ggrepel)
> #reloading
>
> # conditions:
> conditionColor=ifelse(safeINLongTop$top,'black','grey90')
> conditionLabel=ifelse(safeINLongTop$top,safeINLongTop$city,"")
> #base
> basep1b = ggplot(safeINLongTop, aes(x = variable,y = value,
+                                     group = city))
> #theme
> basep1b = basep1b + theme_classic()
> #lines
> para11b = basep1b +  geom_path(color=conditionColor)
> #text
> para11b = para11b + geom_text_repel(aes(label=conditionLabel),
+                                     size=4)
```

The object `para11b` has a different order of columns; you can see that in Figure 6.5.

You can replicate this plot in **Python** (after the `safeINLongTop` is reordered and sorted):

```
# reloading data frame

conditionColor=np.where(safeINLongTop['top'],
                  'black',"silver")
conditionLabel=np.where(safeINLongTop['top'],
                  safeINLongTop['city'],"")
basep1b = ggplot(safeINLongTop, aes(x = 'variable',
                                    y = 'value',
                                    group = 'city'))
basep1b += theme_classic()
para11b  = basep1b +  geom_path(color=conditionColor)
para11b += geom_text(aes(label=conditionLabel))
```

You can plot some cases, as I will do in Figure 6.6, using this code:

```
> basep2 = ggplot(safeINLongTop[safeINLongTop$top,],
+                 aes(x = variable,
+                     y = value, group = city)) + theme_classic()
```

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

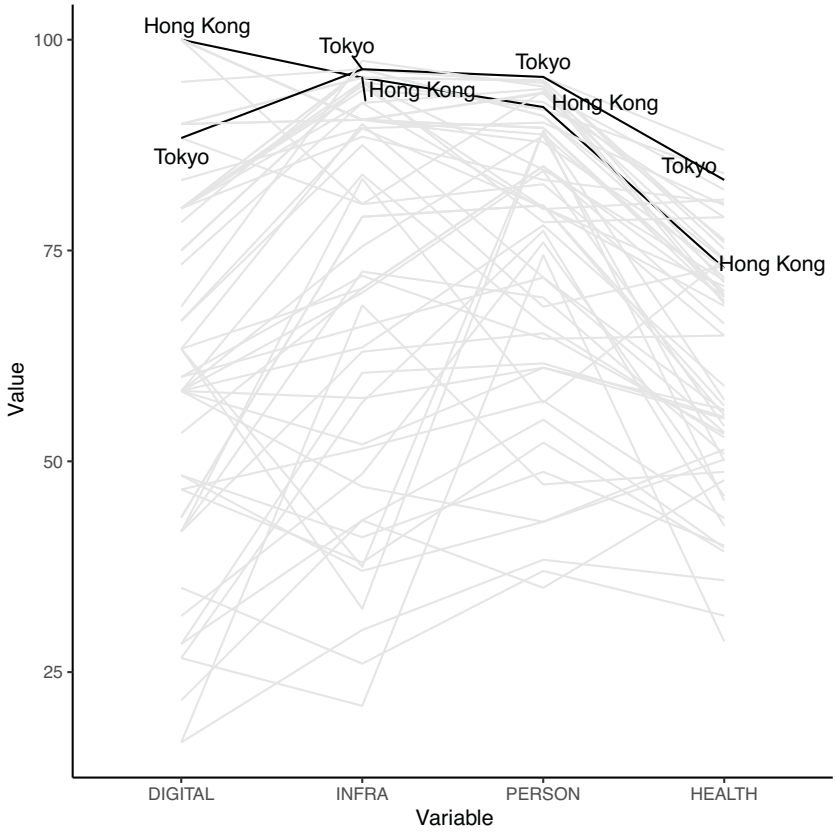

Figure 6.5 Parallel plot for selected cases
Data from City Safety Index (The Economist Intelligence Unit, 2017).

```
> paral2 = basep2 +  geom_path(aes(color=city))
> paral2 = paral2 + theme(legend.position="top",
+                         legend.title.align=0.5)
> paral2 = paral2 +  guides(color=guide_legend(nrow = 1,
+                                    title.position = "top"))
>
```

The **Python** version of Figure 6.6 follows next:

```
basep2 = ggplot(safeINLongTop[safeINLongTop.top],
                aes(x = 'variable',
                    y = 'value',
                    group = 'city')) + theme_classic()

paral2 = basep2 +  geom_path(aes(color='city'))

paral2 += theme(legend_position="top",
                legend_title_align="center")
```

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

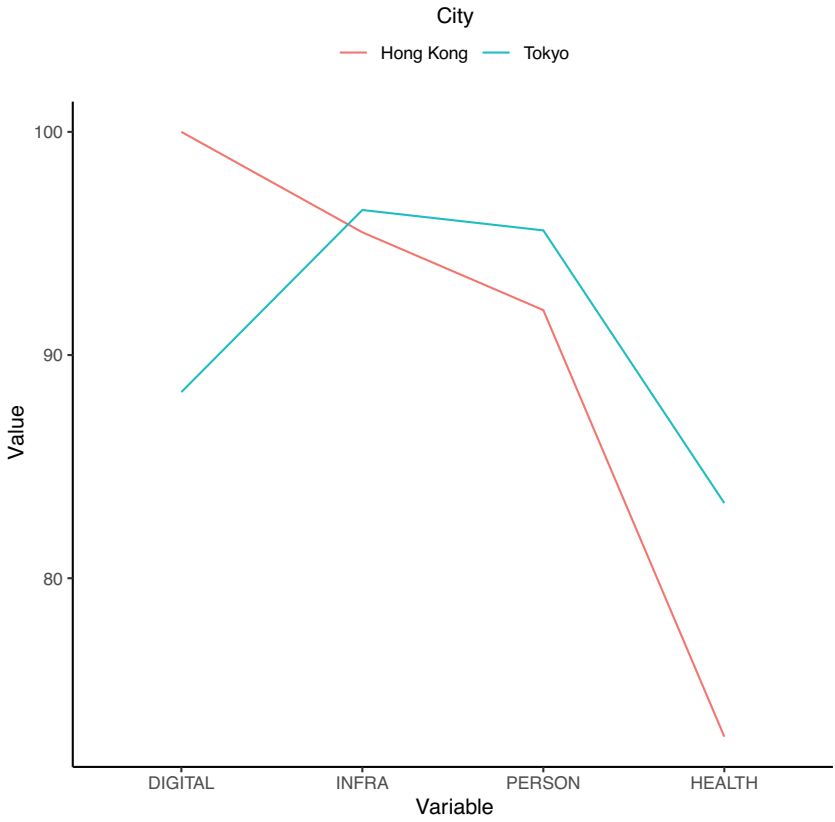

Figure 6.6 Parallel plot for selected cases
Data from City Safety Index (The Economist Intelligence Unit, 2017).

6.4 Area

The previous plot put the variables (columns) in the horizontal axis and you saw the level reached by each case (rows). We could use areas to represent the same. This strategy is called the *radial plot*. Imagine you are wrapping the horizontal and the values will form some polygon. Let me subset `safeINS` into `safeRadarINS` with some cities and without the `top` variable previously created:

```
> cities=c('Abu Dhabi', 'Lima' , 'Zurich','London')
> #safeRadar
> safeRadarINS=safeINS[safeINS$city %in% cities,]
> safeRadarINS$top=NULL
> row.names(safeRadarINS)=NULL
```

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

Let me do the same in **Python**. Notice that in **Python** I will set the index values with the city names:

```
cities=['Abu Dhabi', 'Lima' , 'Zurich','London']
# a copy (avoids changes to original)
safeRadarINS=safeINS.copy()
#index will be the city instead of usual numbers
safeRadarINS.index=safeRadarINS.city
#no need for column city, it is the index,
#and the column 'top' is deleted
safeRadarINS.drop(columns=['city','top'],inplace=True)
#choosing by index value with 'loc' (not 'iloc')
safeRadarINS=safeRadarINS.loc[cities,:]
```

As usual, *ggplot* in **R** needs the data in long format (Python will not need that as I will propose a different strategy):

```
> safeRadarINS_long=melt(safeRadarINS,id.vars = 'city')
```

Now the radar plot:

- Prepare the base layer:

```
> base  = ggplot(safeRadarINS_long,
+                aes(x = variable,
+                    y = value,
+                    group = city))
> base  = base + theme_minimal()
```

- Draw a polygon using polar coordinates, so each value in the variable or index is a vertex or corner of the polygon:

```
> radar = base + geom_polygon(fill = 'gray90',
+                             size=2,
+                             col='black')
> radar = radar + coord_polar()
```

- By default, the radar plots may not use the whole range of possible values, in this case from 0 to 100, so you may want to make sure the whole range is shown:

```
> radar = radar +   scale_y_continuous(limits = c(0,100))
```

- You can customize some elements:

```
> # for the grid and text:
> GridChanges=element_line(size = 0.8,
+                          colour = "grey80")
> TextChanges= element_text(size=10,
+                           color = 'black')
> ### more customization
> radar = radar + theme(panel.grid.major = GridChanges,
+                       axis.text.x =TextChanges)
```

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

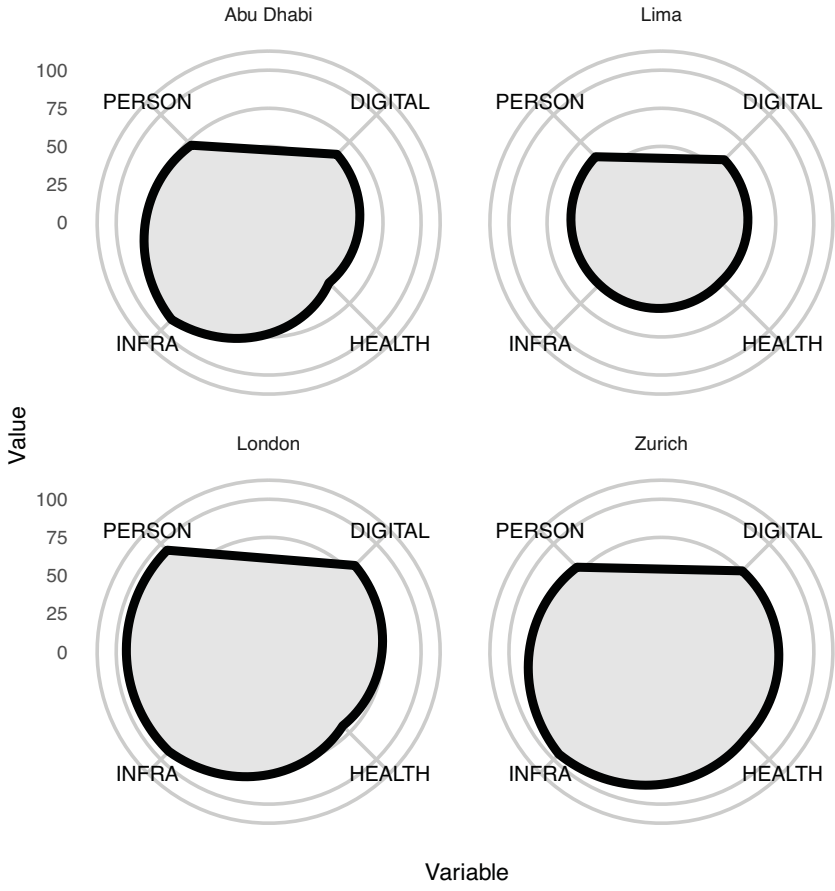

Figure 6.7 Area and Radar plot faceted
Data from City Safety Index (The Economist Intelligence Unit, 2017).

- Finally, you should use facets to plot a radar per city. If you do not use facets, each area will be plotted on top of the other. Use `facet_wrap` to control the amount of columns:

```
> radar = radar + facet_wrap(~city,# one plot per city
+                            ncol = 2)
```

The object `radar` can be seen in Figure 6.7.

Producing a similar plot using *plotnine* is not possible as it lacks `coord_polar` (as of January 2020). So, I decided to offer you a solution in *plotly* (Plotly Technologies Inc., 2015). *Plotly* allows different types of charts

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

to be created but it has a different approach that does not follow the grammar of graphics (it is very declarative, as you will soon see).

Plotly does not require a long format data set, so I can use the current `safeRadarINS` data frame in **Python**. Let me first choose Abu Dhabi, the city in the first row of the data frame `safeRadarINS` (index `0`). Remember that the cities are not a column, but the row name or index:

```
# city Name - index.values[0]= ABU DHABI
currentName=safeRadarINS.index.values[0]
# variableNames
VarNames=safeRadarINS.columns
```

The object `currentName` is simply a string; and the object `VarNames` has a list with all the variable names. Then, you need to use *plotly*, if it is installed, this way:

```
# calling plotly
import plotly.graph_objects as go
from plotly.offline import plot
```

In *ggplot*, we have been creating a *base* layer first, and then adding more layers on top. In *plotly*, we create the **figure**, and then add *traces*; you should understand a trace as a collection of data that defines a visual object. It is actually an element that will be part of the **figure**:

```
fig = go.Figure() # "fig" created, then add info:
fig.add_trace(go.Scatterpolar(
                        #data for each variable
                        r=safeRadarINS.loc[currentName],
                        #variable names
                        theta=VarNames))

fig.update_traces(fill='toself')#radar ends in the beginning
fig.update_layout(title = currentName)
```

The code above also used two "updating" functions: `update_traces` and `update_layout`. As the name implies, the former acts at the trace level and the latter at the **figure** level.

Finally, You need a little piece of code to see the final result. Below I am showing two lines; you need to use only one according to what environment you are using:

```
##for Jupyter-like notebooks
fig.show()

##for Spyder-like environments
plot(fig, auto_open=True)
```

The *plotly* result is represented in Figure 6.8.

From Figure 6.8 you can guess that if you need facets, you need to make a figure as a *grid*, and add a *trace* in each location. Let me guide you.

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

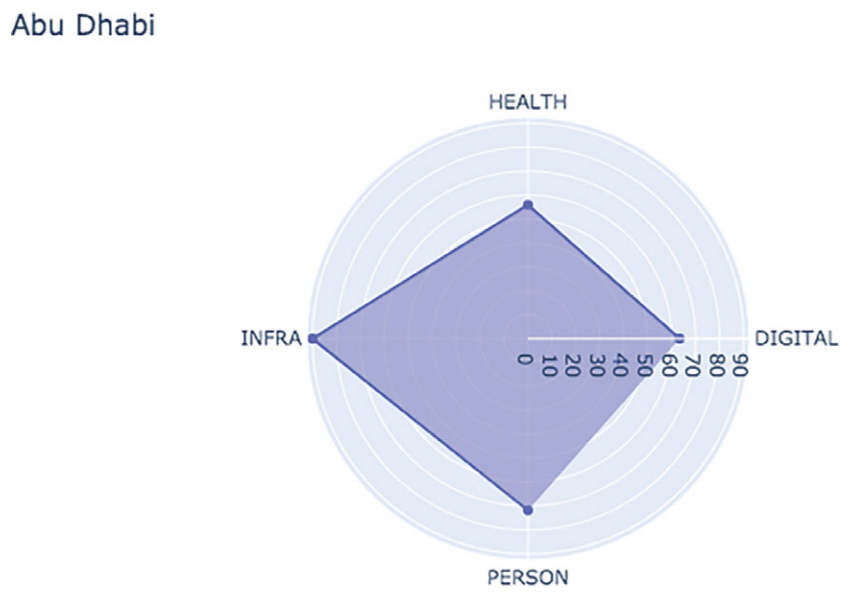

Figure 6.8 Plotly Radar plot
Data from City Safety Index (The Economist Intelligence Unit, 2017).

The first step is to import some libraries:

```
import plotly.graph_objects as go
from plotly.subplots import make_subplots
```

I used the first line before, so you may not need to call it again. The second line is important to create the grid:

```
fig = make_subplots(rows=2,
                    cols=2,
                    specs=[[{'type': 'polar'}]*2]*2)
```

This is a grid with two columns and two rows, and `specs` is setting each as a *polar* element. Now, add the subplots:

```
# the four subplots (traces)
currentName0=safeRadarINS.index.values[0]
fig.add_trace(go.Scatterpolar(
        name = currentName0,
        r = safeRadarINS.loc[currentName0],
        theta = VarNames),
        1, 1) # location of plot (row,columm)

currentName1=safeRadarINS.index.values[1]
fig.add_trace(go.Scatterpolar(
        name = currentName1,
        r = safeRadarINS.loc[currentName1],
        theta = VarNames),
```

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

```
        1, 2)# location of plot

currentName2=safeRadarINS.index.values[2]
fig.add_trace(go.Scatterpolar(
        name = currentName2,
        r = safeRadarINS.loc[currentName2],
        theta = VarNames),
        2, 1)# location of plot

currentName3=safeRadarINS.index.values[3]
fig.add_trace(go.Scatterpolar(
        name = currentName3,
        r = safeRadarINS.loc[currentName3],
        theta = VarNames),
        2, 2)# location of plot

fig.update_traces(fill='toself')
```

The last big piece of code has created a figure with four subplots. However, if you finish there, each radar plot may not use the 0 to 100 range of values (you can try to see if this is the case). So, I need to add some more customization:

```
layoutTrace={'radialaxis' : {'range': [0, 100]}}
fig.update_layout(polar1 = layoutTrace,
                  polar2 = layoutTrace,
                  polar3 = layoutTrace,
                  polar4 = layoutTrace,
                  showlegend=False)
```

Notice that each subplot can be identified (`polar1`, `polar2`, etc.); now, I need to force each one to show the same range. Notice the layout is entered as a *dictionary*. If you want to plot, just use one of the two alternatives shown before (for Jupyter- or Spyder-like environments).

After seeing this process of creating a *plotly* visual, and if you like programming, you might be tempted to use a *loop* to produce the same result as above. I leave the code for you to have fun with (if you do not like programming just ignore it):

```
# libraries needed
from plotly.subplots import make_subplots
import plotly.graph_objects as go
from plotly.offline import plot

# variableNames
VarNames=safeRadarINS.columns
# cityNames
CaseNames=safeRadarINS.index.values

# making up my theme_minimal
layoutTrace={'radialaxis':{'visible':True,
                           'linecolor': 'black',
                           'gridcolor': 'silver',
                           'range' :[0, 100]},
             'angularaxis': {'gridcolor': 'silver',
                             'linecolor':'black'},
             'bgcolor': 'white'} #background
```

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

```
#number of rows and columns
nR=2;nC=2


# producing figure as collection of subplots
fig = make_subplots(rows=nR, cols=nC, #dimensions
                    specs=[[{'type': 'polar'}]*nC]*nR,
                    subplot_titles=CaseNames) #city name on top

# altering shape from linear list
# to list of two lists, each with two elements
CaseNames.shape = (nR,nC)

# each polar element requires a number
NumForPolar=1 # initial number for subplot name

for row in range(1,nR+1):
#do this for each row
    for column in range(1,nC+1):
    #do this for each column (create  subplotplot)
        # get city name
        currentName=CaseNames[row-1,column-1]
        # create a name for the polar
        # this will be: 'polar1','polar2', etc
        polar_name='polar'+str(NumForPolar)
        # creating one subplot
        figINFO=go.Scatterpolar(
                # basic details for Scatterpolar
                r=safeRadarINS.loc[currentName],
                theta=VarNames,
                name=currentName,
                subplot=polar_name) #polar1,polar2, etc
```

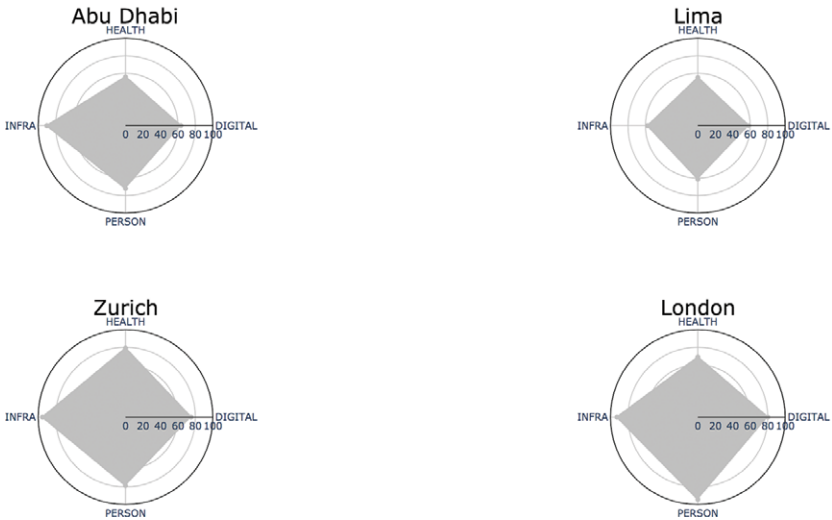

Figure 6.9 Plotly Radar plot in a grid

Data from City Safety Index (The Economist Intelligence Unit, 2017).

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press

```
        #adding a trace with previous info
        fig.add_trace(figINFO,
                      row, column)# location of subplot

        fig.update_layout({polar_name:layoutTrace,
                           'showlegend':False})
        NumForPolar+=1 # number for next polarName


fig.update_traces(fillcolor = 'silver',
                  line_color = 'silver',
                  fill='toself')

fig.update_annotations({'font':{'size':25,
                                'color':'black'},
                        "xanchor": "center",
                        "yanchor": "bottom",
                        "yref": "paper",
                        "borderpad":12})

# choose one:
## jupyter
fig.show()
## spyder
plot(fig, auto_open=True)
```

The final result of this last code, shown in Figure 6.9, will be the closest you get to Figure 6.7 created in **R**.

https://doi.org/10.1017/9781108625425.008 Published online by Cambridge University Press