5

Insights from TWO Variables

We analyze two variables to find out if there might be some kind of relationship between them. This is a step needed in social and policy analysis to identify possible candidates of association or cause–effect relationships. As before, the nature of the data types allows for some particular analytical techniques; that is, the visual will depend on the data type of the two variables of under exploration.

I will continue to use the data on crime we saw in Chapter 4. Allow me to reload it again:

```
> # opening file
> linkRepo='https://github.com/resourcesbookvisual/data/'
> linkCRI='raw/master/crime.csv'
> fullLink=paste0(linkRepo,linkCRI)
> crime=read.csv(fullLink,stringsAsFactors = F)
```

Let me see what data types we have:

```
> str(crime,width = 60,strict.width='cut')
```

```
'data.frame':   499698 obs. of  17 variables:
 $ ReportNumber              : num  2.01e+13 2.01e+13 2.01..
 $ OccurredDate              : chr  "2013-07-09" "2013-07"..
 $ year                      : int  2013 2013 2013 2013 20..
 $ month                     : int  7 7 7 7 7 7 7 7 7 7 ...
 $ weekday                   : chr  "Tuesday" "Tuesday" ""..
 $ OccurredTime              : int  1930 1917 1900 1900 18..
 $ OccurredDayTime           : chr  "evening" "evening" ""..
 $ ReportedDate              : chr  "2013-07-10" "2013-07"..
 $ ReportedTime              : int  1722 2052 35 1258 1846..
 $ DaysToReport              : int  1 0 1 1 0 0 0 0 1 0 ...
 $ crimecat                  : chr  "NARCOTIC" "BURGLARY""..
 $ CrimeSubcategory          : chr  "NARCOTIC" "BURGLARY-"..
 $ PrimaryOffense.Description: chr  "NARC-FRAUD-PRESCRIPT"..
 $ Precinct                  : chr  "NORTH" "NORTH" "SOUT"..
 $ Sector                    : chr  "U" "L" "R" "U" ...
 $ Beat                      : chr  "U3" "L3" "R2" "U3" ...
 $ Neighborhood              : chr  "SANDPOINT" "LAKECITY"..
```


https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

Based on what we have, let me present the type of plots you can use to reveal the possible relationships between two variables.

5.1 Cat–Cat Relationships

If you consider two categorical variables that are related, you need to first prepare a **contingency table** or **crosstab**.

```
> #contingency table of counts
> (PrecintDaytime=table(crime$Precinct,crime$OccurredDayTime))
```

```
           afternoon   day evening night
EAST           20774 15976   17380 19880
NORTH          48754 33744   39867 37942
SOUTH          22147 17322   16240 15497
SOUTHWEST      14221 10595   11169 11034
WEST           48931 30366   33766 30925
```

The contingency table shows the concurrent counts for every category. We can get the total and marginal counts using `addmargins`:

```
> #sum per rows and columns
> addmargins(PrecintDaytime)
```

```
           afternoon    day evening  night    Sum
EAST           20774  15976   17380  19880  74010
NORTH          48754  33744   39867  37942 160307
SOUTH          22147  17322   16240  15497  71206
SOUTHWEST      14221  10595   11169  11034  47019
WEST           48931  30366   33766  30925 143988
Sum           154827 108003  118422 115278 496530
```

Keep in mind that when a table tries to hypothesize a relationship, you should have the *independent* variable in the columns, and the *dependent* one in the rows; then, the percent should be calculated by column, to see how the levels of the dependent variable vary by each level of the independent one, and compare along rows:

```
> #marginal per column (column adds to 1)
> (PrecDayti_mgCol=prop.table(PrecintDaytime,
+                             margin = 2))
```

```
            afternoon        day    evening      night
EAST      0.13417556 0.14792182 0.14676327 0.17245268
NORTH     0.31489340 0.31243577 0.33665197 0.32913479
SOUTH     0.14304353 0.16038443 0.13713668 0.13443155
SOUTHWEST 0.09185090 0.09809913 0.09431525 0.09571644
WEST      0.31603661 0.28115886 0.28513283 0.26826454
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

Let me use the object `PrecDayti_mgCol` to prepare the input for *ggplot*:

```
> #making a data frame from contingency table
> PrecDaytiDF=as.data.frame(PrecintDaytime)
> names(PrecDaytiDF)=c("precinct","daytime","counts")
> #adding marginal columns percents:
> PrecDaytiDF$pctCol=as.data.frame(PrecDayti_mgCol)[,3]
> # we have:
> PrecDaytiDF # see result below
```

```
    precinct   daytime counts     pctCol
1       EAST afternoon  20774 0.13417556
2      NORTH afternoon  48754 0.31489340
3      SOUTH afternoon  22147 0.14304353
4  SOUTHWEST afternoon  14221 0.09185090
5       WEST afternoon  48931 0.31603661
6       EAST       day  15976 0.14792182
7      NORTH       day  33744 0.31243577
8      SOUTH       day  17322 0.16038443
9  SOUTHWEST       day  10595 0.09809913
10      WEST       day  30366 0.28115886
11      EAST   evening  17380 0.14676327
12     NORTH   evening  39867 0.33665197
13     SOUTH   evening  16240 0.13713668
14 SOUTHWEST   evening  11169 0.09431525
15      WEST   evening  33766 0.28513283
16      EAST     night  19880 0.17245268
17     NORTH     night  37942 0.32913479
18     SOUTH     night  15497 0.13443155
19 SOUTHWEST     night  11034 0.09571644
20      WEST     night  30925 0.26826454
```

Notice that the data types may need some formatting:

```
> summary(PrecDaytiDF)
```

```
      precinct      daytime      counts          pctCol
 EAST     :4   afternoon:5   Min.   :10595   Min.   :0.09185
 NORTH    :4   day      :5   1st Qu.:15856   1st Qu.:0.13437
 SOUTH    :4   evening  :5   Median :20327   Median :0.15415
 SOUTHWEST:4   night    :5   Mean   :24826   Mean   :0.20000
 WEST     :4                 3rd Qu.:33750   3rd Qu.:0.29196
                             Max.   :48931   Max.   :0.33665
```

The column `daytime` should be formatted as ordinal:

```
> # reformatting ordinal data
> RightOrder=c("day","afternoon","evening","night")
> PrecDaytiDF$daytime=ordered(PrecDaytiDF$daytime,
+                             levels=RightOrder)
```

5.1.1 Within-/Between-Group Differences

When we want to represent the counts we have in the contingency table, we can use the **grouped barplot** or **dodged barchart** :

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

```
> library(ggplot2)
> base1=ggplot(data=PrecDaytiDF,
+              aes(x=precinct,
+                  y=counts,
+                  fill=daytime)) + theme_classic()
> barDodge1= base1 +  scale_fill_brewer(palette = "Greys")
> barDodge1= barDodge1 + geom_bar(stat="identity",
+                                 position="dodge", # DODGE
+                                 color='grey') #border of bar
```

In this grouped barplot, the precinct serves as the grouping variable, so you can visualize the particular behavior of the daytime categories *within* each precinct. At the same time, you can analyze how a particular level of the daytime categories behaves *between* precincts. The default grouped bar chart is shown in Figure 5.1, using the *Greys* sequential color palette (Brewer, 2009).

If the default plot is not facilitating that within-/between-group exploration, you may need to make some changes. For instance, let me this time change the order of `precincts` in ascending order of the minimal count of `daytime`:

```
> minMargiPrecint=apply(PrecintDaytime,1,min)
> sortedMinPrecint=sort(minMargiPrecint)
> sortedMinPrecint
```

```
SOUTHWEST     SOUTH      EAST      WEST     NORTH
    10595     15497     15976     30366     33744
```

I can use that precinct order to change the horizontal axis as we have done before:

```
> # improved values for horizontal axis
> newHorizontals=names(sortedMinPrecint)
> newHorizontals
```

```
[1] "SOUTHWEST" "SOUTH"     "EAST"      "WEST"      "NORTH"
```

However, this time, I will use the function `reorder`[1].

```
> base2=ggplot(data=PrecDaytiDF,
+              # using reorder
+              aes(x=reorder(precinct,counts,FUN = min),
+                  y=counts,
+                  fill=daytime)) + theme_classic()
> base2 = base2 + labs(x="precinct") # not needed in Python
> barDodge2= base2 +  scale_fill_brewer(palette = "Greys")
> barDodge2= barDodge2 + geom_bar(stat="identity",
+                                 position="dodge", # DODGE
+                                 color='grey')
```

[1] Notice that every time I use `reorder` in **R**, I need to change the axis title explicitly. I will not need to do that in **Python** as I will enter the order of the levels in the same way I did before (using the argument `limits` in the scale).

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

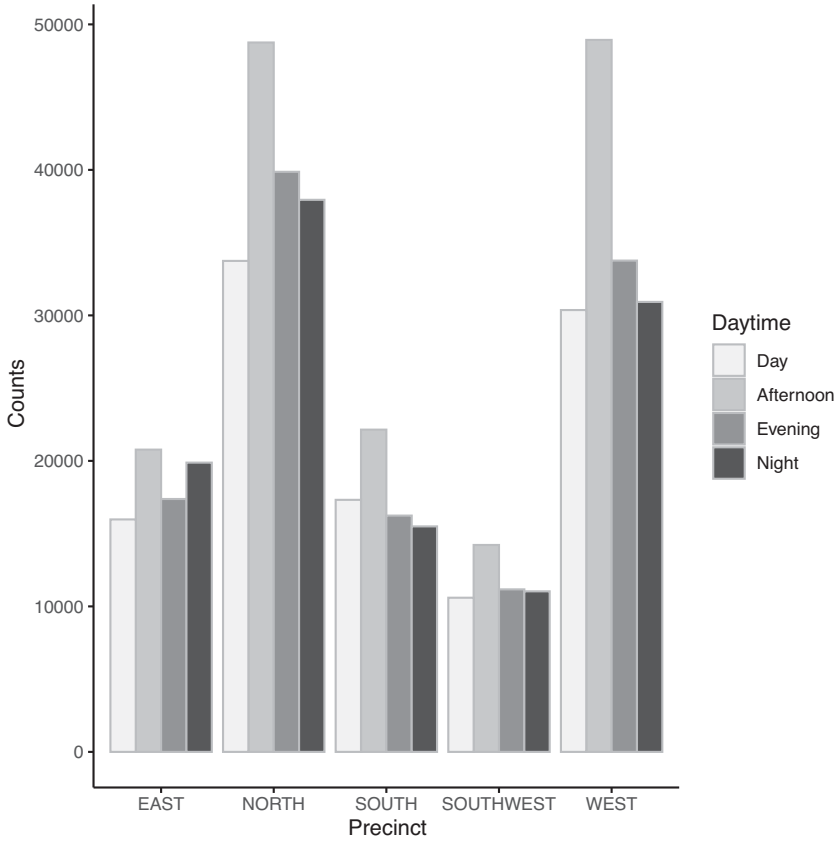

Figure 5.1 Grouped barplot (default)
Data from Seattle Open Data Portal (City of Seattle, 2019).

You can see the result in Figure 5.2.

If you wanted to add the counts on top of the bars, you could use this code:

```
> barDodge2= barDodge2 + geom_text(aes(label=counts),
+                                  angle=0,
+                                  vjust=0,
+                                  hjust=0.5,
+                          position = position_dodge(width =0.9))
```

The code above used the command `position_dodge`. You need this command to distribute the labels along the bars. If you omit it, all the labels will appear one on over the other along in each group. You can alter the `width` parameter (currently 0.9) as needed.

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

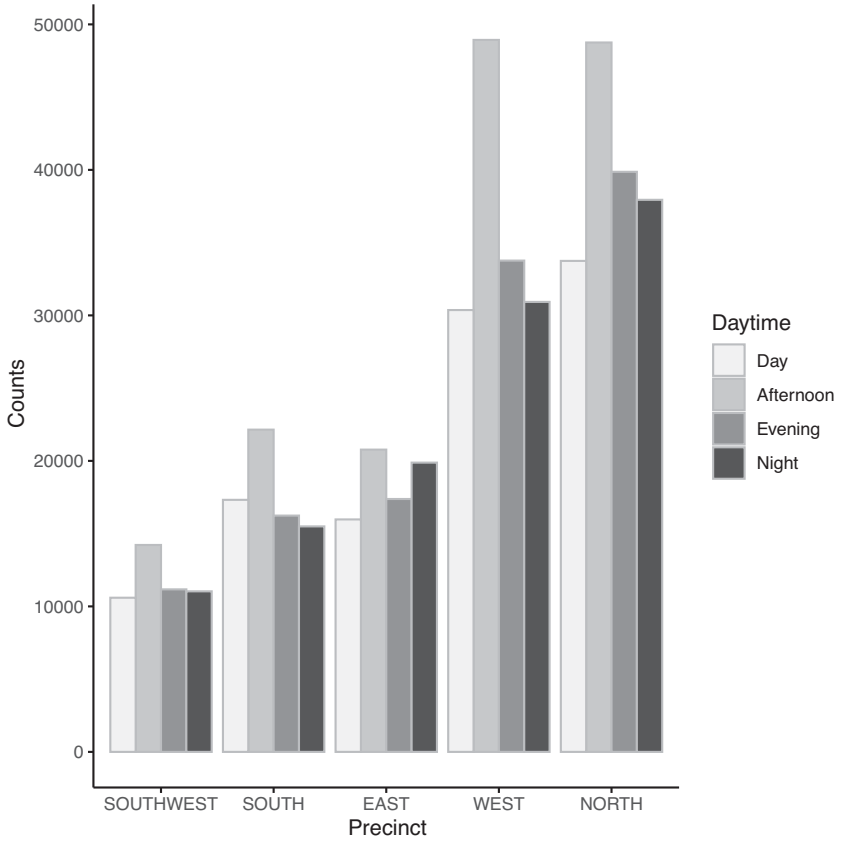

Figure 5.2 Grouped barplot (improved)

Groups have reorganized in an ascending fashion using command `reorder`. Data from Seattle Open Data Portal (City of Seattle, 2019).

Let me prepare my **Python** version. As always, I get the data:

```
import pandas as pd

#link to data
linkRepo='https://github.com/resourcesbookvisual/data/'
linkFile='raw/master/crime.csv'
fullLink=linkRepo+linkFile
crime=pd.read_csv(fullLink)
```

Once the data are available I can prepare the contingency table for *Pandas* and *plotnine*:

```
#contingency table of counts
PrecintDaytime=pd.crosstab(crime.Precinct,crime.OccurredDayTime)
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

```
#marginal per column (column adds to 1)
PrecDayti_mgCol=pd.crosstab(crime.Precinct,crime.OccurredDayTime,
                            normalize='columns')

#making a data frame from contingency table
PrecDaytiDF=PrecintDaytime.stack().reset_index()
PrecDaytiDF.columns=["precint","daytime","counts"]

#adding marginal columns percents:
PrecDaytiDF['pctCol']=PrecDayti_mgCol.stack().reset_index().iloc[:,2]

# reformatting ordinal data
RightOrder=["day","afternoon","evening","night"]
PrecDaytiDF.daytime=pd.Categorical(PrecDaytiDF.daytime,
           categories=RightOrder,ordered=True)
```

Notice that `pd.crosstab` brings a result comparable with `table` in **R**. However, while **R** only requires one function to turn the table into a data frame (`as.data.frame`), *Pandas* needs a couple: `stack` and `reset_index`. The crosstab has a wide format[2]; but *plotnine* needs a long (or stacked) format, see p. 19. The `stack` function will turn "wide" data frames into "long" ones. However, `stack` will turn the `Precinct` column into the data frame index (or "row names" as it is called in **R**); then, the function `reset_index` will number each row and send the values in the index as a new column.

In **Python**, I cannot use a command like `reorder` in **R** inside *plotnine*, so I need to keep the order of precincts from the contingency tab:

```
# improved values for horizontal axis
minMargiPrecint=PrecintDaytime.apply(min,axis=1)
sortedPrecint=minMargiPrecint.sort_values(ascending=True)
newPrecintAxis=sortedPrecint.index
```

Then, I can replicate Figure 5.2, including labels for each bar with this code:

```
from plotnine import *

base2= ggplot(PrecDaytiDF,
              aes(x='precint',y='counts',
                  fill='daytime')) + theme_classic()

barDodge2 = base2 +  geom_bar(stat="identity",
                              position="dodge",
                              color='grey')

barDodge2 += scale_fill_brewer(palette = "Greys")

barDodge2 += scale_x_discrete(limits=newPrecintAxis)

barDodge2 += geom_text(aes(label='counts'),angle=0,
                       va='bottom',ha='center',
                       position=position_dodge(width=0.9))
```

[2] see Section 2.3.1.

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

Notice that in the previous code, I started using the symbol **+=**, which simpifies the code a little. This cannot be used in **R**.

5.1.2 Whole-Part Differences

The **stacked barplot** is the most familiar option to highlight the difference that each category makes to another category. In this situation, we can use the same counts we had before. This time, let me draw the bars in a descending fashion.

```
> base3= ggplot(data=PrecDaytiDF,
+               aes(x=reorder(precinct,-counts,FUN=max), #- counts
+                   y=counts,
+                   fill=daytime)) + theme_classic()
> base3 = base3 + labs(x="precinct") # not needed in Python
> barStacked1 = base3 + scale_fill_brewer(palette = "Greys")
> barStacked1 = barStacked1 + geom_bar(stat = "identity",
+                                      color='grey') # no position
>                                                  # stack is default
```

Object *barStacked1* can be seen in Figure 5.3.

If you need to change the order of the colors in Figure 5.3, you need to do a couple of things:

- Create a copy of daytime, but with an inverse ordering

```
> # inversed copy
> PrecDaytiDF$daytime2 = factor(PrecDaytiDF$daytime,
+                       levels = rev(levels(PrecDaytiDF$daytime)))
```

- Use that new variable in the `fill` aesthetics:

```
> base3= ggplot(data=PrecDaytiDF,
+               aes(x=reorder(precinct,-counts,FUN=max),
+                   y=counts,
+                   fill=daytime2)) + theme_classic()
> base3 = base3 + labs(x="precinct") # not needed in Python
```

- Add `direction` in the color palette with the value **-1**. And no more changes.

```
> barStacked2 = base3 + scale_fill_brewer(palette = "Greys",
+                                         direction = -1)
> # no changes here:
> barStacked2 = barStacked2 + geom_bar(stat = "identity",
+                                      color='grey')
```

The result can be seen in Figure 5.4.

If you want to annotate the bars, you have to do it within them. In that situation you need to use the `position_stack` parameter in the `geom_text`:

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

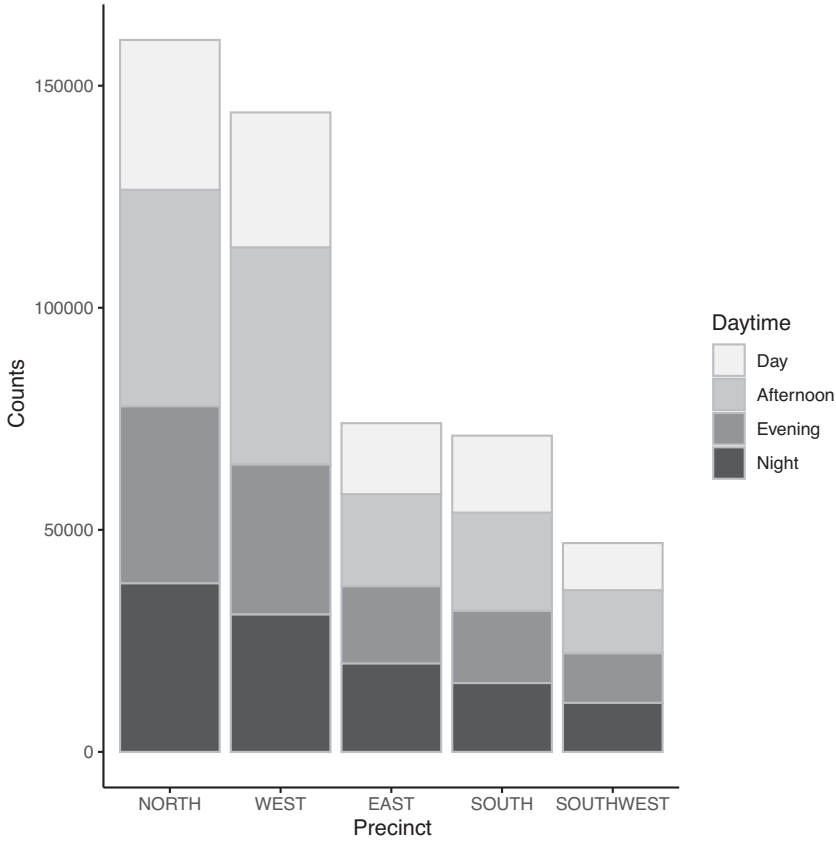

Figure 5.3 Stacked barplot

Groups have been reorganized in a descending fashion using command `reorder`. Data from Seattle Open Data Portal (City of Seattle, 2019).

```
> barStacked2b= barStacked2 + geom_text(aes(label=counts),
+                                       size = 3,
+                                       color='black',
+                              position = position_stack(vjust = 0.5))
```

However, you might have a hard time reading some of the texts because of the lack of contrast, as you can see in Figure 5.5.

A possible solution is to create a set of colors that deal with that:

```
> # ad-hoc set of colors
> adHoc=c('white','white','black','black')
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

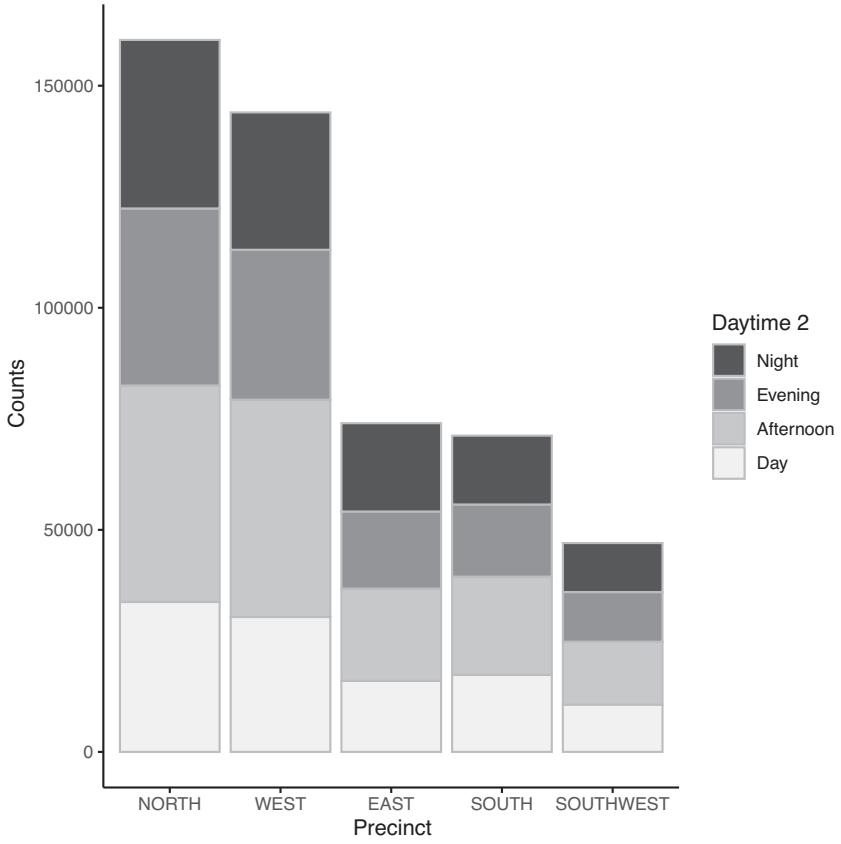

Figure 5.4 Stacked barplot recolored

Colors have been reorganized by changing the palette order and the order of the ordinal variable `daytime`. Data from Seattle Open Data Portal (City of Seattle, 2019).

Then, use those values for the label color:

```
> # annotating with color (default color will be assigned)
> barStacked2c= barStacked2 + geom_text(aes(label=counts,
+                                           color=daytime2),
+                                       size = 3,
+                             position = position_stack(vjust = 0.5))
> # customized colors
> barStacked2c= barStacked2c + scale_colour_manual(values = adHoc)
> # use this to avoid text over legend symbols
> barStacked2c= barStacked2c + guides(color=FALSE)
```

Figure 5.6 shows you the result of the previous code.

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

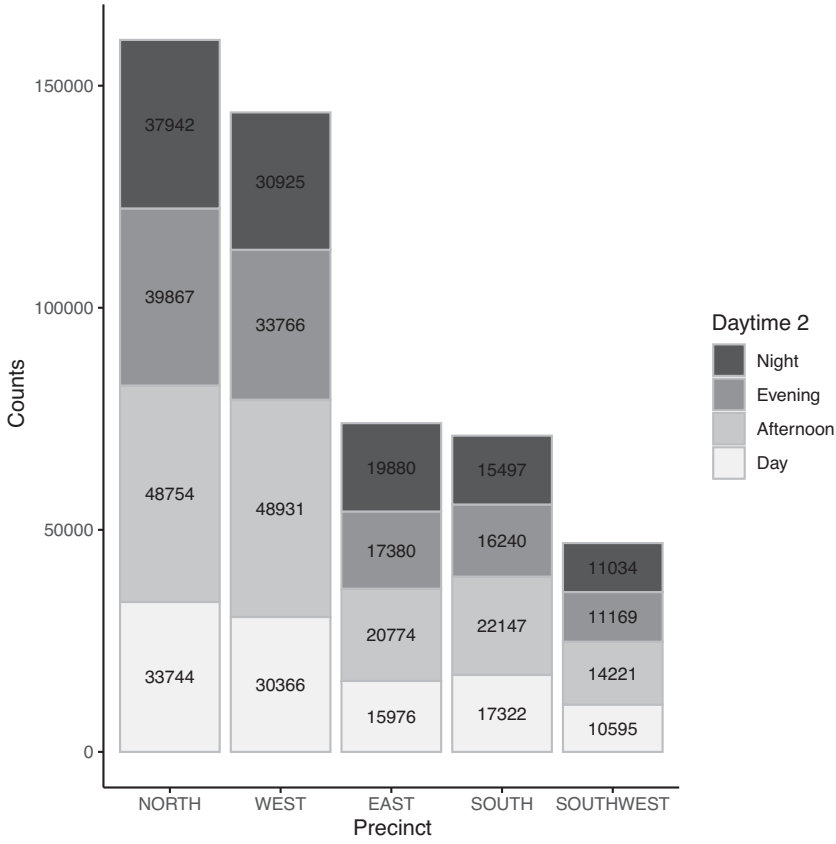

Figure 5.5 Stacked barplot annotated

Count annotations are difficult to see in some bars. Data from Seattle Open Data Portal (City of Seattle, 2019).

Notice the trick of coloring the text using aesthetics, to later assign color manually. This will have an annoying result: a letter `a` will appear over each legend symbol. I avoided that with the last line (`color=FALSE`).

You can follow similar steps to reproduce Figure 5.6 in Python:

- Change order of groups:

```
# improved values for horizontal axis
minMargiPrecint=PrecintDaytime.apply(min,axis=1)
sortedPrecint2=minMargiPrecint.sort_values(ascending=False)
newPrecintAxis2=sortedPrecint2.index
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

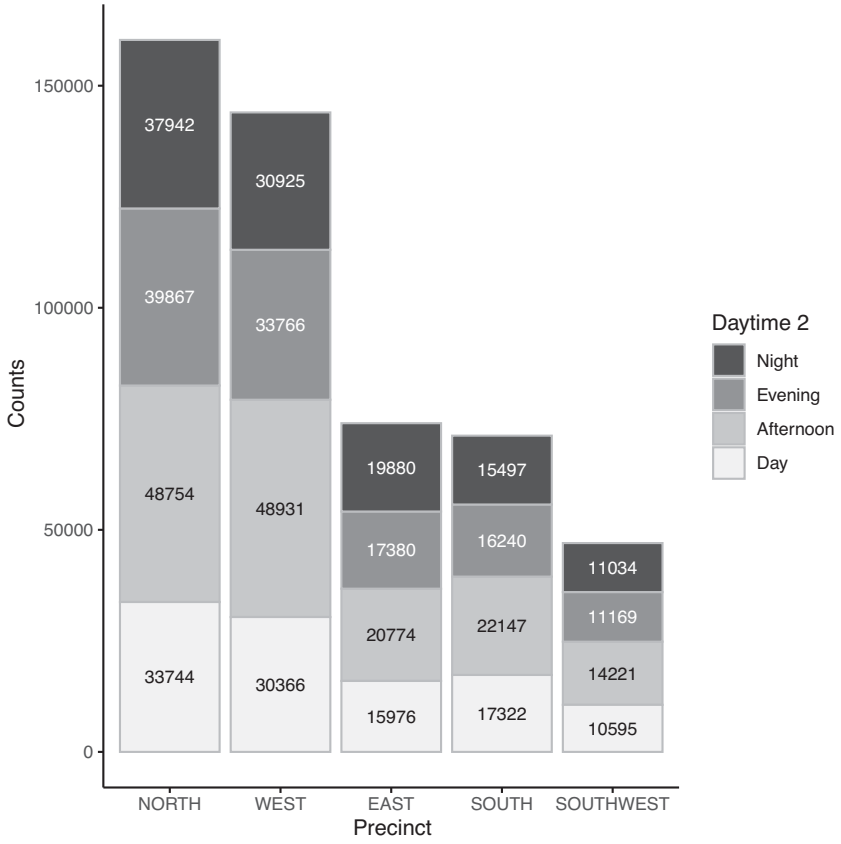

Figure 5.6 Stacked barplot with improved annotations

Count annotations are easier to see in comparison to Figure 5.5. Data from Seattle Open Data Portal (City of Seattle, 2019).

- Create copy of `daytime`:

```
# inversed copy
PrecDaytiDF['daytime2']=pd.Categorical(PrecDaytiDF.daytime,
                            categories=RightOrder[::-1],
                            ordered=True)
```

- Create a set of colors. Notice that I can not reverse the Brewer palette in *plotnine*, so I just call the amount of brewer colors I need from the *Greys* palette in `mizani`[3] (Kibirige, 2020a) to input them manually.

```
# ad-hoc set of colors
adHoc=['white','white','black','black']
```

[3] This library should be already among your libraries, as *plotnine* requires it.

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

```
# manual colors according to Brewer.
from mizani.palettes import brewer_pal
Greys4=brewer_pal(palette='Greys')(4)[::-1]
```

- Add this to the previous code to reproduce Figure 5.6:

```
base3= ggplot(PrecDaytiDF,
              aes(x='precint', y='counts',
                  fill='daytime2')) + theme_classic()
# order of horizontal
barStacked2c = base3 +scale_x_discrete(limits=newPrecintAxis2)
# manual Brewer palette
barStacked2c += scale_fill_manual(values=Greys4)
# usual
barStacked2c += geom_bar(stat="identity", color='grey')
# annotating with color (default color will be assigned)
barStacked2c += geom_text(aes(label='counts',
                              color='daytime2'),
                          size = 8,
                          position = position_stack(vjust = 0.5))
# customized colors
barStacked2c += scale_colour_manual(values = adHoc)
# use this to avoid text on top of legend symbols
barStacked2c += guides(color=False) # want to omit?
```

Keep in mind that the stacked barplot is not good at making between-group comparisons; so it is not a good choice if you plan to reveal trends.

5.1.3 Relative Contribution

The bars in the previous cases represented counts. As we mentioned at the begining of Section 5.1 we first prepared a contingency table and took it from there, so any of the previous plots could have changed what goes into a row or a column. In this subsection, however, we should have decided clearly what will be the row and the column variable, as we will use *marginal* shares. Take a look again at the marginals I am computing with totals per row and column:

```
> addmargins(PrecDayti_mgCol)
```

```
             afternoon        day    evening      night        Sum
EAST       0.13417556 0.14792182 0.14676327 0.17245268 0.60131333
NORTH      0.31489340 0.31243577 0.33665197 0.32913479 1.29311592
SOUTH      0.14304353 0.16038443 0.13713668 0.13443155 0.57499619
SOUTHWEST  0.09185090 0.09809913 0.09431525 0.09571644 0.37998172
WEST       0.31603661 0.28115886 0.28513283 0.26826454 1.15059284

Sum        1.00000000 1.00000000 1.00000000 1.00000000 4.00000000
```

That information is already in the `PrecDaytiDF` data frame which we have been using for the previous plots, and since the percentage stacked bar

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

represents the relative contribution of a category to a level of the other, each of our bars must represent 100 percent. There will be several instructions that have changed, so let me guide you:

- The new base information will need different variables in the aesthetics:

```
> base4=ggplot(data=PrecDaytiDF,
+              aes(x=daytime, # changes in aes!
+                  y=pctCol,
+                  fill=precinct)) + theme_classic()
```

- Since the horizontal is `daytime`, the `precinct` bars will represent the contribution. Notice that `precinct` is a nominal variable, then we should look for a qualitative palette. I will choose `Paired` as my palette. This palette is the only safe palette for color-blindedness (Brewer, 2009), and it is also safe for printing (not for photocopying) if you have four levels; so it is safe for our case.[4]

```
> barStPct1= base4 + scale_fill_brewer(type='qual', #not needed
+                                      palette = 'Paired')
```

  Notice that if I write the palette name, I do not need to specify the type.
- By now, you should have realized that the vertical axis is not needed (it was not needed in other previous plots). Let me remind you how to get rid of it:

```
> barStPct1= barStPct1 + theme(axis.title.y = element_blank(),
+                              axis.text.y  = element_blank(),
+                              axis.line.y  = element_blank(),
+                              axis.ticks.y  = element_blank())
```

- Finally, just set the position to `fill` in the `geom_bar`:

```
> barStPct1= barStPct1 +  geom_bar(stat = "identity",
+                                  position="fill") # you need this
```

Figure 5.7 shows you the result of plotting the `barStPct1` object.

Annotating the bars requires transforming the values, currently from 0 to 1, into percentages, and also adding the percent symbol. In this situation, *ggplot* needs the help of the library *scales* (Wickham et al., 2019b).

```
> library(scales) # for labelling
>                          #label in % with ONE decimal position
> barStPct2= barStPct1 + geom_text(aes(label=percent(pctCol,
+                                                    accuracy=0.1)),
+                           size = 4, fontface='bold',
+                           position = position_fill(vjust = 0.5))
```

[4] No qualitative Brewer palette is safe for colorblindness, neither photocopying nor printing if you have more than four levels.

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

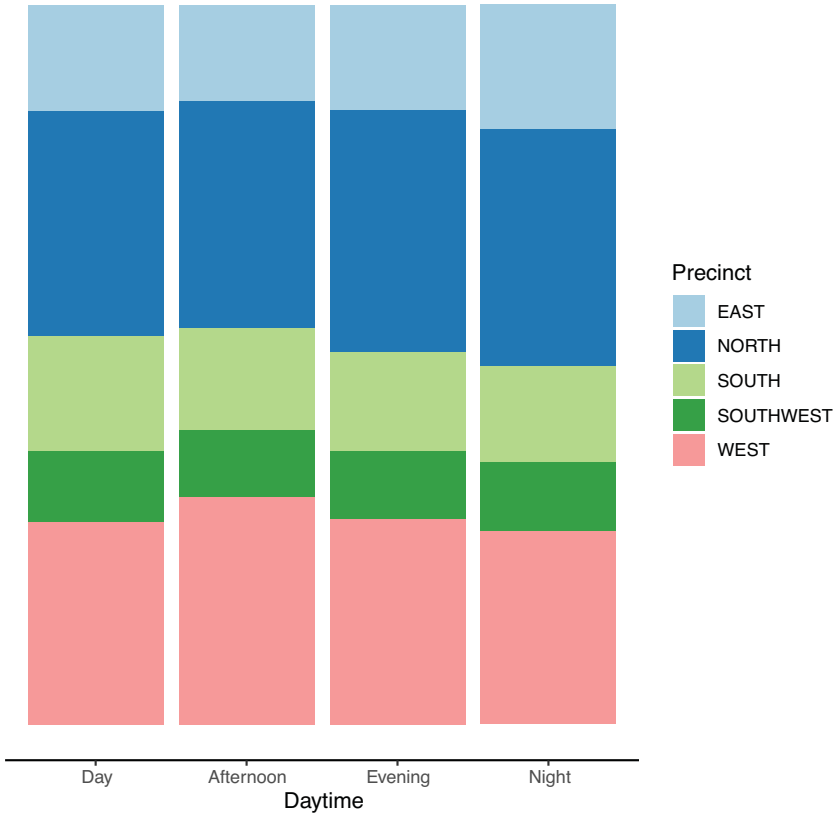

Figure 5.7 Percentage stacked bar chart
Data from Seattle Open Data Portal (City of Seattle, 2019).

Remember that you may clutter your plot with these annotations, so if you just need the audience to realize the qualitative contribution of each row level, it is better to omit them.

Let me plot object `barStPct2` in **Python**, which will include the elements from object `barStPct1`:

```
base4= ggplot(PrecDaytiDF,
              aes(x='daytime',y='pctCol',
                  fill='precint')) + theme_classic()

barStPct2 = base4 + scale_fill_brewer(type='Qualitative',
                                      palette = "Paired")

barStPct2 += theme(axis_title_y = element_blank(),
                   axis_text_y  = element_blank(),
                   axis_line_y  = element_blank(),
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

```
                  axis_ticks_major_y=element_blank())

barStPct2 += geom_bar(stat="identity",
                      position='fill')

barStPct2 += geom_text(aes(label='pctCol*100'),
                       format_string='{:.1f}%',
                       size = 8, fontweight='bold',
                       position = position_fill(vjust = 0.5))
```

5.1.4 Bigger Tables: Focusing on Highs/Lows

Categorical data do not usually have several levels, but there are circunstances when they do. The plots we have just seen may not suit more complex situations. Take a look at this contingency table:

```
> #contingency table of counts
> (CrimeDay=table(crime$crimecat,crime$OccurredDayTime))
```

```
                          afternoon   day evening night
  AGGRAVATED ASSAULT           5366  3564    4884  7501
  ARSON                         167   196     191   486
  BURGLARY                    22288 24139   14121 16082
  CAR PROWL                   38273 26740   42595 34839
  DISORDERLY CONDUCT             81    41      67    79
  DUI                           939   706    2038  8522
  FAMILY OFFENSE-NONVIOLENT    2516  1748    1217  1120
  GAMBLE                          4     4       7     2
  HOMICIDE                       46    41      49   131
  LIQUOR LAW VIOLATION          491   112     410   606
  LOITERING                      31    20      25     9
  NARCOTIC                     6416  2415    3924  4109
  PORNOGRAPHY                    53    65      17    31
  PROSTITUTION                  675   115    1425  1340
  RAPE                          318   332     354   854
  ROBBERY                      4737  2584    4139  5372
  SEX OFFENSE-OTHER            1759  1501    1014  1776
  THEFT                       64868 38687   38980 28410
  TRESPASS                     5184  4848    2598  3289
  WEAPON                       1445   735     947  1624
```

In situations like this, you can represent the contingency table with other elements but the possibility of finding a clear relationship gets more troublesome, so the next plots will mainly allow you to identify worst or best situations, or patterns.

Let me turn the table into a data frame before plotting:

```
> #marginal per column (column adds to 1)
> CrimeDay_mgCol=prop.table(CrimeDay,margin = 2)
> #making a data frame from contingency table
> CrimeDayDF=as.data.frame(CrimeDay)
> names(CrimeDayDF)=c("crime","daytime","counts")
> #adding marginal columns percents:
> CrimeDayDF$pctCol=as.data.frame(CrimeDay_mgCol)[,3]
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

```
> # reformatting ordinal data
> CrimeDayDF$daytime=factor(CrimeDayDF$daytime,
+                           levels = RightOrder,
+                           ordered=TRUE)
```

You need to do the same thing in **Python**:

```
#contingency table of counts
CrimeDay=pd.crosstab(crime.crimecat,crime.OccurredDayTime)

#marginal per column (column adds to 1)
CrimeDay_mgCol=pd.crosstab(crime.crimecat,crime.OccurredDayTime,
                           normalize='columns')

#making a data frame from contingency table
CrimeDayDF=CrimeDay.stack().reset_index()
CrimeDayDF.columns=["crime","daytime","counts"]

#adding marginal columns percents:
CrimeDayDF['pctCol']=CrimeDay_mgCol.stack().reset_index().iloc[:,2]

# reformatting ordinal data
RightOrder=["day","afternoon","evening","night"]
CrimeDayDF.daytime=pd.Categorical(CrimeDayDF.daytime,
                                  categories=RightOrder,
                                  ordered=True)

# reformatting ordinal data
maxMargiCrime=CrimeDay.apply(max,axis=1)
sortedCrime=maxMargiCrime.sort_values(ascending=True)
newCrimeAxis=sortedCrime.index
```

Once I have the data frame, I can reproduce the percentages for this crosstab with a **dot plot** with a **theme_minimal**:

```
> # reorder table vertically by max count per daytime
> base5 = ggplot(CrimeDayDF,aes(x=daytime,
+                y=reorder(crime, pctCol,FUN=max))) + theme_minimal()
> base5 = base5 + labs(y="crime") # not needed in Python
> # plot value as point, size by value of percent
> BTableDot = base5 + geom_point(aes(size = pctCol))
> # label points, label with 2 decimal positions (accuracy)
> # percent() need library "scale"
> BTableDot = BTableDot + geom_text(aes(label = percent(pctCol,
+                                                       accuracy = 0.01)),
+                                   # push text to the right
+                                   nudge_x = 0.4,
+                                   size=3)
> # no need for legend
> BTableDot = BTableDot + theme(legend.position="none")
```

The object `BTableDot` is shown in Figure 5.8.

The **Python** code to get Figure 5.8 is:

```
# reorder table vertically by max count per daytime
base5 = ggplot(CrimeDayDF,
               aes(x='daytime',
                   y='crime')) + theme_minimal()
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

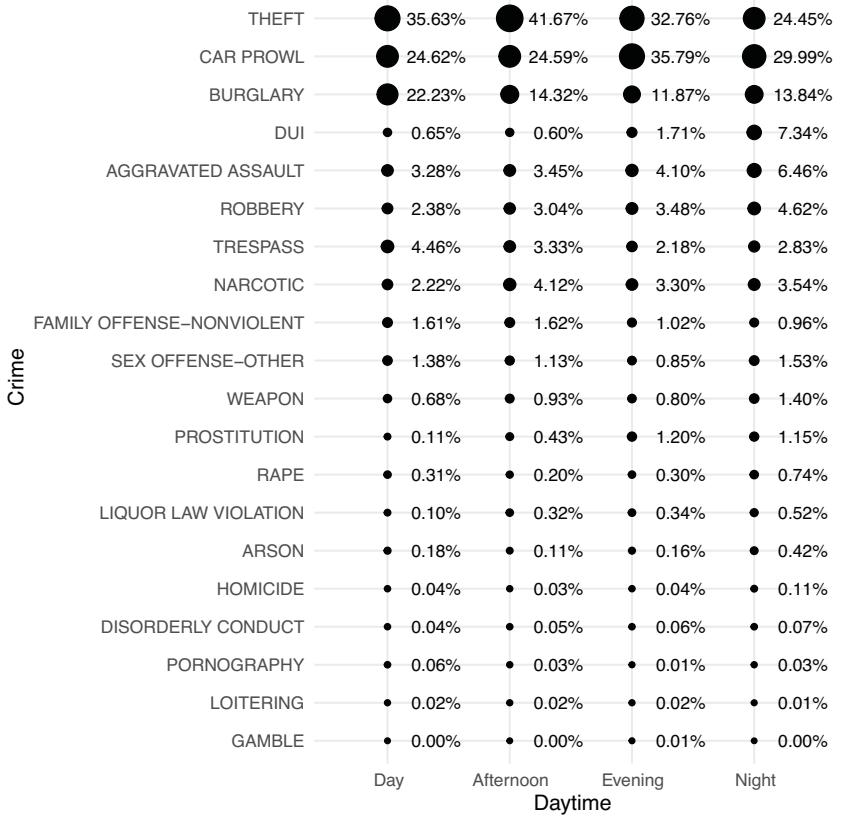

Figure 5.8 Dot plot for big crosstab
Data from Seattle Open Data Portal (City of Seattle, 2019).

```
base5 += scale_y_discrete(limits=newCrimeAxis)

# plot value as point, size by value of percent
BTableDot = base5 + geom_point(aes(size = 'counts'))

# label points, label with 2 decimal positions:
BTableDot += geom_text(aes(label = 'pctCol*100'),
                                  format_string='{:.2f}%',
                                  # push text to the right
                                  nudge_x = 0.3,
                                  size=8)
# no need for legend
BTableDot += theme(legend_position="none")
```

The plot looks nice, but unless the differences are notorious, you may see more noise than information, which distracts and delays decision-making.

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

Remember that the dot plot depends on how easily we can decode size or area; this is harder to compare than length. So, the bars should come back, but with the help of `facets`, which will create a plot for a particular level of a categorical variable:

- Prepare the bars. Again, I will reorder the bars according to the maximum share by the day time. Notice that I am using `percent_format`, from the library *scales* used before, to format the axis text; the `accuracy` argument is set to **1** to indicate that I need no decimals (I used **0.01** before to request two decimal places). Notice that in the **Python** code, *plotnine* will use a function from *mizani* which is also named `percent_format`.

```
> #crime ordered
> base6  = ggplot(CrimeDayDF,
+                 aes(x = reorder(crime,pctCol,FUN=max),
+                     y = pctCol) ) + theme_minimal()
> base6 = base6 + labs(x="crime") # not needed in Python
> #formatting text axis
> base6 = base6 +
+         scale_y_continuous(labels = percent_format(accuracy = 1))
> #basic bar
> BTableBar = base6 + geom_bar( stat = "identity" )
>
```

  The previous code will produce a bar for each crime, as no information on `daytime` has been input.
- Partitioning. Now we create a barplot for each day time:

```
> #Facetting: one plot per 'daytime'
> BTableBar = BTableBar + facet_grid(.~daytime)
```

- Flipping. Now I flip the plot so that it resembles the original crosstab:

```
> #Flipping
> BTableBar= BTableBar + coord_flip()
```

You could make some changes to the text of the crimes:

```
> # altering axis text for crime
> BTableBar= BTableBar + theme(axis.text.y = element_text(size=8,
+                                                         angle = 45))
> # altering axis text for percent
> BTableBar= BTableBar + theme(axis.text.x = element_text(size=6,
+                                                         angle = 45))
>
```

Figure 5.9 shows how the `BTableBar` object is drawn.

Figure 5.9 allows you to clearly identify the worst crimes in the region, even if there is not a clear relationship at the variable level.

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

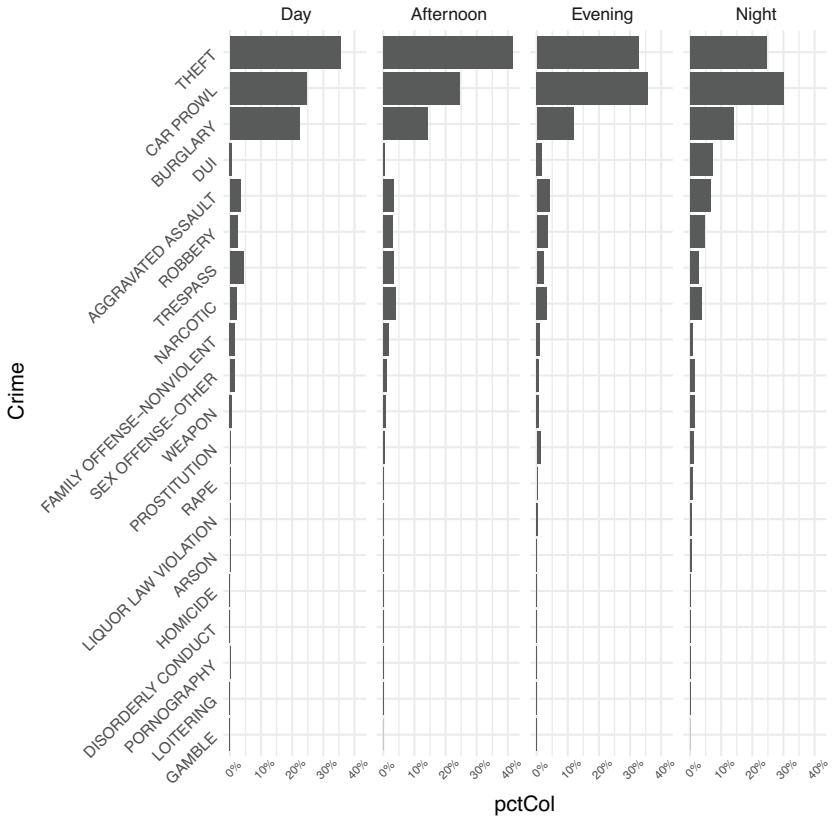

Figure 5.9 Barplot facetted and flipped

Data from Seattle Open Data Portal (City of Seattle, 2019).

The **Python** code to get Figure 5.9 is:

```
from mizani.formatters import percent_format

#crime ordered
base6  = ggplot(CrimeDayDF,
                aes(x = 'crime',
                    y = 'pctCol') ) + theme_minimal()
#formatting text axis
base6 += scale_x_discrete(limits=newCrimeAxis)
base6 += scale_y_continuous(labels = percent_format())

#basic bar
BTableBar = base6 + geom_bar( stat = "identity" )

#Facetting
BTableBar += facet_grid('~ daytime')

#Flipping
BTableBar += coord_flip()
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

```
# altering axis text for crime
BTableBar += theme(axis_text_y = element_text(size=8,
                                              angle = 45,
                                              va='top'))
# altering axis text for percent
BTableBar += theme(axis_text_x = element_text(size=6,
                                              angle = 45,
                                              va='bottom'))
```

The **heatmap** is an alternative in a similar situation. However, keep in mind that you might need a legend for the audience to understand the color weight and to avoid plotting the numbers on top of the cells or *tiles*. Besides, you need to assess if the heatmap can be easily understood by the audience.

The heatmap can plot the relative weight of each cell to the total of the table or just the marginal, so make sure to clarify this to the audience. Let me plot this one relative to the whole table (reorder by counts of crimes):

```
> base7  = ggplot(CrimeDayDF,
+                 aes(x = daytime,
+                     y = reorder(crime, counts,FUN=max),
+                     fill = counts)) + theme_minimal()
> base7 = base7 + labs(y="crime") # not needed in Python
> # default heatmap
> heat1 = base7 +  geom_tile()
> # customizing color
> heat1 = heat1 +scale_fill_gradient(low = "gainsboro",
+                                    high = "black")
> # moving legend to the top
> heat1 = heat1 + theme(legend.title = element_blank(),
+                       legend.position="top")
> # making legend colorbar wider
> heat1 = heat1 + guides(fill=guide_colorbar(barwidth=10))
```

Figure 5.10 shows how `heat1` object is drawn.

This is the first time that I am representing a color weight relative to the whole table, in all the previous cases the weight for areas or length was computed per column. Our data frame `CrimeDayDF` does not have the relative share to the whole table, only per column,[5] so I just used the counts.

The **Python** code to get Figure 6.2 is given next:

```
base7 = ggplot(CrimeDayDF,
               aes(x = 'daytime',y = 'crime',
                   fill = 'counts')) + theme_minimal()
base7 += scale_y_discrete(limits=newCrimeAxis)

# default heatplot
heat1 = base7 + geom_tile()
# customizing color
heat1 += scale_fill_gradient(low = "gainsboro",
                             high = "black")
# moving legend to the top
```

[5] You can easily add that column to `CrimeDayDF`; you just need to request `prop.table` without using the `margin` argument.

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

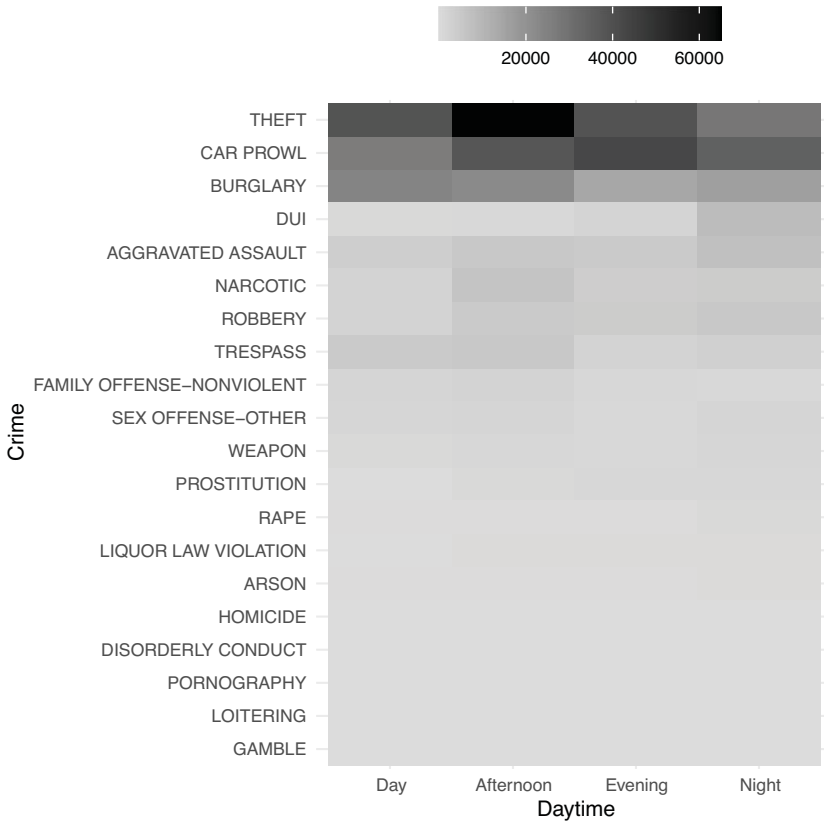

Figure 5.10 Heatmap – tiles weighted relative to total

The darker the greater the count of crimes for the whole table. Data from Seattle Open Data Portal (City of Seattle, 2019).

```
heat1 += theme(legend_title = element_blank(),
               legend_position="top")
# making legend colorbar wider
heat1 += guides(fill=guide_colorbar(barheight=200))
```

5.2 Cat–Num Relationship

In this section, I am interested in how a categorical variable can help us better understand the behavior of a numeric variable. Given the data we will be using, the curiosity and experience of the analyst is critical in mining the data to reveal some insight, as numerical data have longer value ranges than categorical data.

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

In our current `crime` data frame we had a variable that informs the amount of days it takes someone to report a crime:

```
> summary(crime$DaysToReport)
```

```
    Min.  1st Qu.   Median     Mean  3rd Qu.     Max.     NA's
    0.00     0.00     0.00     7.65     1.00 36525.00        2
```

You can see there is a lot of variation: from crimes that were reported the same day they occurred (**0**) to crimes that took around 100 years (**36525**). These data might be wrong, but I will assume they are not.

Remember the `crime` data frame collected information per year, so let me explore the distribution of crime reports per year:

```
> # counting crimes per year
> table(crime$year)
```

```
 1908  1964  1973  1974  1975  1976  1977  1978  1979  1980  1981  1985
1986
    1     1     1     1     2     2     1     1     2     2     2
2     1
 1987  1988  1989  1990  1991  1993  1994  1995  1996  1997  1998  1999
2000
    1     1     1     2     2     5     2     6     4     5    20
8    40
 2001  2002  2003  2004  2005  2006  2007  2008  2009  2010  2011  2012
2013
   53    17    28    41    49    99   627 42792 45055 43350 41296 41005
   45548
 2014  2015  2016  2017  2018
49315 47687 49202 50302 43114
```

Let me keep the crimes since 2008, as the counts in that year are of the same magnitude as the remaining years. I am not saying you should do this in other cases, but the other years may be still in the process of being digitalized.[6]

```
> # keeping years since 2008
> crime2=crime[crime$year≥2008,]
```

Let me get rid of missing rows in the numeric variable, and rename the missing crime categories this way:

```
> # making data without missing values
> crime2=crime2[complete.cases(crime2$DaysToReport),]
> crime2$crimecat[is.na(crime2$crimecat)]='UNcategorized'
```

[6] The portal does state that the data are available from 2008 https://data.seattle.gov/Public-Safety/SPD-Crime-Data-2008-Present/tazs-3rd5.

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

The **Python** code for this pre-processing stage is shown next:

```
# counting crimes per year
crime.year.value_counts()

# keeping years since 2008
crime2=crime[crime.year≥2008].copy()

# making data without missing values
crime2.dropna(subset=['DaysToReport'],inplace=True)
crime2.fillna(value={'crimecat': 'UNcategorized'},inplace=True)

# sorting crimes for plotting
maxD=crime2.groupby('crimecat').describe()['DaysToReport'][['max']]
maxDSort=maxD.sort_values(by=['max'],ascending=True).index
```

Now, let me show you how to explore the time it takes to report a crime by crime category:

```
> base8 = ggplot(data=crime2,
+              aes(x=reorder(crimecat,DaysToReport,FUN=max),
+                  y=DaysToReport))  + theme_minimal()
> base8= base8 + labs(x="crime")
> boxCrime=base8 + geom_boxplot() + coord_flip()
```

Figure 5.11 presents `boxCrime` object using boxplots. As you can see, I have used a boxplot to display the distribution of `DaysToReport` for every category:

As shown in Figure 5.11, most crimes take a few days to report, but the presence of outliers makes it difficult to find some other pattern. The code to reproduce Figure 5.11 in **Python** is the following:

```
base8=ggplot(crime2,
             aes(x='crimecat',
                 y='DaysToReport')) + theme_minimal()
base8 += labs(x="crime")
base8 += scale_x_discrete(limits=maxDSort)
boxCrime=base8 + geom_boxplot() + coord_flip()
```

Discovering some pattern in the variability requires that we focus on a subset of the data. In situations like this, I recommend you compute the second and third quartile to decide what to keep:

```
> # computing quartiles 2 and 3
> q23=function(x){quantile(x,probs = c(0.5,0.75))}
> aggregate(data=crime2,DaysToReport~crimecat,q23)
```

```
                    crimecat DaysToReport.50% DaysToReport.75%
1          AGGRAVATED ASSAULT                0                0
2                      ARSON                0                0
3                   BURGLARY                0                1
4                  CAR PROWL                1                1
5          DISORDERLY CONDUCT                0                0
6                         DUI                0                0
7  FAMILY OFFENSE-NONVIOLENT                0                0
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

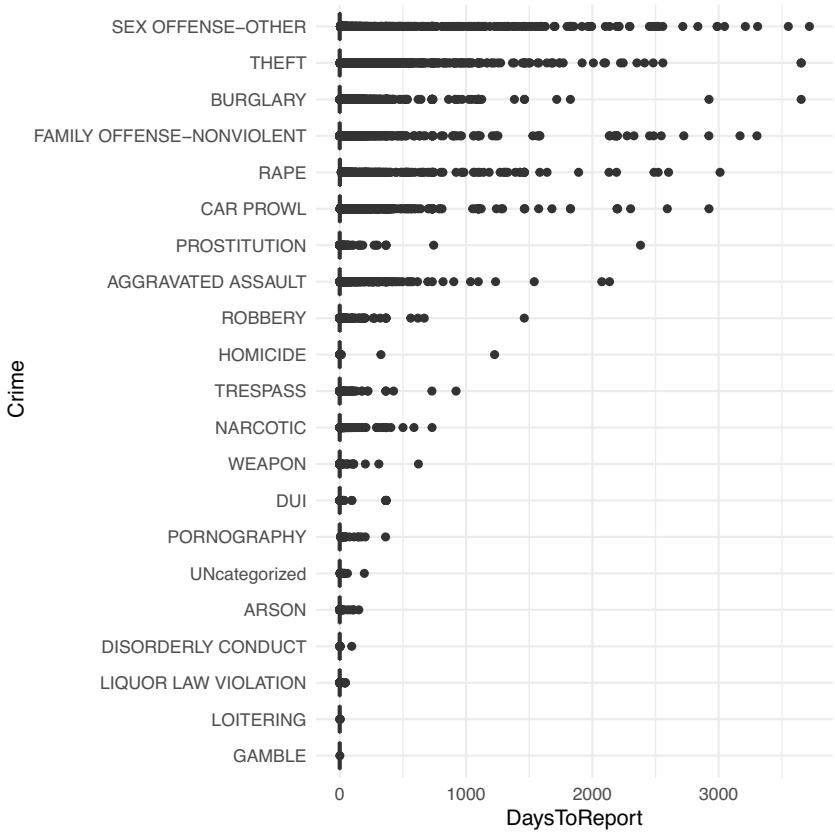

Figure 5.11 Boxplots per category

Categories have been reordered. Data from Seattle Open Data Portal (City of Seattle, 2019).

8	GAMBLE	0	0
9	HOMICIDE	0	0
10	LIQUOR LAW VIOLATION	0	0
11	LOITERING	0	0
12	NARCOTIC	0	0
13	PORNOGRAPHY	0	2
14	PROSTITUTION	0	0
15	RAPE	1	5
16	ROBBERY	0	0
17	SEX OFFENSE-OTHER	0	2
18	THEFT	0	1
19	TRESPASS	0	0
20	UNcategorized	0	1
21	WEAPON	0	0

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

The code in **Python** for this operation is:

```
q23=['50%','75%'] # list of quartiles
crime2.groupby('crimecat').describe()['DaysToReport'][q23]
```

These confirms that 75 percent of the crimes took less than a week (five days in worst case) to be reported. Let me keep the cases that took a year or longer to report for my next plot:

```
> #subsetting
> crimeYear=crime2[crime2$DaysToReport≥365,]
> #creating new variable
> crimeYear$YearsToReport=crimeYear$DaysToReport/365
```

Now, let's try a similar plot, this time ordering the crimes by the third quartile of `YearsToReport` (notice I am using the extra argument `probs` in `FUN` for the function `quantile`):

```
> base9=ggplot(data=crimeYear,
+              aes(x=reorder(crimecat,YearsToReport,
+                            FUN=quantile,probs=0.75),
+                  y=YearsToReport)) + theme_minimal()
> base9= base9 + labs(x="crime")
> boxCrimeY=base9 + geom_boxplot() + coord_flip()
```

Object `boxCrimeY` can be seen in Figure 5.12.

Remember that *plotnine* does not have the exact functionality of `reorder` from **R**'s *ggplot*; so, you need to get the crime names ordered by the third quartile or Q75 (75th quantile) first:

```
#subsetting
crimeYear=crime2[crime2.DaysToReport>365].copy()

#creating new variable
crimeYear['YearsToReport']=crimeYear.DaysToReport/365

# ordering the crimes by the q75 of YearsToReport
## subset with variables needed
subCrimeYear=crimeYear[['crimecat','YearsToReport']]
## grouping the subset by Q75
Q75=subCrimeYear.groupby('crimecat').quantile(q=0.75)
## sorting the grouping by Q75 of 'YearsToReport'
Q75Sort=Q75.sort_values(by=['YearsToReport'],ascending=True)
## just getting the names in order
sortedCrimesbyQ75=Q75Sort.index
```

The object `sortedCrimesbyQ75` in **Python** is just the names of the crimes, not a data frame. I can use those names to reproduce Figure 5.12:

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

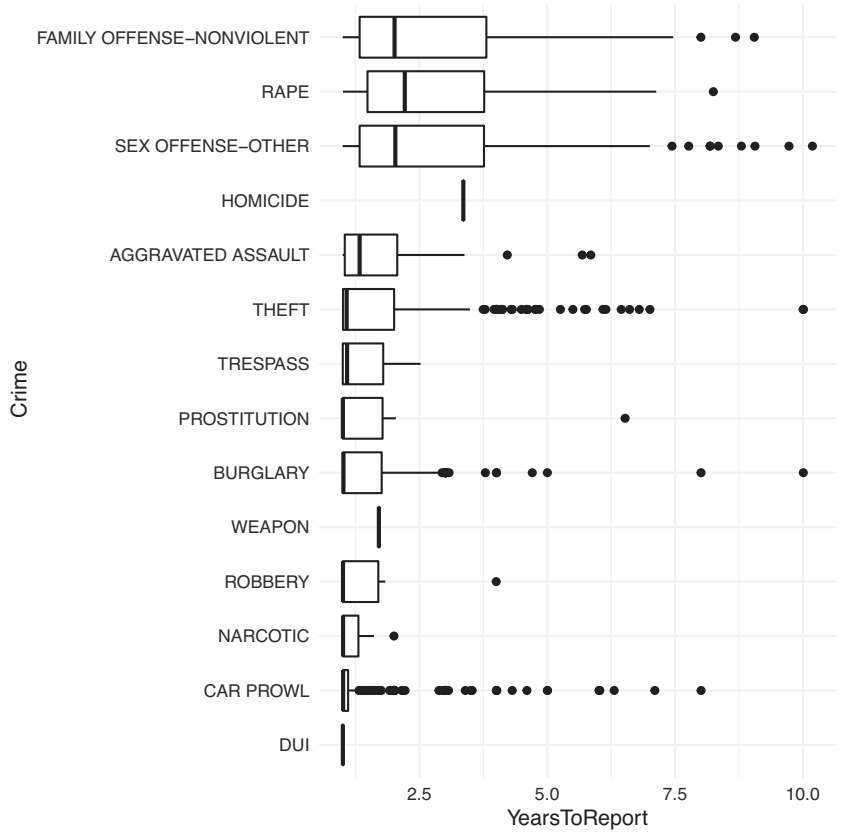

Figure 5.12 Boxplots per category (II) for crimes that took a year or longer to report

Categories have been reordered. Data from Seattle Open Data Portal (City of Seattle, 2019).

```
base9=ggplot(crimeYear,
             aes(x='crimecat',
                 y='YearsToReport')) + theme_minimal()
base9 += labs(x="crime")
#manually sorting
base9 += scale_x_discrete(limits=sortedCrimesbyQ75)
boxCrimeY = base9 + geom_boxplot() + coord_flip()
```

Figure 5.12 still has several crimes with outliers but you can see a better ordering of crimes and prepare an explanation for the crimes that take longer to report. Pay attention to the top three types of crime whose third quartiles are the highest: the crimes that take longer to report since 2008 are related to

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

family and sex offenses (notice the horizontal dimension is showing counts in years).

The problem with Figure 5.12 is that boxplots are not a very common visual for non-academic audiences. Let me plot the third quartiles only, following the next steps:

- Prepare a function. Functions that require *one* argument to compute an statistic can be used immediately with *ggplot*, but in this case the `quantile` function requires two. I did not need it above, because the `reorder` function allows adding another argument. So, let me create my **q3** function:

```
> # ad-hoc q3
> q3=function(x){quantile(x,probs = 0.75)}
> theQ3='75%'## Labels for 'color'
```

- **Group** all the third quantiles and draw a line connecting them. Notice the color in `aes` is a label – it is a constant value, as the color will not change for every crime:

```
> # line of q3 grouped using last base9
> q3Y = base9 + stat_summary(aes(group=T, color=theQ3),
+                                fun=q3,
+                                geom="line",
+                                size=4)
```

- Flip the coordinates (optional):

```
> #flipping
> q3Y = q3Y + coord_flip()
```

The object `q3Y` is drawn in Figure 5.13. Notice this plot gives you a default color, as the color in `aes` was just a label.

The code in **Python** to reproduce Figure 5.13:

```
# ad-hoc q3
import numpy as np

q3=lambda x:np.quantile(x,0.75)
theQ3='75%'## Labels for 'color'

# line of q3 grouped using last base9
q3Y = base9 + stat_summary(aes(group=True,color='theQ3'),
                           fun_y=q3,
                           geom='line',
                           size=5)
# flipping
q3Y = q3Y + coord_flip()
```

Let me show you how to add the minima and maxima values to the plot, by following the same strategy:

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

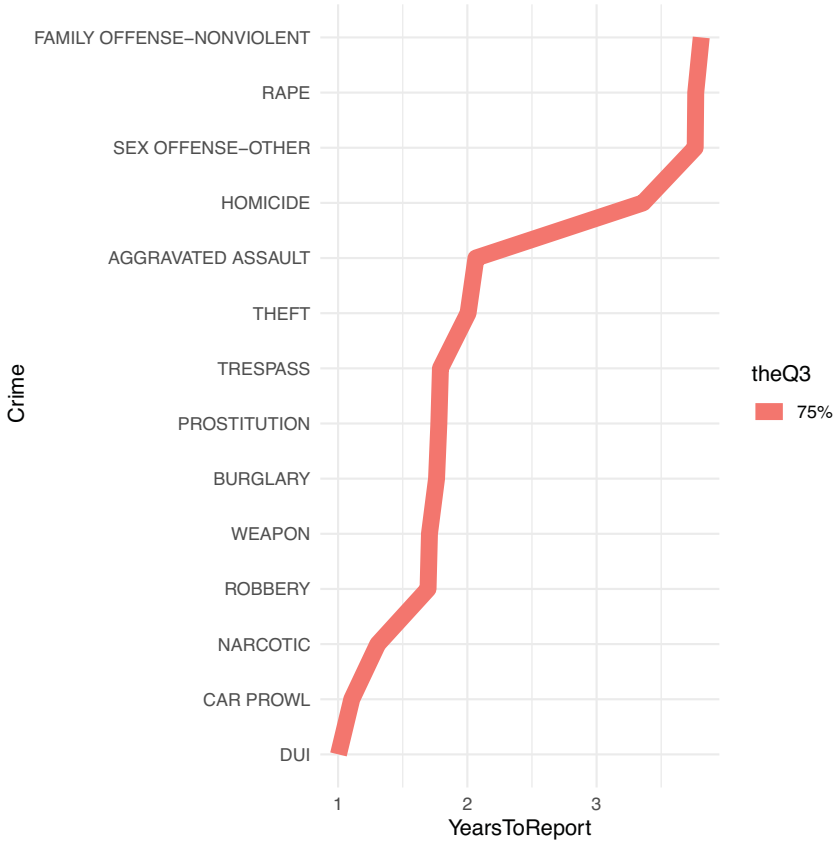

Figure 5.13 Line plot for grouped values

The line shows the third quartile of the time it takes to report a crime, separated by crime. Data from Seattle Open Data Portal (City of Seattle, 2019).

```
> # More labels for 'color'
> theMin='Minima'
> theMax='Maxima'
> #adding minima (group of minima):
> mq3Y =q3Y + stat_summary(aes(group=T, color=theMin),
+                          fun=min,
+                          geom="point",
+                          size=3)
> #
> #adding maxima (group of maxima):
> Mmq3Y=mq3Y + stat_summary(aes(group=T, color=theMax),
+                           fun=max,
+                           geom="point",
+                           size=1)
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

If I plot the `Mmq3Y` object, I will get default colors. If I want my own colors, I should do this:

```
> #customizing legend and colors
> orderStats=c("Minima","75%","Maxima")
> cols_orderStats=c("grey80","grey50","black")
> Mmq3Yfin= Mmq3Y + scale_colour_manual(name='Stats',
+                                       limits = orderStats,
+                                       values = cols_orderStats)
```

The object `Mmq3Yfin` is shown in Figure 5.14. Notice that combining the grey scale I made every *geom* visible.

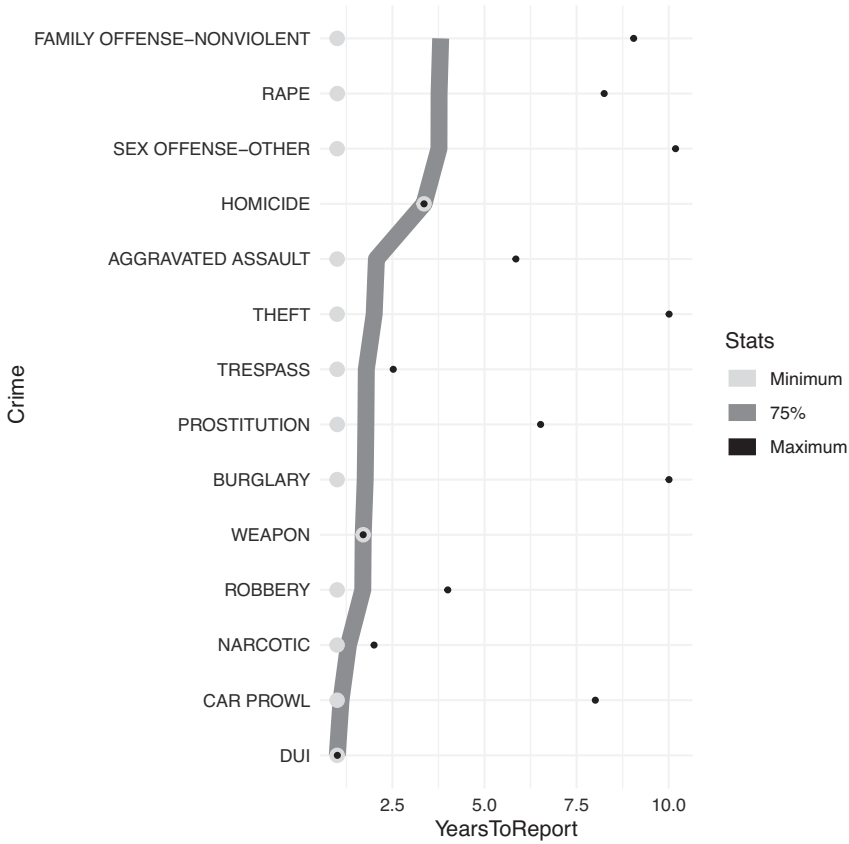

Figure 5.14 Line plot for grouped values

The line shows the third quartile of the time it takes to report a crime, separated by crime. Data from Seattle Open Data Portal (City of Seattle, 2019).

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

The **Python** version for the object `Mmq3Yfin`:

```
#Labels for colors
theMin='Minima'
theMax='Maxima'

mq3Y = q3Y  + stat_summary(aes(group=True,color='theMin'),
                           fun_y=np.min,
                           geom='point',
                           size=4)

Mmq3Y = mq3Y + stat_summary(aes(group=True,color='theMax'),
                           fun_y=np.max,
                           geom='point',
                           size=1)

#customizing legend and colors
orderStats=["Minima","75%","Maxima"]
cols_orderStats=['silver','grey','black']
Mmq3Yfin= Mmq3Y + scale_color_manual(name='Stats',
                                     limits = orderStats,
                                     values = cols_orderStats)
```

You are welcome to alter the code to produce the object `Mmq3Yfin`, for instance, you can change any other statistical value you believe summarizes the numerical variable. Most people understand the `mean`, but in a situation where there is too much skewness, like the ones shown in Figures 5.12 and 5.14, you may prefer quartiles.

Also, the cat–num relationships are used to reveal if the summary statistics differ from one another. Several tests, parametric and non-parametric, exist to test for those differences, so you may want to annotate those plots with the coefficient that confirms if the distribution of a numerical variable differs significantly between or among categories. For further information on these tests I recommend you review Pons (2014) and Alvo and Yu (2018).

5.3 Num–Num Relationship

5.3.1 Num–Date Relationship

Time belongs to the interval scale (Stevens, 1946). The *zero* does not mean absence of time, the multiplicative interpretations do not make sense (16:00 is not twice 08:00), and the difference between two pairs of values is comparable (the time differences between 18:00 and 15:00, and 11:00 and 8:00, are the same).

I have a column with dates in the *crime2* data frame:

```
> str(crime2$OccurredDate)
```

```
 chr [1:498666] "2013-07-09" "2013-07-09" "2013-07-09" "2013-07-09" ...
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

Let me create a frequency table with that column, so I can see how many crime events have happened per day:

```
> # frequency of events
> allCrimes=as.data.frame(table(crime2$OccurredDate))
> names(allCrimes)=c('dates','count')
> head(allCrimes)
```

```
       dates count
1 2008-01-01   178
2 2008-01-02   103
3 2008-01-03   116
4 2008-01-04   114
5 2008-01-05   104
6 2008-01-06    95
```

The next code **should not be run**.

```
> baset=ggplot(allCrimes, aes(x=dates, y=count)) + theme_classic()
> # line of time by count:
> tsl=baset + geom_line(alpha=0.25)
> # result:
> tsl
```

There is nothing wrong with the code itself, the problem is the data frame format:

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

```
'data.frame':   3963 obs. of  2 variables:
 $ dates: Factor w/ 3963 levels "2008-01-01","2008-01-02",..: 1..
 $ count: int  178 103 116 114 104 95 105 114 131 100 ...
```

The column `date` is currently in text format, so we need to convert it into **Date** format. For that, you first need to know the symbols used for formatting dates:

- For a date number (0–31): **%d**.
- For weekday:
  - For abbreviated weekday (Mon, Tue, etc): **%a**.
  - For non-abbreviated weekday (Monday, Tuesday, etc): **%A**.
- For month:
  - For month number (00–12): **%m**.
  - For abbreviated month (Jan., Feb., etc): **%b**.
  - For non-abbreviated month: **%B**.

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

- For year:
  - For a 2-digit year: **%y**.
  - For a 4-digit year: **%Y**.

Also, you may need to use some symbol, like *dash* or *slash*, to concatenate date elements.

Then, you can convert the column **allCrimes$date** into a date format like this:

```
> #formatting date (respect the format found in text):
> allCrimes$dates=as.Date(allCrimes$dates,format="%Y-%m-%d")
```

Once it is in the right format, you can request a statistic:

```
> median(allCrimes$dates)
```

```
[1] "2013-06-04"
```

The code in **Python** to accomplish the same follows:

```
allCrimes = pd.value_counts(crime2.OccurredDate).reset_index()
allCrimes.columns=['dates','counts']
allCrimes['dates']=pd.to_datetime(allCrimes.dates,
                                  format="%Y-%m-%d")
```

Now, this code will work:

```
> baset = ggplot(allCrimes,
+                aes(x=dates, # already a DATE
+                    y=count)) + theme_classic()
> # lines highly transparent:
> tsl=baset + geom_line(alpha=0.2)
```

The object `tsl` is plotted in Figure 5.15.

The **Python** code is almost the same (with the variable names in quotations):

```
baset= ggplot(allCrimes,
              aes(x='dates', # already a DATE
                  y='counts')) + theme_classic()
tsl = baset + geom_line(alpha=0.2)
```

Time data are generally plotted with lines to show continuity, but the use of dots can be a good alternative to show the date-to-date behavior:[7]

```
> # Using previous dots with some transparency (same 'baset')
> tsp=baset + geom_point(alpha=0.2, #transparency
+                        shape=4)
```

[7] Remember that I presented the **R** symbols for dots in Figure 3.8 (For **Python** you should visit https://matplotlib.org/3.1.1/api/markers_api.html).

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

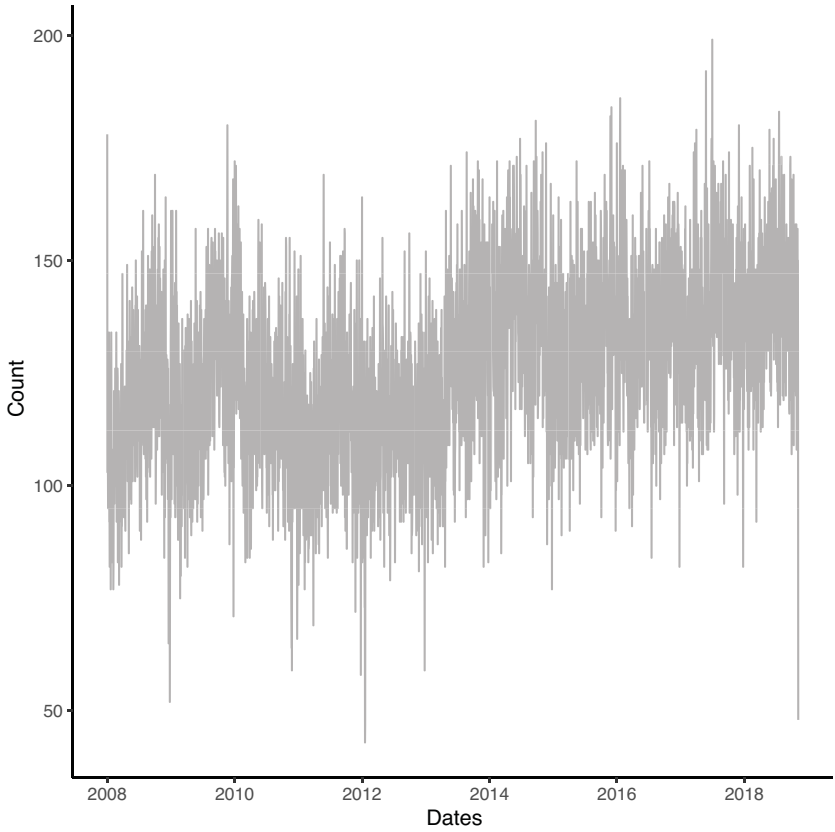

Figure 5.15 Basic time series

The horizontal dimension should be of *date* data type for *ggplot* to understand this is a time series. The frequency is the count of crime events on a particular day since 2008. Data from Seattle Open Data Portal (City of Seattle, 2019).

Now, I can use a line to show some pattern in time. Let me use a local regression line, also known as **loess** (Fox and Weisberg, 2019), for that purpose:

```
> tsp=tsp +geom_smooth(fill='grey70', #color around line
+                      method = 'loess', # to compute line
+                      color='black') #color of line
```

The date format on the horizontal axis can be customized in two ways. First, you can customize the text of the date shown (`date_labels`) and where you need the ticks (`date_breaks`). So, you may need to add some elements to the object `tsp`:

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

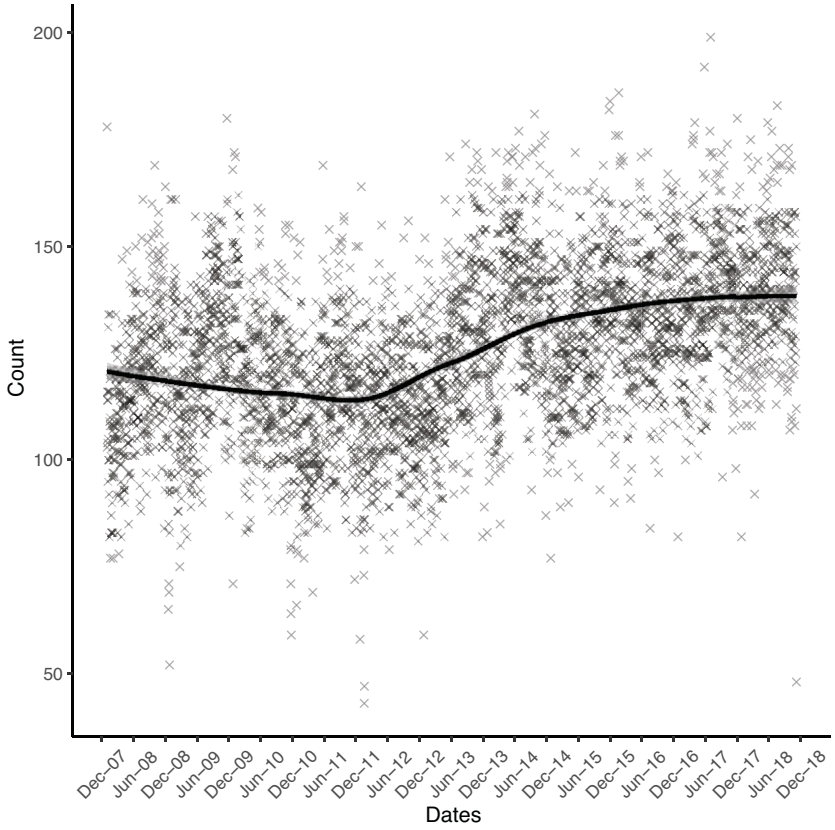

Figure 5.16 Time series using dots and lines

The dots represent the count of crime events per day since 2008; the line summarizes the long run pattern using a local regression approach. Data from Seattle Open Data Portal (City of Seattle, 2019).

```
> #add format to axis:
> tsp= tsp+scale_x_date(date_labels = "%b-%y", #how
+                       date_breaks = "6 months") #where
> #set up text values on each tick:
> tsp= tsp + theme(axis.text.x = element_text(angle=45,
+                                             vjust=0.5))
```

The object `tsp` is plotted in Figure 5.16.

Let me share the **Python** code to produce a result similar to Figure 5.16.[8] Notice that the text alignment using `vjust` in `element_text`, was not needed in **Python**.

[8] You may need to install *scikit-misc* (Kibirige, 2020b) for **loess** to work. Please follow instructions from https://github.com/has2k1/scikit-misc.

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

```
# dots with some transparency (same 'baset')
tsp= baset + geom_point(alpha=0.2,
                        shape='+')
#line for pattern
tsp += geom_smooth(fill='silver',
                   method='loess',
                   color='black')
#add format to axis:
tsp += scale_x_datetime(date_labels='%b-%Y',
                        date_breaks='6 months')
#set up text values on each tick:
tsp += theme(axis_text_x = element_text(angle=45))
```

We have used the dates as we found them. However, aggregating data may prove a useful option you can explore. Let me aggregate my daily data into week averages:

```
> library(lubridate)
> library(magrittr)
> library(dplyr)
> weekCrimes=allCrimes %>%
+             group_by(weekly=ceiling_date(dates, "week")) %>%
+               summarize(averages=mean(count)) %>% #create variable
+                 as.data.frame()
```

The previous code produced an aggregated data frame where you have the last day of the week as the label for the average of that week using the `ceiling_date` function from the *lubridate* package (Grolemund and Wickham, 2011). This is the first time I have used the *pipe* operator (%>%); which requires the package *magrittr* (Bach and Wickham, 2014b). I used the pipes to concatenate a series of actions; in this case, `group_by`, and `summarize` (which creates a variable named `averages`), both functions from the package *dplyr*(Wickham et al., 2020a). Each operation concatenated with a pipe receives the output of the previous action.

Notice that the product of this aggregation kept the date format:

```
> str(weekCrimes)
```

```
'data.frame':   567 obs. of  2 variables:
 $ weekly  : Date, format: "2008-01-06" "2008-01-13" ...
 $ averages: num  123 113.6 104 106.9 92.1 ...
```

Notice that using the function `ceiling_date` you kept the date format.[9] This function will take the first day available in the data for the first week; so, in this case, the mean could be computed for less than 7 days.

Let me produce the same aggregation using **Python**. *Pandas* has the function `resample`, which simplifies the previous **R** code. The function `mean` was then applied.

[9] You can get the week average per year with other classic functions, like *tapply*, but you will need to reformat the date column

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

```
myArguments={'rule':'W', 'on':'dates', 'closed':'left'}
#using myArguments in resample with **
weekCrimes=allCrimes.resample(**myArguments).mean()
```

Notice that I have used a nice **Python** feature to input the arguments using a dictionary. You simply prepare the arguments using the names (keys) and values the function requires; notice that the keys have to be written as text (inside quotations). The function `resample`, or any other function, simply uses the dict by prefixing `**`.

Then, let me prepare a code similar to the one that produced Figure 5.16 with the `weekCrimes` data frame:

```
> basetW = ggplot(weekCrimes,
+                 aes(x=weekly, # formatted as DATE
+                     y=averages)) + theme_classic()
> tspW=basetW + geom_point(alpha=0.2, #transparency
+                          shape=4)
> tspW=tspW +geom_smooth(fill='grey70', #color around line
+                        method = 'loess', # to compute line
+                        color='black') #color of line
> #add format to axis:
> tspW=tspW+scale_x_date(date_labels = "%b-%y", #how
+                        date_breaks = "6 months") #where
> #set up text values on each tick:
> tspW=tspW + theme(axis.text.x = element_text(angle=45,
+                                              vjust=0.5))
>
```

The object `tspW` is represented in Figure 5.17. Notice that I needed to adjust the vertical alignment of the tick labels (not needed in Python).

Let me show you the code in **Python**. Notice that the previous use of `resample` sent the dates column as the *index* of the *Pandas* data frame (the row names in **R**); so I have to specify that in the aesthetics.

```
basetW = ggplot(weekCrimes,
                aes(x='weekCrimes.index', #for index
                    y='counts')) + theme_classic()

tspW= basetW + geom_point(alpha=0.2,
                          shape='+')

tspW += geom_smooth(fill='silver',
                    method='loess',
                    color='black')

tspW += scale_x_datetime(date_labels='%b-%Y',
                             date_breaks='6 months')

tspW += theme(axis_text_x = element_text(angle=45))
```

The previous plots have allowed you to become familiar with basic time series plotting; however, we have counted all the crime types in the previous plots. Remember that there are several crimes in the `crime2` data frame.

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

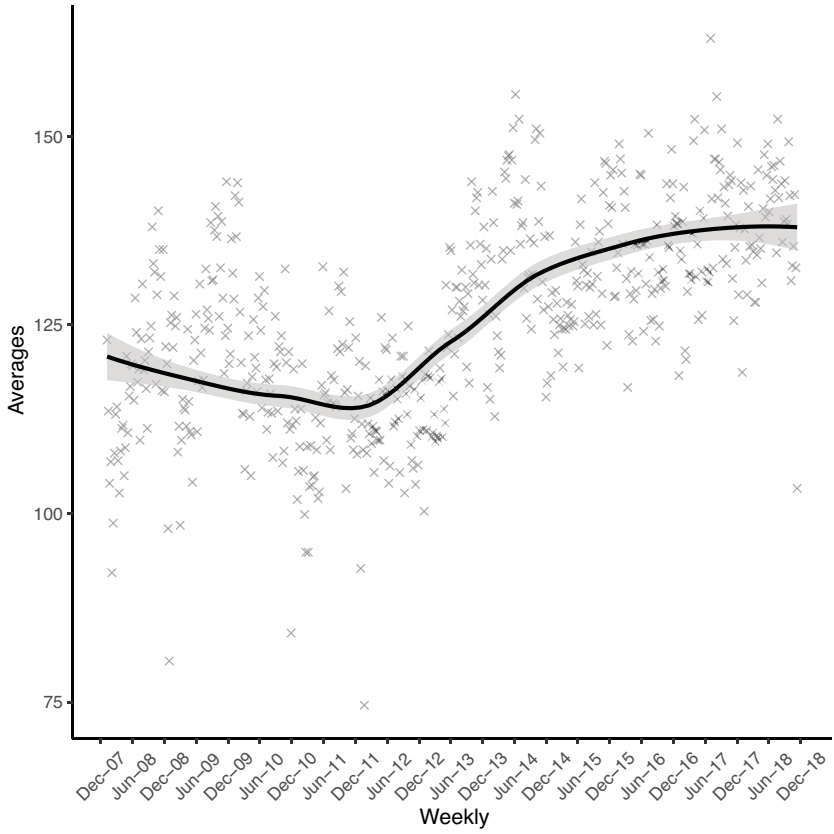

Figure 5.17 Aggregated time series using dots and lines

The dots represent the average of crime events per week since 2008; the line summarizes the long-run pattern using a local regression approach. Data from Seattle Open Data Portal (City of Seattle, 2019).

Let me make a contingency table between `OccurredDate` and `crimecat` to see time series of counts per crime:

```
> # making table
> crimeDate=table(crime2$OccurredDate,crime2$crimecat)
> # seeing first ten counts of four crimes:
> crimeDate[1:10,1:4]
```

```
            AGGRAVATED ASSAULT ARSON BURGLARY CAR PROWL
  2008-01-01                 13     0       18         25
  2008-01-02                  1     0       15         32
  2008-01-03                  2     0       27         33
  2008-01-04                  2     1       27         33
  2008-01-05                  5     0       17         22
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

```
2008-01-06                6     0       12        25
2008-01-07                3     0       17        26
2008-01-08                4     0       24        28
2008-01-09                3     0       24        40

2008-01-10                3     0        6        25
```

The previous table will be a lot of help for plotting each crime time series, as each row is a unique date, and each column represents the counts. Let's turn that table into a data frame for *ggplot* to work:

```
> #table of dates to data frame
> crimeDateDF=as.data.frame(crimeDate)
> #renaming columns
> names(crimeDateDF)=c("date",'crime','count')
> # formatting date column as Date type
> crimeDateDF$date=as.Date(crimeDateDF$date)
```

The steps to get the same data frame in **Python** are shown below:

```
# making table
crimeDate=pd.crosstab(crime2.OccurredDate,crime2.crimecat)
#table of dates to data frame
crimeDateDF=crimeDate.stack().reset_index()
#renaming columns
crimeDateDF.columns=['date','crime','count']
# formatting date column as Date type
crimeDateDF['date']=crimeDateDF['date'].astype('datetime64')
```

However, **Python** needs to organize, outside *plotnine*, the crimes in descending order by count of events (remember it does have the `reorder` function like in **R**), so let me do that next:

```
# sum by crimes, resulting in data frame:
ordCrime=crimeDateDF.groupby('crime').sum().reset_index()
# sum by crimes, resulting in data frame:
ordCrimeSort=ordCrime.sort_values(by=['count'],
                                  ascending=False)
#saving only names of crimes
descendCrimes=ordCrimeSort.crime.values
#setting the variable as an ordinal
crimeDateDF['crime']=pd.Categorical(crimeDateDF.crime,
                                    ordered=True,
                                    categories=descendCrimes)
```

Let's pay attention to these two crimes and make a subset:

```
> selection=c("AGGRAVATED ASSAULT", 'WEAPON')
> crimeDateDF_sub=crimeDateDF[crimeDateDF$crime %in% selection,]
```

Let me do the subsetting in **Python**, but we will need some extra steps. Since we reformatted the *crime* variable as an ordinal category in the `crimeDateDF` *Pandas* data frame, we need to get rid of the **unused** categories:

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

```
selection=["AGGRAVATED ASSAULT", 'WEAPON']
crimeDateDF_sub=crimeDateDF[crimeDateDF.crime.isin(selection)]

#extra step, get rid of values NOT used
crimeDateDF_sub.crime.cat.remove_unused_categories(inplace=True)
```

You may have needed to do that in **R** if *ggplot* could not use the `reorder` function. The **R** data frame has crimes as *text* or character type, so you do not need to do anything to make the unused categories disappear (see also Figure 6.1). However, **R** has its function `droplevels` to serve that purpose.

Let me set two date points to zoom into some date range when I plot my series:

```
> mini = as.Date("2014/1/1")
> maxi = as.Date("2018/12/31")
```

After the previous steps, let me share the code to see the behavior of both crimes during this particular range of time:

```
> basetSub = ggplot(crimeDateDF_sub,
+                   aes(x=date,y=count)) + theme_minimal()
> # all points for both crimes
> tspSub = basetSub + geom_point(alpha=0.3,
+                                shape=4,
+                                color='grey70')
> # loess lines for each crime
> tspSub = tspSub + geom_smooth(aes(color=crime),
+                               fill='white',size=2,
+                               method='loess',alpha=1)
> # color for each loess line
> tspSub = tspSub + scale_color_manual(values = c("grey", "black"))
> # format for dates on the horizontal
> tspSub = tspSub + scale_x_date(date_labels = "%b/%Y",
+                                date_breaks='2 months',
+                                limits = c(mini,maxi),
+                                expand=c(0,0)) # to look better!
> # Changing legend defaults and horizontal text default
> tspSub=tspSub+ theme(legend.title = element_blank(),
+                      legend.position="top",
+                      axis.text.x = element_text(angle=90,
+                                                 vjust=1,
+                                                 size=6))
```

Notice that I have included the argument `expand` in `scale_x_date`. Using those values avoids extra space before and after the date limits. I have used the same in **Python**. You can observe the parallel behavior of the crimes in object `tspSub`, represented in Figure 5.18.

The **Python** code to replicate Figure 5.18 follows below. Notice the code used to set the text values as date format using `Timestamp` from *Pandas*.

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

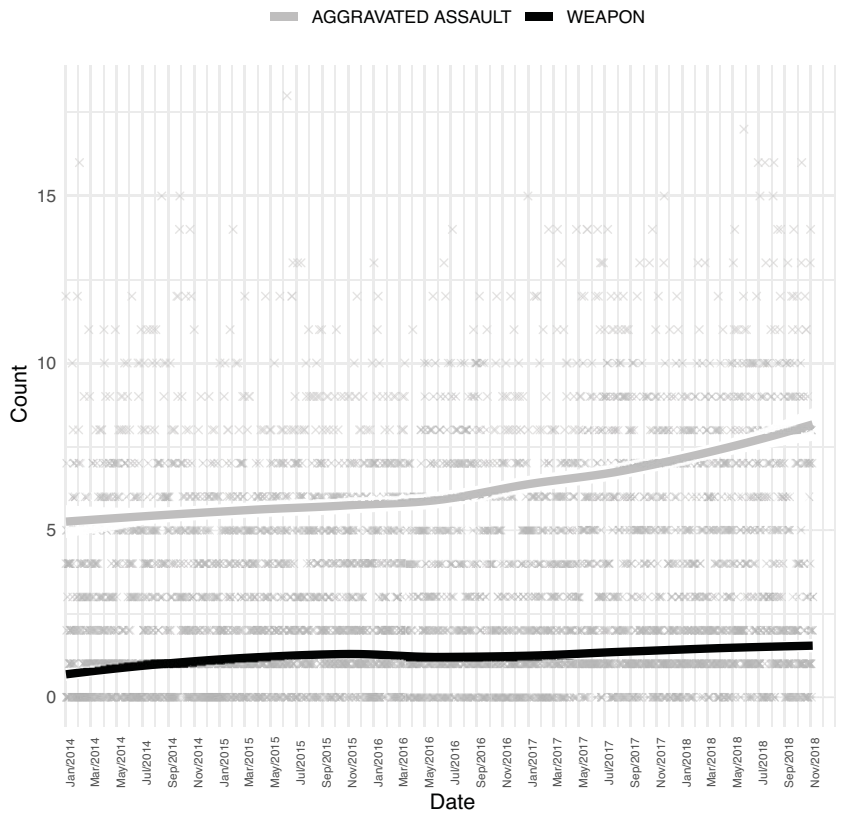

Figure 5.18 Multiple time series

The dots represent the daily count of both crime events, and the loess lines represent the pattern of the crimes selected. Data from Seattle Open Data Portal (City of Seattle, 2019).

```
mini = pd.Timestamp("2014/1/1") #51
maxi = pd.Timestamp("2018/12/31")

basetSub=ggplot(crimeDateDF_sub,
                aes(x='date',y='count')) + theme_minimal()

tspSub  = basetSub + geom_point(alpha=0.3,
                                shape='x',
                                color='silver')

tspSub += geom_smooth(aes(color='crime'),
                      fill='white',size=2,
                      method='loess',alpha=1)

tspSub += scale_color_manual(values = ["grey", "black"])
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

```
tspSub += scale_x_datetime(date_labels='%b/%y',
                      date_breaks='2 months',
                      limits = [mini,maxi],
                      expand=[0,0])

tspSub += theme(legend_title = element_blank(),
                legend_position="top",
                axis_text_x = element_text(angle=90,
                                           va='top',
                                           size=6))
```

You should not go for areas as your first option for comparison purposes; however, you may try them and see if some pattern appears. Let me use a set of density plots, also known as *ridge plots*:

```
> baseTR = ggplot(allCrimes,
+                 aes(x = count)) + theme_void()
> tsDens = baseTR + geom_density(fill='grey', color=NA)
> tsRidge= tsDens + facet_grid(year(dates)~.) #lubridate
> tsRidge= tsRidge + theme(axis.text.x   = element_text())
```

The object `tsRidge` is plot in Figure 5.19.

The **R** version extracted the years directly in the `facet_grid` command using the function `year` from *lubridate*. The **Python** version is almost identical, but the year will be recovered as an attribute using `dt.year`:

```
baseTR  = ggplot(allCrimes,
                 aes(x = 'counts')) + theme_void()
tsDens  = baseTR + geom_density(fill='grey')
tsRidge = tsDens + facet_grid("allCrimes.dates.dt.year~")
tsRidge += theme(axis_text_x   = element_text())
```

In this case, this plot can complement the previous ones. Also, think how you can adapt Figure 5.14 to represent this time series data.

5.3.2 Correlation

The study of bivariate relationships among numerical variables is known as correlation analysis. The data we have been using has few numerical columns, but I will produce two by aggregating the original data since 2015 by neighborhood.

Let me first prepare my subset of data:

```
> #subsetting the data:
> crime2015=crime[crime$year≥2015,]
> # keeping non missing
> crime2015=crime2015[complete.cases(crime2015),]
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

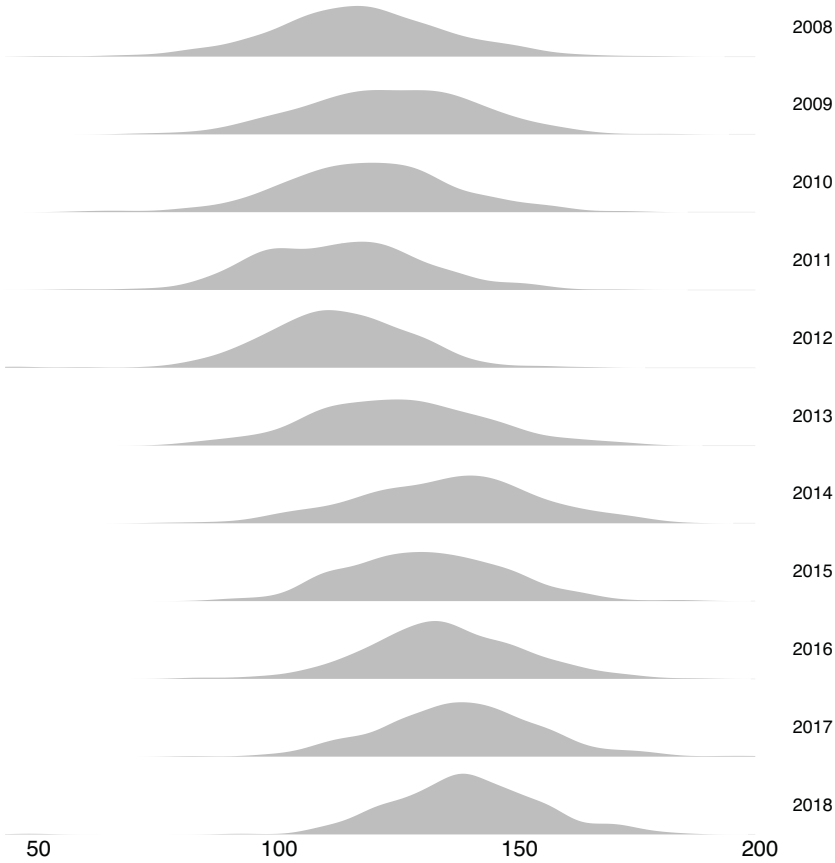

Figure 5.19 Time data as a ridge
Each year is a density plot. Data from Seattle Open Data Portal (City of Seattle, 2019).

You can do the same in **Python** like this:

```
baseTR  = ggplot(allCrimes,
                 aes(x = 'counts')) + theme_void()
tsDens  = baseTR + geom_density(fill='grey')
tsRidge = tsDens + facet_grid("allCrimes.dates.dt.year~")
tsRidge += theme(axis_text_x   = element_text())
```

Now, I want a data frame, `num_num`, where every row is a neighborhood. This will have two columns, the first one will have the mean of the time it takes to report a crime; and the second will have the share of total crimes.

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

Let me create this first:

```
> num_num= crime2015%>%
+              group_by(Neighborhood) %>%
+                  summarise(meanDaysToReport=mean(DaysToReport),
+                            CrimeShare=length(Neighborhood))
> head(num_num)
```

```
# A tibble: 6 x 3
  Neighborhood    meanDaysToReport CrimeShare
  <chr>                      <dbl>      <int>
1 ALASKA JUNCTION             3.62       2901
2 ALKI                        3.92        838
3 BALLARD NORTH               4.25       4177
4 BALLARD SOUTH               3.84       6274
5 BELLTOWN                    2.50       5283
6 BITTERLAKE                  4.38       3504
```

As you see, the second column represents the count, not the share. Let me get it:

```
> num_num=num_num%>%
+              mutate(CrimeShare=100*CrimeShare/sum(CrimeShare))
> #you get
> head(num_num)
```

```
# A tibble: 6 x 3
  Neighborhood    meanDaysToReport CrimeShare
  <chr>                      <dbl>      <dbl>
1 ALASKA JUNCTION             3.62       1.53
2 ALKI                        3.92       0.443
3 BALLARD NORTH               4.25       2.21
4 BALLARD SOUTH               3.84       3.32
5 BELLTOWN                    2.50       2.79
6 BITTERLAKE                  4.38       1.85
```

The steps to get the `num_num` data frame in **Python** follows:

```
# operation to perform in each column:
operations={'DaysToReport': 'mean','Neighborhood': 'count'}
# grouping and applying operation
num_num=crime2015.groupby('Neighborhood').agg(operations)
# computing the total crimes
sumOfCrimes=num_num.Neighborhood.sum()
# overwriting column
num_num['Neighborhood']=100*num_num.Neighborhood/sumOfCrimes
# renaming data frame
num_num.columns=['meanDaysToReport','CrimeShare']
# Neighborhood is the index (row names),
# moving it to a column
num_num.reset_index(inplace=True)
```

Having two numeric columns, you can produce a scatter plot (see Figure 5.20):

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

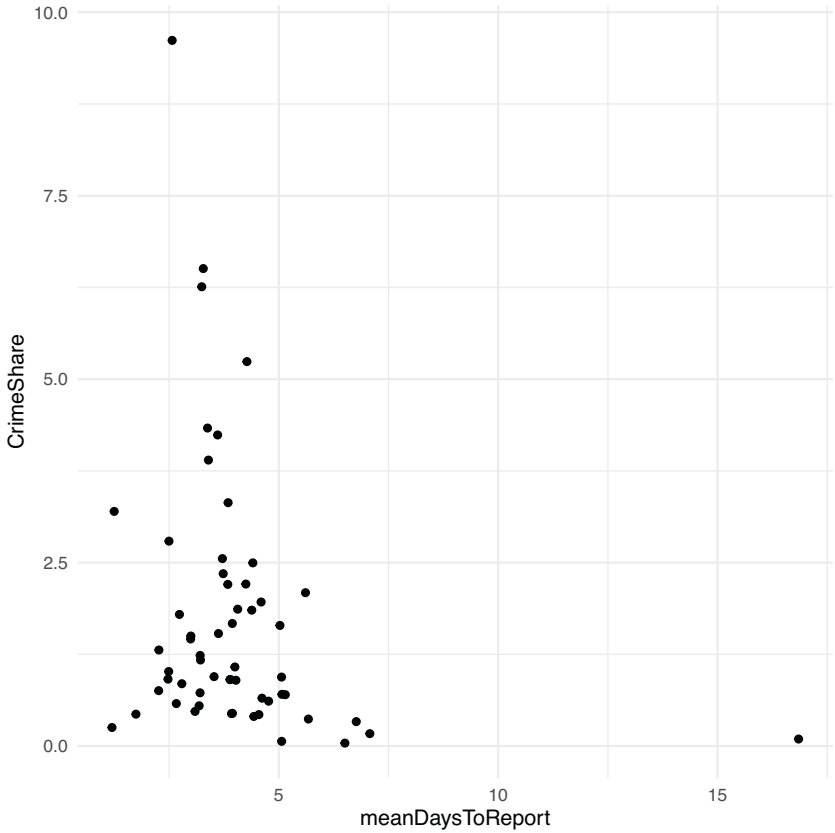

Figure 5.20 Basic scatter plot

The values of each variable serves to determine the position of each point; points represent a neighborhood. Data from Seattle Open Data Portal (City of Seattle, 2019).

```
> baseNN = ggplot(num_num,
+                 aes(x=meanDaysToReport,
+                     y=CrimeShare)) + theme_minimal()
> scat1 = baseNN +  geom_point(color='black')
```

Once you have a scatter plot, the usual next step is to highlight the level of correlation between the variables. This requires the computation of a correlation coefficient and its significance.

```
> # coefficient
> corVal=cor.test(num_num$meanDaysToReport,
+                 num_num$CrimeShare,method = 'spearman')$estimate
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

```
> # significance
> pVal=cor.test(num_num$meanDaysToReport,
+               num_num$CrimeShare,method = 'spearman')$p.value
> # rounding
> corVal=round(corVal,2); pVal=round(pVal,2);
> # building message
> TextCor=paste0('Spearman:\n',corVal,'\n(p.value:',pVal,')')
```

The **Python** code to the object `TextCor` follows:

```
import scipy.stats  as stats

corVal,pVal=stats.spearmanr(num_num.meanDaysToReport,
                            num_num.CrimeShare)

corVal=str(round(corVal,2))
pVal=str(round(pVal,2))
TextCor='Spearman:\n' + corVal +'\n(p.value:'+ pVal +')'
```

You can then produce an annotated scatter plot this way:

```
> scat2=scat1 + geom_smooth(method = lm,
+                           se=FALSE,
+                           color='grey60')
> scat2=scat2 + annotate(label=TextCor,
+                        geom = 'text',
+                        x=10,y=5)
```

The object `scat2` is plotted in Figure 5.21.

Scatter plots are easy to understand for people with a basic statistical background; for that reason, be careful with the annotations I included in Figure 5.21. That plot could be good for teaching purposes, but not for a short presentation to an audience that may not understand what *p.values* or correlation coefficients are.

You can use **Python** to replicate Figure 5.21 this way:

```
baseNN = ggplot(num_num,
                aes(x='meanDaysToReport',
                    y='CrimeShare')) + theme_minimal()
scat1  = baseNN +  geom_point(color='black')
#
scat2  = scat1 + geom_smooth(method = 'lm',
                         se=False,
                         color='silver')
scat2 += annotate(label=TextCor,
                      geom = 'text',
                      x=10,y=5)
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

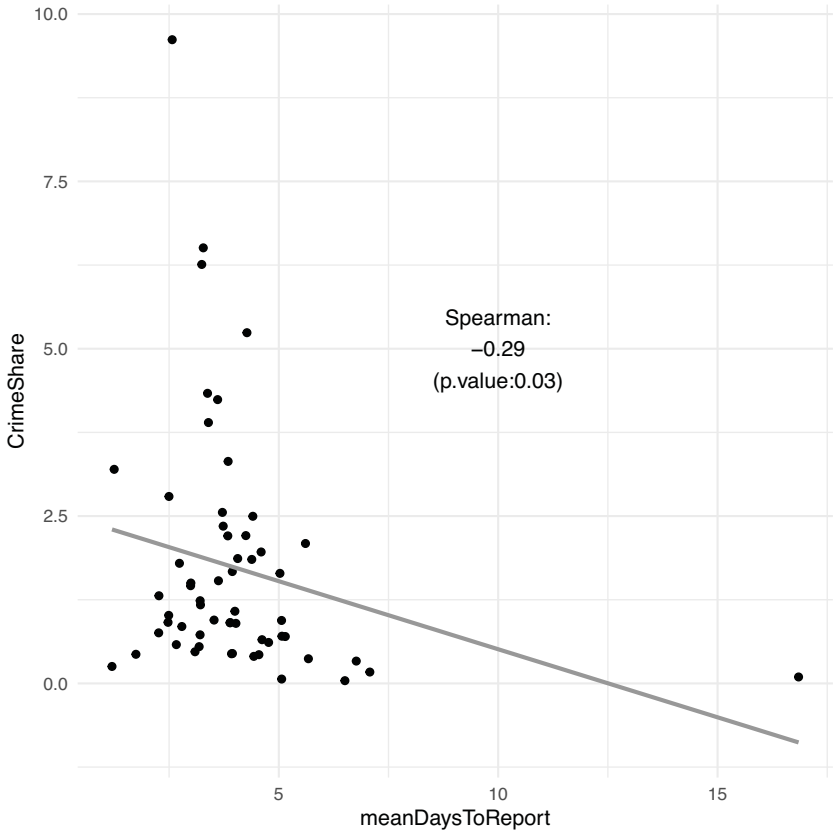

Figure 5.21 Scatter plot with correlation coefficient
The line has been computed using a lineal regression, and the text represents the Spearman's *rho*. Data from Seattle Open Data Portal (City of Seattle, 2019).

5.3.3 Focusing

Beyond showing the weak negative correlation between these variables, you can show some important information. In this case, as I have identified outlying values, I could try computing which they are using a linear regression:

```
> #regression
> relationship=lm(CrimeShare~meanDaysToReport,data = num_num)
> #save cook distance in data frame
> num_num$cook=cooks.distance(relationship)
> #compute threshold
> threshold=4/nrow(num_num)
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

I have computed the *Cook* distance (Cook, 1979) and saved it as another column in my data frame. I will use that value to create a condition:

```
> # 'cond' will be the names of the neighborhood is 'cook'
> # is above the 'threshold'
> cond=ifelse(num_num$cook>threshold,
+             num_num$Neighborhood, "")
```

The **Python** code for both previous steps is shown next:

```
#library needed
from statsmodels.formula.api import ols

#regression
relationship = ols('CrimeShare~meanDaysToReport',num_num).fit()
#influential values
influences = relationship.get_influence()
#saving Cook distance
num_num['cook'], pval = influences.cooks_distance
#computing thresold
threshold=4/len(num_num)
# condition
condition=np.where(num_num['cook'] > threshold,
                   num_num['Neighborhood'],"")
```

Notice that in this case, I have called the *statsmodel* library (Seabold and Perktold, 2010), whose function `get_influence` will return some influential measures. I just saved the Cook distance. Notice you will get two values from `cooks_distance`; I have just saved the first one as a new column (I got rid of the *p-values*)

And then, with help of the library *ggrepel* (Slowikowski et al., 2020). I can make this code:

```
> library(ggrepel)
> scat3 = scat1 + geom_text_repel(aes(label=cond),
+                                 color='grey50')
```

Plotnine does not have a native solution like *ggrepel*; however, if you have at least *plotnine* version**6**, you can mimic the repelling text in Figure 5.22 by installing *adjustText* (Flyamer, 2019) and preparing a dictionary like this:

```
# needs installation: pip install adjustText
from adjustText import adjust_text

## parameters as a dictionary
# 'expand_objects'expand the bounding box of
# texts when repelling them from other object
# 'arrowstyle' can also be '->' or '←'
adjustParams= {'expand_points': (0,0),
               'arrowprops': {'arrowstyle': '-',
                              'color': 'silver'}}
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

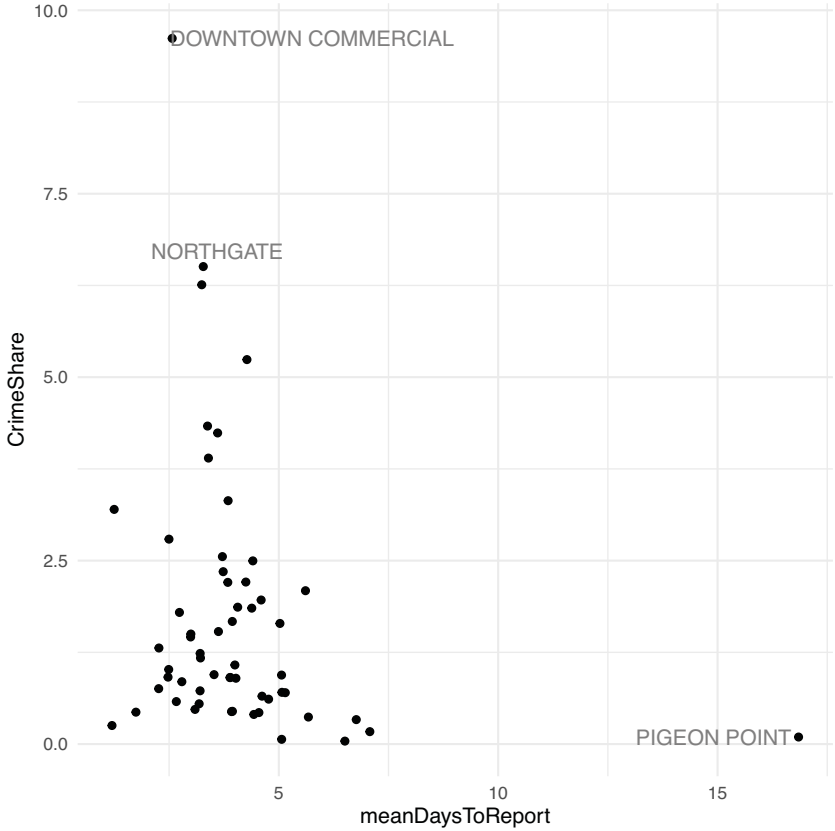

Figure 5.22 Scatter plot with annotated outliers

Outliers have been computed using the *cook distance* from a linear regression model. Data from Seattle Open Data Portal (City of Seattle, 2019).

You can get a similar result to Figure 5.22 using the following code:

```
scat3 = scat1 + geom_text(aes(label=condition),
                          adjust_text=adjustParams) #repel!
```

5.3.4 Concentration

The symbols in the scatter plot are usually plotted on top of each other; so while outliers are easy to identify, areas where several symbols are overplotted may hide some information. One smart way to solve this is abandoning the use of dots and trying alternative ways. Let me first compute the most common values in our bivariate relationship for annotating my final visual:

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

```
> # library(dplyr) for function "between"
> xVals=num_num$meanDaysToReport
> yVals=num_num$CrimeShare
> tVals=num_num$Neighborhood
> num_num$CondText=ifelse(between(xVals,3, 5) &
+                         between(yVals,1, 3),
+                         tVals,"")
```

I have created the variable `CondText`, which will have a neighborhood names only if the other numerical variables are within a data range. The function `between` from *dplyr* (Wickham et al., 2020a) came in handy for that. Let me show you how to achieve the same with Python:

```
xVals=num_num.meanDaysToReport
yVals=num_num.CrimeShare
tVals=num_num.ConText


num_num['CondText']=np.where(xVals.between(3, 5) &
                             yVals.between(1, 3),
                             tVals,None)
```

Now that I have created a new variable, I need to reload the data frame:

```
> baseNN_Re = ggplot(num_num,
+                    aes(x=meanDaysToReport,
+                        y=CrimeShare)) + theme_minimal()
```

This will create an *hexabin plot*, which consists of hexagons whose width values represent the interval width in each axis, and whose color intensity will represent the count of cases in each hexagon; in this case, darker the more cases there are (you can represent the opposite by changing `direction` to -1).

```
> scatHex = baseNN_Re +  geom_hex(binwidth = 1)
> scatHex = scatHex + scale_fill_distiller(palette ="Greys",
+                                          direction=1)
```

The object `scatHex` will show you in Figure 5.23 where the cases are concentrated.

You can annotate the Figure 5.23. However, if your purposes is to label the cities in the darkest hexagons, consider this:

- Text will be cluttered, so you will need to zoom into the darkest area.
- Zooming in will make areas disappear (outliers will not be seen).
- You may need to highlight the text, because you will get closer to darkest areas which might hide some hexagons.

Let me illustrate those considerations:

```
> #Zooming in:
> scatHexAn = scatHex + ylim(c(1,4)) + xlim(c(1.5,7))
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

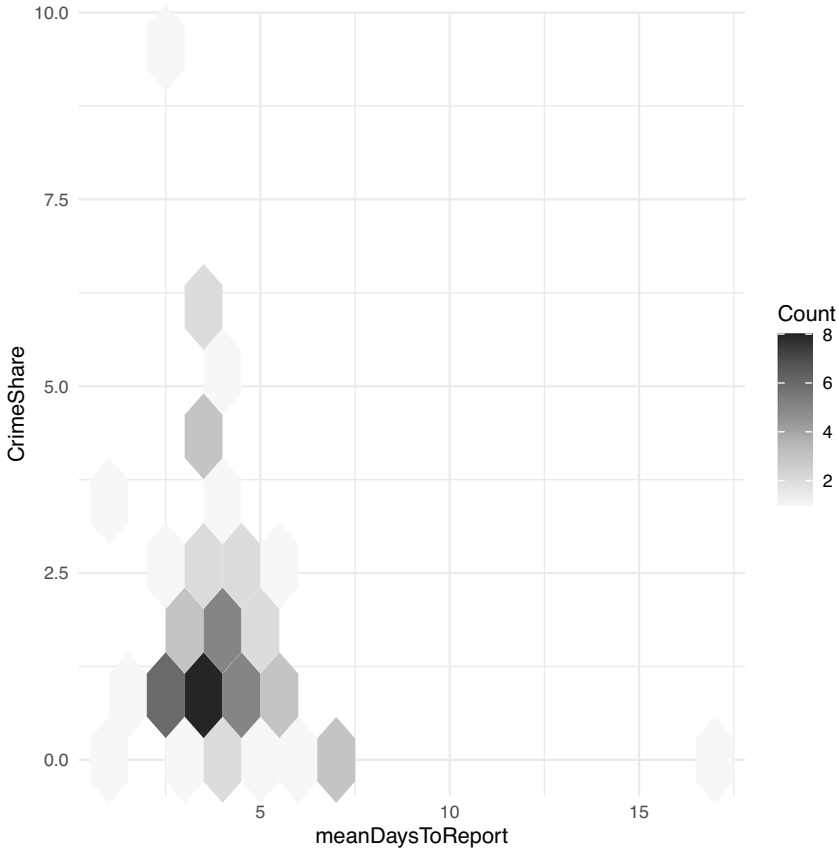

Figure 5.23 Basic Hexabin plot

This shows you the values where neighborhoods are concentrated. Data from Seattle Open Data Portal (City of Seattle, 2019).

```
> #Annotating:
> scatHexAn = scatHexAn + geom_label_repel(aes(label=CondText),
+                                          size=2.5,
+                                          color='grey50')
```

Figure 5.24 represents object `scatHexAn`.

Hexabin plots can be done in **Python**, but at the beginning of year 2020 it was not implemented in *plotnine*. Alternatives such as *seaborn*, or *Pandas* itself, facilitate making easy and fast hexabin plots, but lack some customization details. So, my best alternative is to use *matplotlib* and build it from scratch. Let me build this plot in several steps, until I reach Figure 5.24.

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

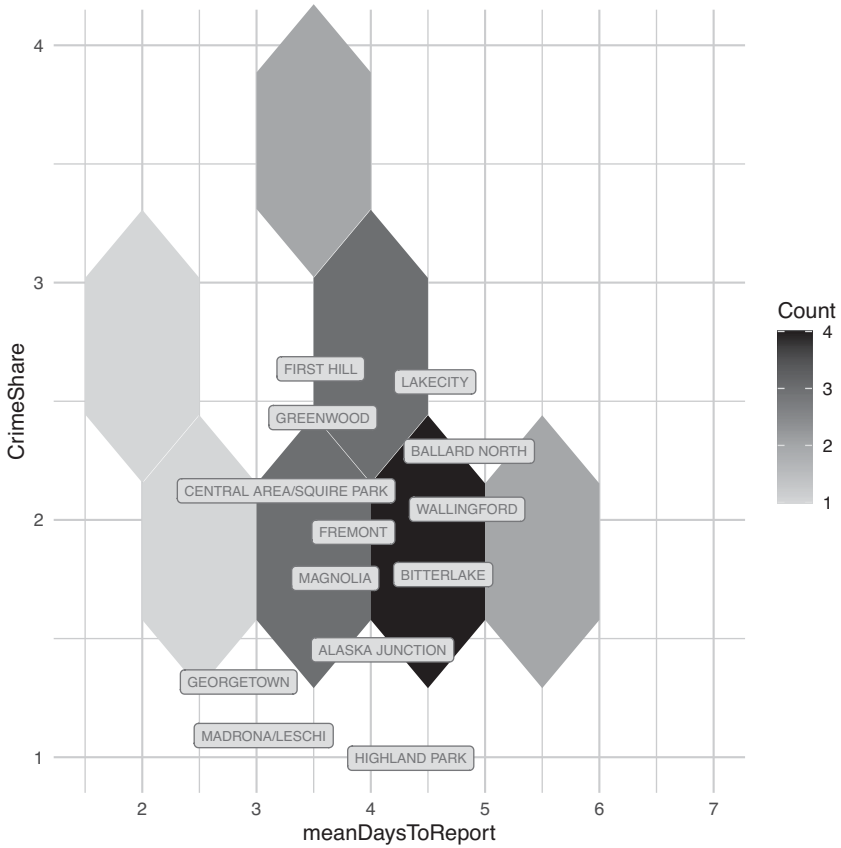

Figure 5.24 Annotated Hexbin plot

This plot shows you the locations where neighborhoods are concentrated. The plot has been zoomed-into to make labels visible. Data from Seattle Open Data Portal (City of Seattle, 2019).

The output of this **Python** code will closely resemble Figure 5.23:

```
import matplotlib.pyplot as plt

# size of plot
plt.figure(figsize=(10,6))

# plot hexabin
plt.hexbin(xVals,
           yVals,
           cmap=plt.cm.Greys, #colormap
           gridsize=10,
           mincnt=1) # at least

# color bar that represents counts
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

```
TheLegend = plt.colorbar()
TheLegend.ax.set_title('Legend\nTitle')
```

The previous code shows how every step uses **plt**, which represents most of the *matplotlib* functionality. Most is pretty straightforward: setting the size (the plot may be small by default), calling the *hexabin* function, and adding a color bar for legend. However, pay attention to color selection. The argument `cmap` is calling a color map from *matplotlib*. When you decide on a color map, get the name and append it at the end of `plt.cm.` You can find the color bars available on *matplotlib*'s webpage.[10] Notice that *matplotlib* uses the argument `gridsize`, which will be used to divide the plotting space and locate the hexagons, so that the greater the value the smaller the hexagon (the opposite of **R**). Finally, the argument `mincounts` tells you the minimum counts that will make a hexagon exist.

The previous code will not control the background appearance, as we used to do it with `theme_classic` or `theme_minimal`. This adaptation will help you achieve that:

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10,6))
# theme
plt.grid(color='silver') #lines color
plt.gca().set_facecolor("white") #background
#

plt.hexbin(xVals,
           yVals,
           cmap=plt.cm.Greys,
           gridsize=10,
           mincnt=1)
TheLegend = plt.colorbar()
TheLegend.ax.set_title('Legend\nTitle')
```

The last code gives me a good opportunity to share the difference between a *figure* and an *axis*. While a *figure* is the whole of what you see, including the space for the legend, the *axes* is the plotting area. Some properties have similar names in both, but that will create a different behavior. Obviously, figures have properties that axes do not, and vice versa. The function **gca()** recovers the current *axes* and makes changes there, so the function **set_facecolor** is an operation on *axes*, not on *figures*: this will change the background or the patches behind the grid. Also, notice that my color bar is an *axes* element too, so I give that axes a title.

The previous code can be adapted to perform the zooming in and the annotation. We need a couple of pieces. First, the zooming in:

[10] https://matplotlib.org/examples/color/colormaps_reference.html

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

```
#ZOOMING IN
plt.axis([1.5, 7, 1, 3])
```

That step was easy. The first two values of the list `[1.5, 7, 1, 3]` are the minimum and maximum of the horizontal, and the last two are the minimum and maximum of the vertical.

A harder piece of code is the annotation section. The annotations will use *labels* as we did in **R**. Labels have bounding boxes, This code will set up the bounding boxes:

```
## settings of the label bounding box
for_bbox = {'boxstyle':"round",
              'fc':"white"}
```

Then, I need to prepare the labels, which requires two steps. The first one is the creation of the labels. I will use the `text` function, a function that needs as input **each** coordinate of the text. So I need a **loop**:

```
## labels with bounding box
labels=[] #empty list of labels
for i in num_num.index:
    #creating each label
    labels.append(plt.text(xVals[i],yVals[i],
                           num_num.CondText[i],
                           color='gray',
                           fontsize=10,
                           bbox=for_bbox))
```

The loop visits each index, and uses that index, represented as `i` to retrieve the values for the coordinates (the numeric values) and the texts. Notice I am using the argument `bbox` explicitly. At this point, I need to repel the labels, so I will use again the `adjust_text` function:

```
## repelling text
adjust_text(labels,expand_text=(1.5, 1.5))
```

The only thing missing will the caption and titles. We have done that before in Subsection 3.3. The only thing you need to adapt in situations like the current one is the **x** coordinate of the caption:

```
## titles
plt.suptitle('The TITLE with matplotlib', y=1, fontsize=18)
plt.title('The SUBTITLE with matplotlib', y=1, fontsize=12)
plt.text(x=1.5,y=0.5,s="The Caption with matplotlib")
plt.xlabel('x-label with matplotlib')
plt.ylabel('y-label with matplotlib')
```

This final code integrates the previous pieces to produce the hexabin plot:

```
plt.figure(figsize=(10,6))
plt.grid(color='silver')
plt.gca().set_facecolor("white")
#
#ZOOMING IN
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

```
plt.axis([1.5, 7, 1, 3])
#
plt.hexbin(xVals,
           yVals,
           cmap=plt.cm.Greys,
           gridsize=10,
           mincnt=1)
#
#
# ANNOTATING
## settings of the label bounding box
for_bbox = {'boxstyle':"round",
             'fc':"white"}
## labels with bounding box
labels=[] #empty list of labels
for i in num_num.index:
    #creating each label
    labels.append(plt.text(xVals[i],yVals[i],
                           num_num.CondText[i],
                           color='gray',
                           fontsize=10,
                           bbox=for_bbox))
## repelling text
adjust_text(labels,expand_text=(1.5, 1.5))
#
#
TheLegend = plt.colorbar()
TheLegend.ax.set_title('Legend\nTitle')

## titles
plt.suptitle('The TITLE with matplotlib', y=1, fontsize=18)
plt.title('The SUBTITLE with matplotlib', y=1, fontsize=12)
plt.text(x=1.5,y=0.5,s="The Caption with matplotlib")
plt.xlabel('x-label with matplotlib')
plt.ylabel('y-label with matplotlib')
```

Another option for representing concentration is the *density plot*. The process is very similar to create object `scatHexAn`, you just replace function `geom_hex` with `stat_density_2d` using the parameters given, which will compute the density and represent it as a *raster* without *contours*.

```
> #limits
> scatDenAn = baseNN_Re + ylim(c(1,4)) + xlim(c(1.5,7))
> #palette
> scatDenAn= scatDenAn + scale_fill_distiller(palette="Greys",
+                                             direction=1)
> #2d density
> scatDenAn = scatDenAn +  stat_density_2d(aes(fill = ..density..),
+                                          geom = "raster",
+                                          contour = FALSE)
> #repelling text
> scatDenAn = scatDenAn + geom_label_repel(aes(label=CondText),
+                                          size=2,
+                                          color='grey50')
> #no legend
> scatDenAn = scatDenAn +  theme(legend.position='none')
```

Notice that I am not requesting a legend as it will not inform the counts, but the share of points. You can keep it by not using the last line.

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

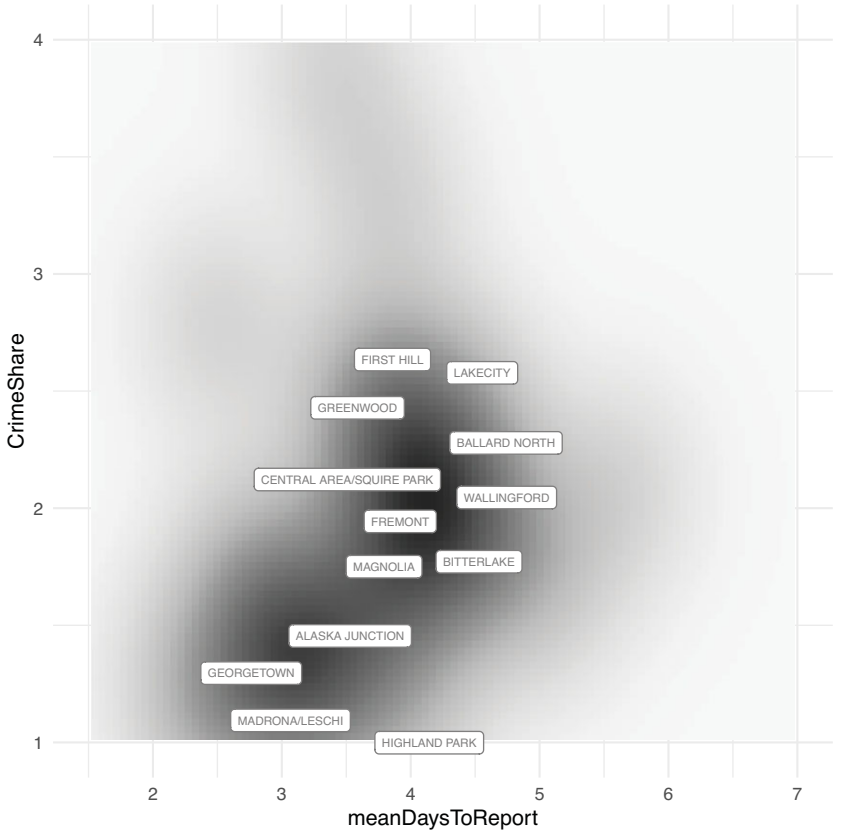

Figure 5.25 Annotated 2D-density plot

This plot lets you know the locations where neighborhoods are concentrated. The plot has been zoomed-into to make labels visible. Data from Seattle Open Data Portal (City of Seattle, 2019).

Let me replicate Figure 5.25 using **Python** and *matplotlib* again:

```
# libraries
import matplotlib.pyplot as plt
import numpy as np
from scipy.stats import kde

#LIMITS, THEME and ZOMING
plt.figure(figsize=(10,6))
plt.grid(color=None)
plt.gca().set_facecolor("white")
plt.axis([1.5, 7, 1, 3])
#
#
# MAKING THE DENSITY PLOT
## dividing plotting area
```

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press

```
nbins=300
## computing densities
k = kde.gaussian_kde([xVals,yVals])
## preparing grid
xi, yi = np.mgrid[xVals.min():xVals.max():nbins*1j,
                  yVals.min():yVals.max():nbins*1j]
## preparing weights
zi = k(np.vstack([xi.flatten(), yi.flatten()]))
## preparing color palette
plt.pcolormesh(xi, yi, zi.reshape(xi.shape), cmap=plt.cm.Greys)
#
#
# ANNOTATING
for_bbox = {'boxstyle':"round",
            'fc':"white"}
labels=[]
for i in num_num.index:
    labels.append(plt.text(xVals[i],yVals[i],
                           num_num.CondText[i],
                           color='gray',
                           fontsize=10,
                           bbox=for_bbox))
adjust_text(labels,expand_text=(1.5, 1.5))
#
#
#TITLES
plt.suptitle('The TITLE with matplotlib', fontsize=18)
plt.title('The TITLE with matplotlib', fontsize=12)
plt.text(x=1.5,y=0.75,s="The Caption with matplotlib")
plt.xlabel('x-label')
plt.ylabel('y-label')
plt.grid(color='silver')
plt.show()
```

The code above shows you that, apart from the 2D-density plot preparation, the other elements were retained. Keep in mind that plots that show concentration have several variants, but the codes shared here can be easily adapted in case you require other alternatives.

https://doi.org/10.1017/9781108625425.007 Published online by Cambridge University Press