7

Geospatial Data

7.1 Data Suitable for Maps

Any data frame that gives you some spatial information is suitable for being represented in a map. These are some options:

- An address or a pair of coordinates: school address, airport address, home address, etc.
- An administrative unit: county, district, region, country, etc.
- A segment: roads, streets, routes, etc.

Addresses or coordinates (latitude-longitude) are generally represented using points; administrative units are generally represented using polygons;[1] segments are generally represented using lines. Points, polygons and lines are generally stored in *shapefiles*.

Let me show you some data I have collected on contributions to candidates and political committees in Washington State. These data are available at the WA State portal for open data.[2] Let's use this code to open the data in **R**:

```
> link1='https://github.com/resourcesbookvisual/data/'
> link2='raw/master/contriWA.RDS'
> LINK=paste0(link1,link2)
> #getting the data TABLE from the file in the cloud:
> contriWA=readRDS(file=url(LINK))
```

You can open it in **Python** like this (notice this is a CSV file):

```
import pandas as pd

link1='https://github.com/resourcesbookvisual/data/'
link2='raw/master/contriWA.csv'
contriWA=pd.read_csv(link1+link2)
```

[1] Points can also be used but this may not be an address but a representative coordinate, such as a 'centroid'.

[2] https://data.wa.gov/Politics/Contributions-to-Candidates-and-Political-Committe/kv7h-kjye

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

The data frame looks like this:

```
            id contributor_state contributor_zip amount election_year
1 6229365.rcpt                WA           98501 106.00          2019
2 6229366.rcpt                WA           98501 110.00          2019
3 6229367.rcpt                WA           98501  69.99          2019
4 6229368.rcpt                WA           98501  40.98          2019
5 6229369.rcpt                WA           98103 100.00          2019
6 6229371.rcpt                WA           98107  20.00          2019
         party cash_or_in_kind   contributor_location      Lat       Lon
1 NON PARTISAN         In kind (47.02872, -122.87765) 47.02872 -122.8777
2 NON PARTISAN         In kind (47.01362, -122.87553) 47.01362 -122.8755
3 NON PARTISAN         In kind (47.01362, -122.87553) 47.01362 -122.8755
4 NON PARTISAN         In kind (47.01362, -122.87553) 47.01362 -122.8755
5 NON PARTISAN            Cash (47.67672, -122.35165) 47.67672 -122.3517
6 NON PARTISAN            Cash (47.66897, -122.40436) 47.66897 -122.4044
```

These data look promising for using maps. First, it has information at the administrative level: the zip code (the state could be used, but not in this case, as it only has information from WA). Second, the location of the contributor, represented by the last three columns. Notice that each row is a contribution from somebody. Someone may have contributed several times, but in this data frame we do not have a way to identify a person; however, this is a simplified version, as the original data available at the open data portal does have the information of the contributor. Let's see the data types:

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

```
'data.frame':   3096599 obs. of  10 variables:
 $ id                  : chr  "6229365.rcpt" "6229366.rcp"..
 $ contributor_state   : chr  "WA" "WA" "WA" "WA" ...
 $ contributor_zip     : chr  "98501" "98501" "98501" "98"..
 $ amount              : num  106 110 70 41 100 ...
 $ election_year       : int  2019 2019 2019 2019 2019 201..
 $ party               : Factor w/ 9 levels "","CONSTITUT"..
 $ cash_or_in_kind     : Factor w/ 2 levels "Cash","In ki"..
 $ contributor_location: chr  "(47.02872, -122.87765)" "(".."
 $ Lat                 : num  47 47 47 47 47.7 ...
 $ Lon                 : num  -123 -123 -123 -123 -122 ...
```

The data types seem right: the location, the contribution and the year have been read as numbers, and the zip codes have been read as strings. The other variables are in the right format, too. However, this data frame is not a map.

7.2 Map Files

Maps are available in different formats. But, you might have heard of the *shapefile* (Environmental Systems Research Institute, 1998). The shapefile

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

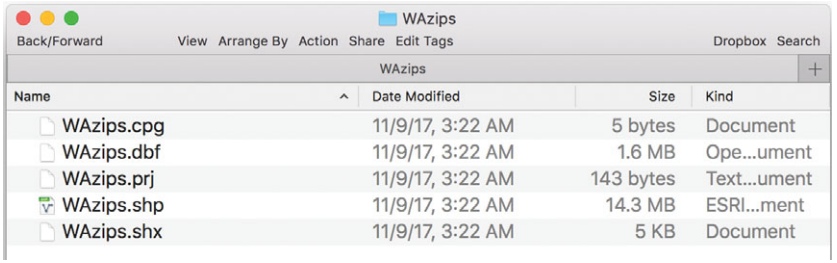


Figure 7.1 Contents of shapefile folder
The file was downloaded from the Office of Financial Management from Washington State.

has something that may sound confusing for the basic user: it is actually a collection or system of files. So, when you download or someone shares a shapefile with you, you actually need to get all those files. As the name implies, you get shapes from this file collection representing either lines, points or polygons.

For the data we have, we need a map where each element (shape) represents a zip code. So, I searched on Google for *Washington zipcode shapefile* and found one available.[3] I downloaded the file and the folder looks like Figure 7.1.

R can not read shapefile from an online repository as *GitHub*, which we have been using along this book, so the alternative is to read it from a folder on your computer:

```
> folderMap="WAzips"
> fileMap="WAzips.shp"
> locationMap=file.path(folderMap,fileMap)
```

The file in your computer can be read using the *sf* library (Pebesma, 2018). Once it is installed in **R**, you can simply call the file like this:

```
> library(sf)
> wazipMap=st_read(locationMap)
```

The function `st_read` creates a data frame with special a column: *geometry*, never get rid of it. That function will also send some information (see Figure 7.2). The *geometry type* informs your map and is made of polygons; the *dimension* confirms it is bidimensional; the *bbox* informs the corners of the rectangle in which the map is represented. The last two ones (*EPSG* and *proj4string*) inform the default coordinate system and projection of the map you have. Remember the world is not flat, so any bidimensional map

[3] www.ofm.wa.gov/washington-data-research/population-demographics/gis-data/census-geographic-files

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

```
geometry type:  MULTIPOLYGON
dimension:      XY
bbox:           xmin: -124.7428 ymin: 45.54354 xmax: -116.9156 ymax: 49.0025
epsg (SRID):    4326
proj4string:    +proj=longlat +datum=WGS84 +no_defs
```

Figure 7.2 Summary of spatial file

The original shapefile was downloaded from the Office of Financial Management from Washington State.

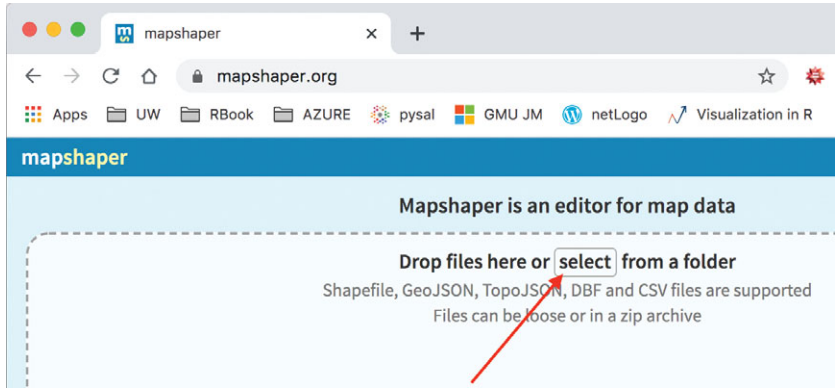


Figure 7.3 From SHP to *geojson* (1)

This steps requires you visit the website of **mapshaper** at https://mapshaper.org/. The original shapefile was downloaded from the Office of Financial Management from Washington State.

is a distorted representation of reality. Of course, there are several projection possibilities. I will use the default to avoid dealing further with this matter.

I will read my map file from *GitHub*. A possible direction will be to transform the original shapefile into another format: **geojson** (Butler et al., 2016). Let me follow these steps:

1. Go to the website of *mapshaper*.[4] As I show in Figure 7.3, click on **select**.
2. When prompted, as shown in Figure 7.4, go to the folder with the shapefile and select all of them.
3. You can click **import** (see Figure 7.5) after you see *mapshaper* has selected the files needed. It may select less files than the ones you have in the folder.
4. After the last step, you will see the map of Seattle, where each polygon represents a zip code area. You will see the option to *simplify* the map,

[4] https://mapshaper.org/

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

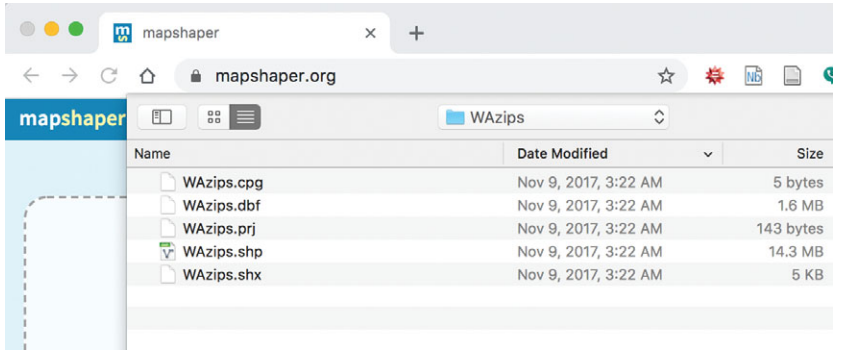


Figure 7.4 From SHP to *geojson* (2)

This steps requires you select all the files in the folder of the shapefile to be upoloaded to **mapshaper**. The original shapefile was downloaded from the Office of Financial Management from Washington State.

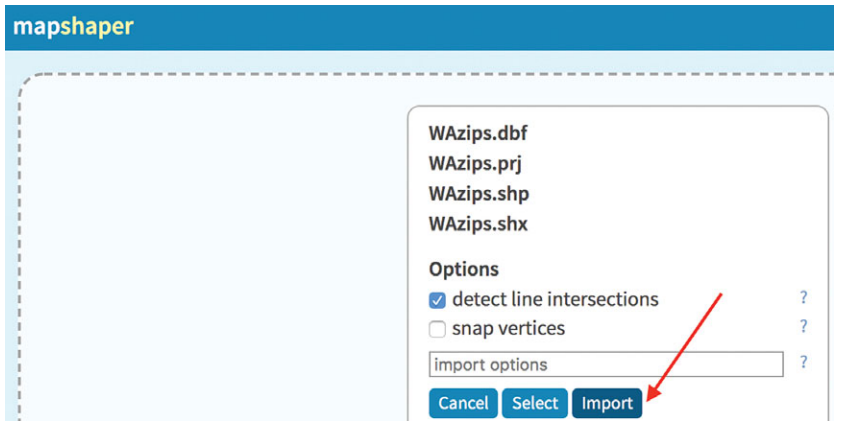


Figure 7.5 From SHP to *geojson* (3)

In this step, you simply import the files that mapshaper requires. The original shapefile was downloaded from the Office of Financial Management from Washington State.

which means you can optimize the current representation and get a map with a smaller size in terms of space occupied in bytes. If you want, you can simply leave the map as it is. When you are done, just click on **export** (see Figure 7.6).

5. At the export menu, select **geojson**, and then **Export**, as shown in Figure 7.7.

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

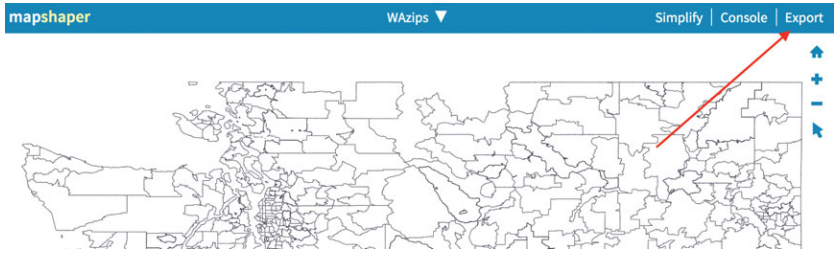


Figure 7.6 From SHP to *geojson* (4)
This steps requires you simply select the **export** option. The original shapefile was downloaded from the Office of Financial Management from Washington State.

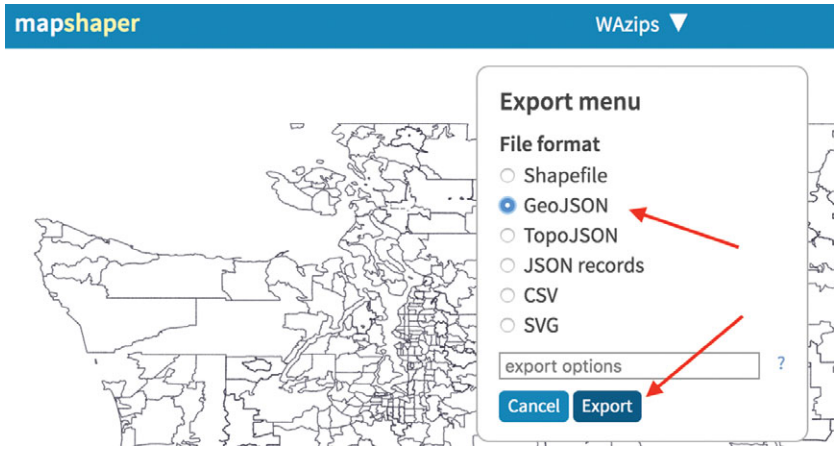


Figure 7.7 From SHP to *geojson* (5)
This steps requires you select the *geojson* option then select the **export** option. The original shapefile was downloaded from the Office of Financial Management from Washington State.

6. *Mapshaper* will ask you to save the file in **geojson** format in your computer. Save it anywhere, and then upload that file to *GitHub*. When it is in *GitHub*, click on the file name so that you see something similar to Figure 7.8. Once you are in that webpage, go to the download button and get the link by right-clicking on it.

By now you have copied a link to the *geojson* file. Let me save the link in an object in my code:

```
> myGit="https://github.com/resourcesbookvisual/data/"
> myGeo="raw/master/WAzipsGeo.json"
> mapLink=paste0(myGit,myGeo)
```

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

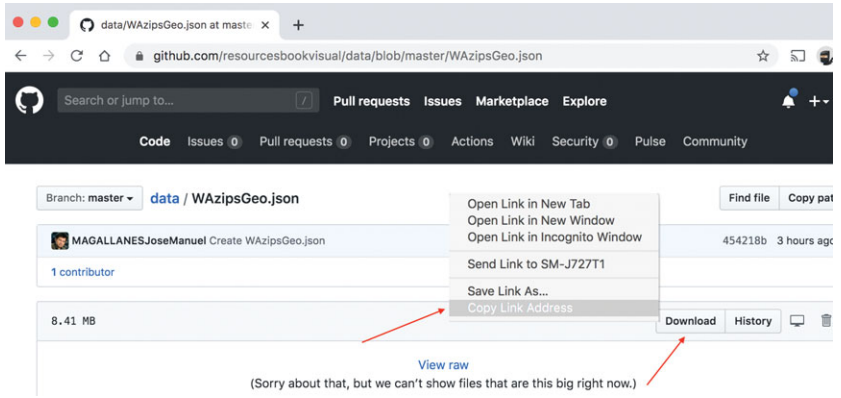


Figure 7.8 From SHP to *geojson* (6)

This step requires that you get the link of the *Geojson* file to be used in your code (R or Python). The original shapefile was downloaded from the Office of Financial Management from Washington State.

Once the link is in **R**, I can use the same functions from the *sf* package with the link to the *geojson* file in *GitHub*:

```
> library(sf)
> wazipMap=read_sf(mapLink)
```

Notice I used read_sf which is a popular alias to st_read in the sf package. I will keep using *ggplot* here, as it can work very well with the *sf* geojson object we have:

```
> library(ggplot2)
> base= ggplot(data=wazipMap) + theme_classic()
> basemap= base + geom_sf(fill='black', #color of polygon
+                         color=NA) #color of border
```

The object `basemap` is storing the visual map, which can be seen in Figure 7.9:

Python can recover the *geojson* file using *GeoPandas*, a library that has geospatial functions and which brings spatial data frames that can be accessed and manipulated like any *Pandas* data frame (Jordahl et al., 2020). This is how you load the file:

```
import geopandas as gpd

myGit="https://github.com/resourcesbookvisual/data/"
myGeo="raw/master/WAzipsGeo.json"
mapLink=myGit + myGeo

wazipMap = gpd.read_file(mapLink)
```

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

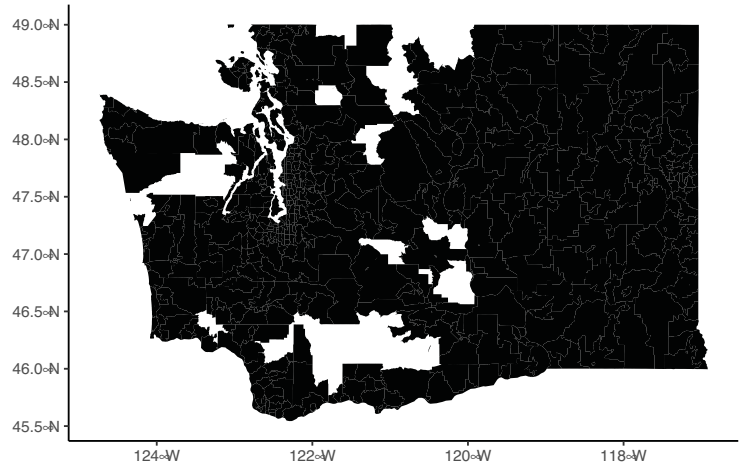


Figure 7.9 Simple map from *geojson* file
The geojson file was produced on the website of *mapshapper*.

Once you have a geodata frame, you can use the `plot` function in *geopandas* easily:

```
wazipMap.plot()
```

I recommend you also install *GeoPlot* (Bilogur et al., 2019), which may give some extra function for plotting *geopandas* dataframes:

```
import geoplot as gplt
gplt.polyplot(wazipMap)
```

7.3 Combining Maps and Data

Maps may come with data, in fact the object `wazipMap` has several columns (102). However, for combining maps and data, I have identified the column with the zip code: `ZCTA5CE10`,[5] let's see:

```
> str(wazipMap$ZCTA5CE10)
```

```
 chr [1:598] "98001" "98002" "98003" "98004" "98005" "98006" "98007" ...
```

You can see that in **Python** too:

```
wazipMap.ZCTA5CE10.describe()
```

[5] `ZCTA5CE10` stands form *2010 Census 5-Digit ZIP Code Tabulation Area*

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

However, while the map may not bring relevant columns for you, you can have a data frame (maybe a spreadsheet) with columns you wish to represent on a map. Then, recall you have information about contributions per zip code, and location of the contributor in Washington State (longitude and latitude). As a zip code is an administrative unit, you should work to put information about contributions in each polygon of `wazipMap`. The other alternative, the location coordinates of the contributor can not be merged into the `wazipMap` object, but can be an additional layer.

7.3.1 Merging

Merging data into maps is not different from common merging. As long as the map and the other data frame have a common column, the merging is possible. However, there is still some thinking. In our case, the data in `contriWA` has more than 3 million rows, and the map offers 598 polygons. In this situation, you need to aggregate the `contriWA` data by zip code. As `contriWA` has varied information, I need to do some filtering before aggregation, and then proceed to merge:

1. Filtering: Let's keep just the contributions in cash, directed to democrats or republicans, for the year 2012:

```
> library(dplyr)
> contriWAsub=contriWA %>%
+               filter(cash_or_in_kind=="Cash" &
+                      party%in%c("DEMOCRAT","REPUBLICAN") &
+                      election_year==2012)
```

   The object `contriWAsub` can be obtained in **Python** using the `query` function. I have chained two queries to improve readability:

```
#conditions
condition1='election_year==2012 and cash_or_in_kind=="Cash"'
condition2='party.isin(["DEMOCRAT","REPUBLICAN"])'

#chained queries
contriWASub=contriWA.query(condition1).query(condition2)
```

   Once you have the filtered data, you should explore what you have got:

   - The count of contributions given:[6]

```
> nrow(contriWAsub)
```

```
[1] 230717
```

[6] This dataset does not have information on the contributor, but the original one does.

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

- The summary of numerical variables `contriWAsub`:

```
> # Distribution of contributions:
> summary(contriWAsub$amount)
```

```
     Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
      0.0      36.0      75.0     208.2     200.0 1500000.0
```

- The frequencies in categorical data:

```
> # Destination of contributions:
> as.data.frame(table(contriWAsub$party))
```

```
                Var1   Freq
1                          0
2 CONSTITUTION PARTY       0
3           DEMOCRAT 118356
4        INDEPENDENT       0
5        LIBERTARIAN       0
6       NON PARTISAN       0
7               NONE       0
8              OTHER       0
9         REPUBLICAN 112361
```

- The count of zip codes:

```
> # Destination of contributions:
> length(unique(contriWAsub$contributor_zip))
```

```
[1] 785
```

You have more zip codes than the map. You can verify the distribution on the zip codes in the data frame:

```
> summary(unique(as.numeric(contriWAsub$contributor_zip)))
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  98001   98251   98528   98585   98933   99403
```

Then, you can compare the previous values to the distribution on the zip codes in the map:

```
> summary(unique(as.numeric(wazipMap$ZCTA5CE10)))
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  98001   98281   98576   98620   98952   99403
```

Both data have the same minimum and maximum values, so whatever is not matching is between those values. It is possible that new zip codes were created after 2010, but I will not deal with that.

Once you have explored the data, you should decide if your audience will need to see a map or not. You can make several plots with the data you have; but if you prepare a map, it is because your audience has familiarity with the geography represented. That is, if your audience is in Brazil (or even from Atlanta), the map of Washington State may not

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

be important; but if the audience is from King County, that map may be well received. If you are still planing to merge this data into the map, the next step will be to aggregate the information on contributions per zip area.

2. Aggregating. You have several rows (230,717) that could give a value to each of the polygons in the map (598). So, you need to create a "summary" at the zip level. The summary is a function of your choice (`mean`, `max`, `min`, etc), let me use `sum` to get the amount of money given to each party per zip, and the total amount per zip as well:

```
> contrisum=contriWAsub %>%
+           group_by(contributor_zip) %>%
+           summarise(REPUBLICAN=sum(amount[party=="REPUBLICAN"]),
+                     DEMOCRAT=sum(amount)-REPUBLICAN,
+                     total=sum(amount)) %>%
+           as.data.frame()
> # you get:
> head(contrisum)
```

```
  contributor_zip REPUBLICAN DEMOCRAT      total
1           98001   67723.12  44654.0  112377.12
2           98002   30372.22  19843.0   50215.22
3           98003  180465.00 242428.0  422892.95
4           98004  782610.28 291740.8 1074351.09
5           98005 2451978.61 147940.0 2599918.61
6           98006  326822.49 174507.0  501329.51
```

You can compute the object `contrisum` in **Python** like this:

```
contrisum=contriWASub.groupby(['contributor_zip','party'])
contrisum=contrisum.agg({'amount':'sum'}).reset_index().fillna(0)
contrisum=contrisum.pivot(index='contributor_zip',
                          columns='party',
                          values='amount').reset_index().fillna(0)
contrisum['total']=contrisum.DEMOCRAT + contrisum.REPUBLICAN
```

Even though you know by now that you do not have some zip codes, you are ready to merge the object `contrisum` into the map `wazipMap`, just keep in mind a couple of things:

- You could keep the columns you need from `wazipMap`, getting rid of what you do not need (this is optional):

```
> # keeping zip (column 2) and last column (the 'geometry')
>
> wazipMap=wazipMap[,c('ZCTA5CE10','geometry')]
```

- The map must go to the "left" (first object) during the merge (this is not optional):

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

```
> allZip=merge(wazipMap,contrisum,
+              by.x='ZCTA5CE10', by.y='contributor_zip',
+              all.x=TRUE)
```

You can get the map `allZip` in **Python** like this:

```
wazipMap=wazipMap.loc[:,['ZCTA5CE10','geometry']]
# contributor_zip as text (it was a number)
contrisum.contributor_zip=contrisum.contributor_zip.astype(str)
allZip=wazipMap.merge(contrisum,
                      left_on='ZCTA5CE10',
                      right_on='contributor_zip',
                      how='left')
```

The object `allZip` is a new map with the three extra columns on the counts of people contributing to the campaign. As we included the option `all.x=TRUE` in **R** or `how='left'` in **Python**, this merged map will have the same amount of polygons as the original map. Keep in mind that choosing that kind of merge can cause the new columns merged into the map to have missing values.

```
> #same amount of rows?
> nrow(allZip)==nrow(wazipMap)
```

```
[1] TRUE
```

Now that we have a map merged with information from another data frame, I can use the new object `allZip` as a normal data frame; for instance, let me create a simple variable:

```
> #creating the dissolving column
> allZip$winnerREP=allZip$REPUBLICAN >allZip$DEMOCRAT
```

Keep in mind that the new variable might have some missing values, as there might not have been some coincidences during merging:

```
> summary(allZip$winnerREP)
```

```
   Mode   FALSE    TRUE    NA's
logical     198     385      15
```

The missing values occurred when at least one `NA` was present in the values. *Pandas*, in **Python**, does not give missing values when the missing values are compared, so my code to create this variable is more verbose:

```
comparison=allZip.REPUBLICAN>allZip.DEMOCRAT
condition=allZip.loc[:,["REPUBLICAN","DEMOCRAT"]].any(axis=1)
allZip['winnerREP']=condition1
allZip['winnerREP']=np.where(condition,
                             comparison,
                             None)
```

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

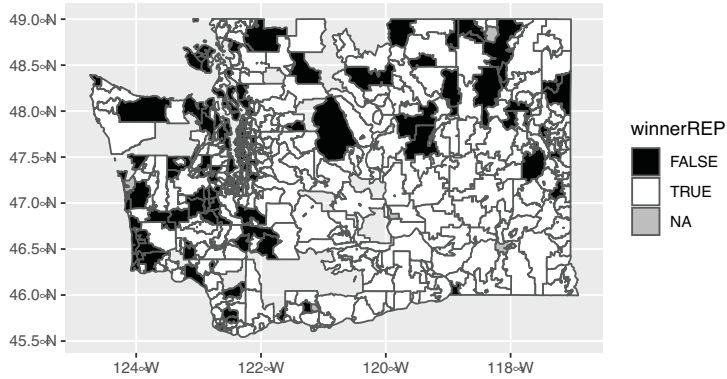


Figure 7.10 Plotting merged map

Notice the presence of missing values and its customization. The original shapefile was downloaded from the Office of Financial Management from Washington State; and the data on contributions was obtained from Washington State portal for open data.

In **Python**, I made sure to put a missing value if missing values were compared; the function `any()`, used in the second condition, help me do that. Now, I am ready to see if my merged map can plot the info that came from the external data frame. Let me prepare the code:

```
> mergedMap = ggplot(data=allZip) +
+             geom_sf(aes(fill=winnerREP)) +
+             scale_fill_manual(values = c('black','white'),
+                               na.value = "grey")
```

You can see the object `mergedMap` in Figure 7.10. Notice that the color for the polygons use `fill` instead of `color` (`color` can be used for the border of the polygon). Also, as I wanted to use the variable `winnerREP` to color the polygons, I put the `fill` argument inside the `aes()`.

Let me use basic *geopandas* plotting capabilities to produce a map similar to Figure 7.10. Notice that I used a particular palette (colormap) to resemble that figure (`gist_gray`).

```
allZip.plot(column='winnerREP', #column to color
            categorical=True,
            edgecolor='black',
            legend=True,
            missing_kwds={'color': 'lightgrey'},
            cmap='gist_gray') # palette for column chosen
```

Notice that the legend for categorical data using **Python** was located on top of the plot. I will show you how to improve that in subsection 7.5.2.

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

7.3.2 Dissolving

Sometimes you want to use your map for more than one purpose. For instance, these many polygons may be aggregated to represent another spatial aspect. In those situations, you may want to *dissolve* the map. Let me create a basemap just with the border of Washington State. Dissolving is not a trivial process, as it will connect shapes. Combining them requires that if two polygons are neighbors, their borders should not have spaces in-between, as any space might be visually interpreted as another polygon. When this happens, you have *sliver polygons*. Shapefiles are not perfect and some operations, like dissolving, can fail if sliver polygons are present. To avoid that situation, you can either simplify the map or build some buffers around the polygon. In **R**, simplifying is fairly easy using *rmapshaper* (Teucher et al., 2020):

```
> library(rmapshaper)
> waBetter=ms_simplify(wazipMap)
```

The object `waBetter` is just a cleaner version of `wazipMap` after using the function `ms_simplify`; it is now time to dissolve using `ms_dissolve`, another function from *rmapshaper*:

```
> waBorder ← ms_dissolve(waBetter)
```

The object `waBorder` can be plotted as before:

```
> borderMap=ggplot(waBorder) +
+              theme_classic() +
+                geom_sf(fill='white',
+                        color='black') # border color
```

You can see the result in Figure 7.11.

Let me get rid of possible sliver polygons by using the alternative strategy mentioned, that is, creating buffers in Python:

```
## a. make copy
waBorder=wazipMap.copy()
## b. Correct map with buffer (may not be needed)
waBorder['geometry'] = waBorder.buffer(0.01)
## c. create a constant column by which to dissolve
waBorder['dummy']=1
## d. dissolving
waBorder= waBorder.dissolve(by='dummy')
## e. plot the dissolved map
waBorder.plot(color='white',edgecolor='black')
```

In **Python**, I also needed to create a new column (a *dummy*) (the function requires one) to be used when dissolving. This process does not need to dissolve all the polygons into one. Notice that **R** dissolves all polygons by

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

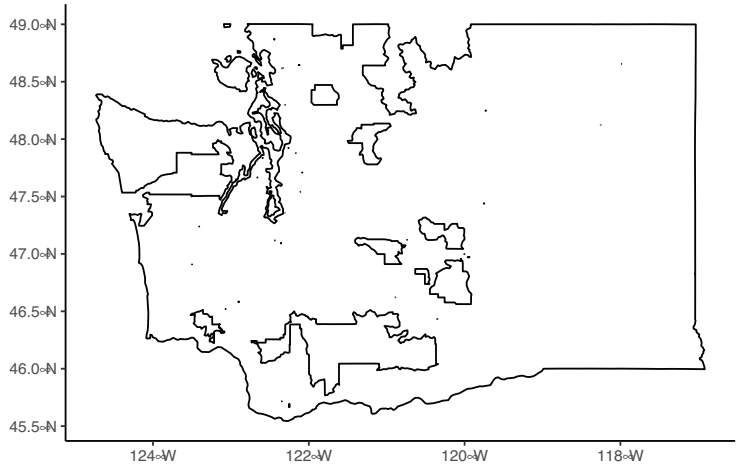


Figure 7.11 The effect of dissolving a map by a constant value

In this case all the internal borders between polygons dissappeared. The original shapefile was downloaded from the Office of Financial Management from Washington State; and the data on contributions were obtained from Washington State portal for open data.

default if no field is present in the function. However, if the "dissolving column" or field had more values, the result would be different. Let me show you that by using my `allZip` map and its `winnerREP` column:

```
> #optimizing
> allZipBetter=ms_simplify(allZip)
> #dissolving
> allZipREP=ms_dissolve(allZipBetter,field = "winnerREP")
> #plotting the dissolved map
> dividedZip= ggplot(data = allZipREP) + theme_classic() +
+             geom_sf(aes(fill=winnerREP))  +
+             scale_fill_manual(values = c('black','white'),
+                               na.value = "grey")
```

As you can see in Figure 7.12, you turn the whole set of polygons into a binary map (notice I show you a way to inform the polygons with missing data).

At this point, remember that simplifying or buffering is not strictly needed, but if your code fails without it, try including those processes. In this case, I needed to make the `allZipREP` object and produce Figure 7.12, whose similar version in **Python** can be achieved like this:

```
## a. make copy
allZipBetter=allZip.copy()
## a.1 saving missing values
```

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

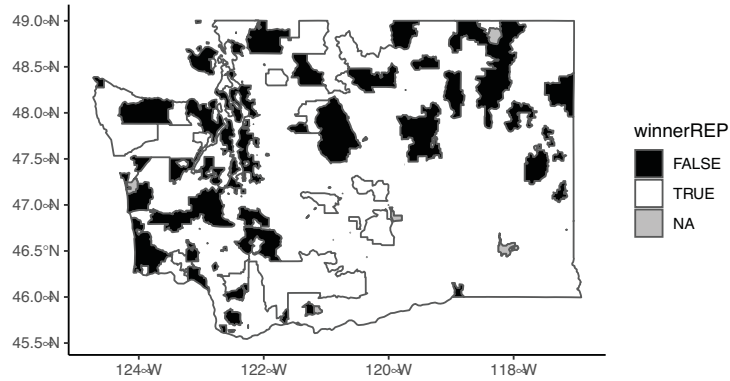


Figure 7.12 The effect of dissolving a map by an aggregating variable
In this case, all the zip codes where one party had more contributions than the other were dissolved. The original shapefile was downloaded from the Office of Financial Management from Washington State; and the data on contributions were obtained from Washington State portal for open data.

```
NAs = allZipBetter[allZipBetter.winnerREP.isna()]
## b. Correct map with buffer (may not be needed)
allZipBetter['geometry'] = allZipBetter.buffer(0.01)
## c. dissolving
allZipREP= allZipBetter.dissolve(by='winnerREP',as_index=False)
## d. plotting the dissolved map
allZipREP.append(NAs).plot(column='winnerREP', #column to color
                edgecolor='black',
                categorical=True,legend=True,
                missing_kwds={'color': 'grey'},
                cmap='gist_gray')
```

Notice that *geopandas* did not create missing values after dissolving. So, I had to save the missing values in the object `NAs` and append it to `allZipREP` before plotting it. It is a functionality not yet present in *geopandas*. However, it is important to think that if you want to have more control over your work, you should use maps with complete data, and take advantage of the *layering* capacities of **R** and **Python** (and every software for maps), which comes next.

7.3.3 Layering

When you have several maps, you can be creative and put each one on top of the other; I like this because it gives me more control over how to plot missing values. Let me create another spatial element, this time using the coordinates I have on the donor location. As you know, the data frame `contriWAsub` has columns with coordinates, which represent a point on a map:

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

```
> contriWAsub[,c(9:10)]%>% head()
```

```
       Lat       Lon
1 47.78220 -122.0626
2 47.42512 -120.3399
3 47.63716 -122.2765
4 47.72272 -122.3513
5 47.52707 -122.0466
6 47.26280 -122.5228
```

Let's use those columns to create a spatial point data frame, while making sure it has the same coordinate system as our map:

```
> WApoints= st_as_sf(contriWAsub,
+                    coords = c("Lon", "Lat"), #in that order
+                    remove = FALSE,
+                    crs = st_crs(allZip)$epsg)
```

The `WApoints` object is now a *spatial points* data frame:

```
> class(WApoints)
```

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

You can achieve the same in **Python** with with this code:

```
WApoints = gpd.GeoDataFrame(contriWASub,
               geometry=gpd.points_from_xy(contriWASub.Lon,
                                           contriWASub.Lat))
WApoints.crs = wazipMap.crs
```

Let me create a map with the polygons that have missing values in the column `total`:

```
> allZipNA=allZip[!complete.cases(allZip$total),]
```

That was easy. It is also easy in Python:

```
allZipNA=allZip[allZip.total.isnull()].copy()
```

Let me use some objects I already have:

- the **border** of Washington State: `waBorder`
- the **zips** with missing information: `allZipNA`
- the **location** of the contributors: `WApoints`.

If we have more than one spatial object, we need layering:

```
> #layers respect the order:
> layerBorder=ggplot(data=waBorder) + theme_void() +
+                                     geom_sf(fill=NA)
> layerMissing=layerBorder + geom_sf(data = allZipNA,
```

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

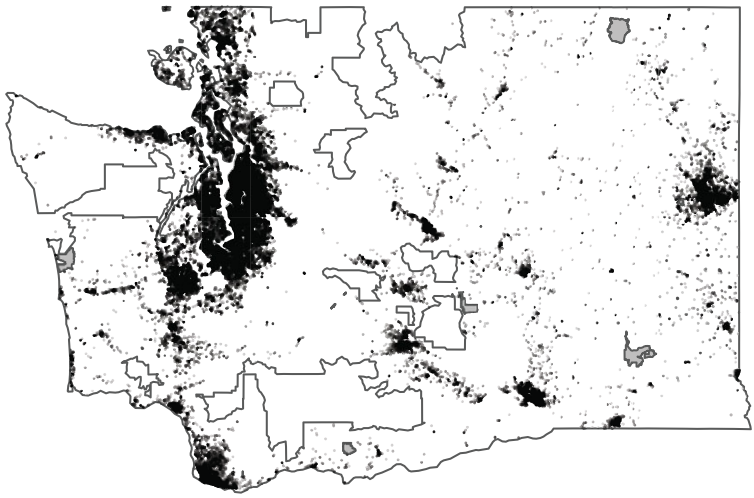


Figure 7.13 Layers

The plot shows a set of points as a layer on top of the border map of Washington State. The original shapefile was downloaded from the Office of Financial Management from Washington State; and the data on contributions were obtained from Washington State portal for open data.

```
+                                      fill='grey')
> layerPoint= layerMissing + geom_sf(data = WApoints,
+                                    size = 0.1,
+                                    color='black',
+                                    alpha=0.1) #transparency
```

As you can see in Figure 7.13, you have a map that combines the information from every layer.

Python with *geopandas* uses the same approach: layering respecting the order of the code:

```
layerBorder= waBorder.plot(edgecolor='grey',color='white')

layerMissing=allZipNA.plot(edgecolor='grey',color='grey',
                           ax=layerBorder)
WApoints.plot(color='black',
              markersize=0.1,alpha=0.1,
              ax=layerBorder) # on top of!)
```

Notice that the argument **ax** is the one which each map uses as the base map. That argument is also used in *geoplot*:

```
layerBorder = gplt.polyplot(waBorder,
                            edgecolor='grey',
                            facecolor='white')
layerMissing = gplt.polyplot(allZipNA,
```

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

```
                            edgecolor='grey',
                            facecolor='grey',
                            ax=layerBorder)
gplt.pointplot(WApoints,
               color='black',
               s=0.1,#size of point
               alpha=0.1,
               ax=layerBorder)# on top of!
```

7.3.4 Layers and Map Reprojection

Keep in mind that the objects `allZipNA` and `waBorder` are "children" of the original map `contriWA`, so all of them share the same projection, and since we used that projection to create the object `WApoints`, our layered map `layerPoint` was perfect. You can get far from perfect results if your layers do not share the same projection (by default the projection assumed is the one from the first or base layer). Let me show you how to set or change a projection on the fly.

In **R**, you only need to add the element `coord_sf()`, indicating the projection needed (*Mercator* (Osborne, 2013) in the code below):

```
> layerPointRP= layerPoint +  coord_sf(crs = "+proj=merc")
```

If you know the EPSG, you can alternatively write:

```
> layerPointRP= layerPoint +  coord_sf(crs = st_crs(3857))
```

You can see the result of this reprojection in Figure 7.14.

Geopandas does not offer a simpler alternative, you just need to reproject every map:

```
layerBorder= waBorder.to_crs("EPSG:3395").plot(edgecolor='grey',
                                               color='white')

layerMissing=allZipNA.to_crs("EPSG:3395").plot(edgecolor='grey',
                                               color='grey',
                                               ax=layerBorder)

WApoints.to_crs("EPSG:3395").plot(color='black',
                                  markersize=0.1,
                                  alpha=0.1,
                                  ax=layerBorder)
```

Geoplot has a simpler way of reprojecting, as it only requires you to reproject the base layer. See how I do that next, and notice I need the activation of `geoplot.crs` first:

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

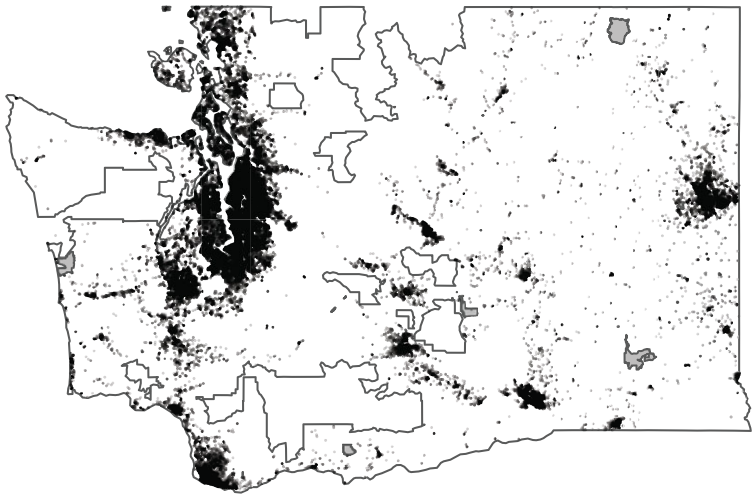


Figure 7.14 Layers reprojected

This is a reprojected representation of Figure 7.13 using the *Mercator* projection. The original shapefile was downloaded from the Office of Financial Management from Washington State; and the data on contributions were obtained from Washington State portal for open data.

```
import geoplot.crs as gcrs #activating!

layerBorder = gplt.polyplot(waBorder,
                            projection=gcrs.Mercator(),#HERE!
                            edgecolor='grey',
                            facecolor='white')

layerMissing = gplt.polyplot(allZipNA,
                             edgecolor='grey',
                             facecolor='grey',
                             ax=layerBorder)
layerPoint= gplt.pointplot(WApoints,
                           color='black',
                           s=0.1,
                           alpha=0.1,
                           ax=layerBorder)# on top of!
```

Notice that the reprojected map got rid of the axis values, so it will be a harder job for the basic user to zoom the reprojected map. Let me talk about this next.

7.3.5 Fixing Projection in Layers

Let me use another map. I have this map of the county borders from Washington State from the *Washington Department of Natural Resource GIS Open Data*. After I downloaded the shapefile from that address, I transformed

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

```
Simple feature collection with 39 features and 1 field
geometry type:  POLYGON
dimension:      XY
bbox:           xmin: -13899440 ymin: 5707530 xmax: -13014940 ymax: 6275274
epsg (SRID):    4326
proj4string:    +proj=longlat +datum=WGS84 +no_defs
# A tibble: 39 x 2
   JURISDIC_2                                                    geometry
   <chr>                                                      <POLYGON [°]>
 1 Grant      ((-13244144 6097477, -13244380 6097576, -13244493 6097651, -132446…
 2 Lincoln    ((-13173695 6089301, -13173818 6089081, -13173992 6088770, -131741…
 3 Whitman    ((-13028820 5984497, -13029142 5984496, -13029471 5984515, -130308…
```

Figure 7.15 Problem with projected geojson

Original shapefile from the Washington Department of Natural Resource GIS Open Data.

those files into a *geojson* map using *mapshaper* as I had done before. Let me open that version currently in the *GitHub* from this book:

```
> myGit="https://github.com/resourcesbookvisual/data/"
> myGeo2="raw/master/WA_County_Boundaries.json"
> mapLink2=paste0(myGit,myGeo2)
> waCounties=read_sf(mapLink2)
> waCounties[,c("JURISDIC_2")]
```

The previous command should have given you the information shown in Figure 7.15, which tells you the *espg* is the same as we have been using for our previous maps. However, you can clearly see a big problem: the values in the geometries are not right. The previous map used a projection with latitude and longitude coordinates; and, you may know, latitude can take values between 0 and 90 (positive or negative), while longitude can take values between 0 and 180 (positive or negative). As the metadata of the file are corrupt, this will not allow you to use this map with the other ones, even though they are illustrating the same geography.

There may be several sources for this problem in Figure 7.15, but the fact is that the *geojson* file I created from the original shapefile using *mapshaper* has no information on what projection to use, as you can see in Figure 7.16.

When no projection is found, **R** and **Python** will assign the EPSG **4326**. I need to tell **R** the right projection, then I advise a couple of things: You can try a program like *QGIS* (QGIS.org, 2020), which may recognize the geometry and choose the right projection; alternatively, you can check the original website and see some extra information, in my case I checked *metadata* and *source*. I did both things, and found out the EPSG was **3857**, so I reloaded the file using that value:

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

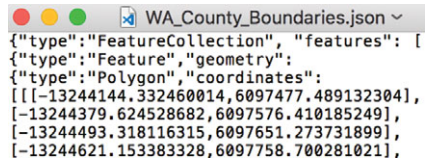


Figure 7.16 Contents of geojson file without projection information

Using a simple text editor I opened and found the contents shown, where I confirm no information on projection is given. The original shapefile was downloaded from the Washington Department of Natural Resource GIS Open Data and converted to *geojson* using mapshaper website.

```
> waCounties=read_sf(mapLink2,crs=3857)
```

The previous code worked very well; I got the data in the right projection (pseudo mercator). However, I need to transform this map of counties into the right projection. Let me do that next:

```
> waCounties=st_transform(waCounties, crs=4326)
```

Python will have the same problem, so I will save the reprojected object in **R**, and I will call this file later in **Python** (I will upload a file named `waCountiesfromR` into the book's *GitHub* and read it from there in Python):

```
> # extension "geojson" is used instead of "json"
> # so the driver for the conversion can be easily assigned.
> st_write(waCounties, "waCountiesfromR.geojson")
```

You may have chosen to keep looking for a map with the right projection, or you can even alter the original geojson file by adding whatever was missing. You can see the reprojected contents of the geojson file in Figure 7.17.

7.4 Focusing on Map Zones

One important operation in any mapping project is zooming. I have some previous maps but now I have the need to focus in a particular area. My intention now is to focus on King County. Since I have the right map, I will keep that county:

```
> kingMap=waCounties[waCounties$JURISDIC_2=="King",]
```

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

waCountiesfromR.geojson


```
{
"type": "FeatureCollection",
"crs": { "type": "name", "properties": { "name": "urn:ogc:def:crs:OGC:1.3:CRS84" } },
"features": [
{ "type": "Feature", "properties": { "OBJECTID": 2480, "JURISDICT_": 25, "JURISDIC_1":
4, "JURISDIC_2": "Grant", "JURISDIC_3": "Grant County", "JURISDIC_4": 13, "JURISDIC_5":
53025, "JURISDIC_6": "", "EDIT_DATE": "2018\/03\/14 00:00:00+00", "EDIT_STATU": 1,
"EDIT_WHO": "TSTE490" }, "geometry": { "type": "Polygon", "coordinates":
[ [ [ -118.974172789337629, 47.943602893987752 ], [ -118.97628645395281,
47.944198145095157 ], [ -118.977307780827559, 47.944648627140772 ],
[ -118.978456144569648, 47.945295046129559 ], [ -118.979268314852789,
47.945855190052626 ], [ -118.980329709493873, 47.94675986658887 ],
[ -118.980441581886723, 47.946842866719095 ], [ -118.980515602977263,
47.946897975348271 ], [ -118.9816420972559, 47.947736641907568 ],
```


Figure 7.17 Contents of geojson file with projection information

Using a simple text editor I opened and found the contents shown, where I confirm the information on projection is given. The original shapefile was downloaded from the Washington Department of Natural Resource GIS Open Data and converted to *geojson* using mapshaper website, the map was then reprojected in **R**.

Remember I did not solve the reprojection in **Python**, so I will use **Python** to call the improved map file I saved in **R**, and keep King County:

```
myGit="https://github.com/resourcesbookvisual/data/"
myGeo2="raw/master/waCountiesfromR.geojson"
mapLink2=myGit + myGeo2

waCounties= gpd.read_file(mapLink2)

kingMap=waCounties[waCounties.JURISDIC_2=="King"]
```

The easiest way to zoom in is to know the limits of the area of interest, also known as the *bounding box*. You can get the bounding box of our King County map with this code:

```
> st_bbox(kingMap)
```

```
      xmin       ymin       xmax       ymax
-122.54166   47.08435 -121.06595   47.78058
```

The values above give you coordinates of the diagonal of the box, I will now use those values to zoom in a particular zone of my map in Figure 7.14. Notice that I need to select the right the zoom area using the right indices:

```
> # zooming area:
> forX=st_bbox(kingMap)[c(1,3)] # recovering X range
> forY=st_bbox(kingMap)[c(2,4)] # recovering Y range
> # maps of WASHINGTON
> Border=ggplot(data=waBorder) + theme_classic() +
+                                  geom_sf(fill='white')
> Zips=Border + geom_sf(data = wazipMap,fill='grey90')
> Points= Zips + geom_sf(data = WApoints,size = 0.1,color='black',
+                        alpha=0.1) #transparency
> # ZOOMING IN:
> zoomedMap=Points +  coord_sf(xlim=forX,ylim = forY)
```

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

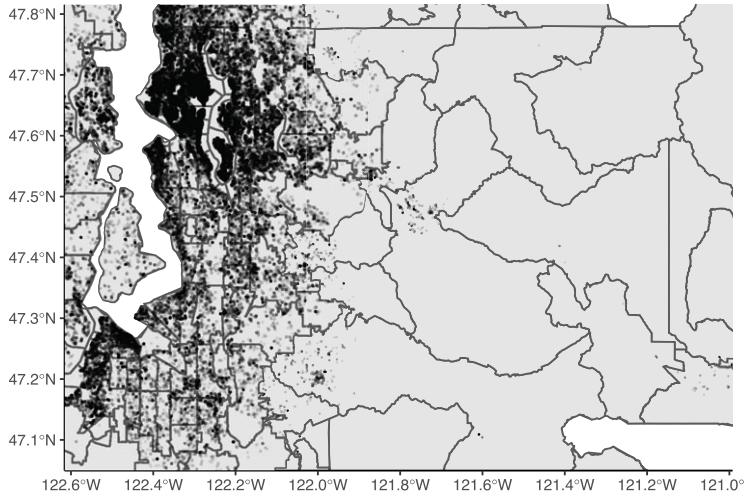


Figure 7.18 Zooming in

The bounding box of one polygon (King County) was used to set the zoom box. The original map of zip codes from Washington State was downloaded from the Office of Financial Management from Washington State, the original map of counties was downloaded from the Washington Department of Natural Resource GIS Open Data and converted to *geojson* using mapshaper website, the map was then reprojected in **R**.

The object `zoomedMap` has zoomed in the information of three map layers, the result is shown in Figure 7.18.

Python needs to use *matplotlib* to produce a zoomed map in an easy way. The alternative in *geopandas* is shown next. Notice I also show you, as a comment, how to alter the size. An important piece of code is the use of `total_bounds` to get the bounding box.

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()#plt.subplots(figsize=(10,6))
# zooming area:
ax.set_xlim(kingMap.total_bounds[0:4:2]) #recovering indices
ax.set_ylim(kingMap.total_bounds[1:5:2]) #recovering indices
# maps of WASHINGTON
Border= waBorder.plot(edgecolor='grey',color='white',ax=ax)
Zips=wazipMap.plot(edgecolor='silver',color='whitesmoke',ax=Border)
# ZOOMING IN:
WApoints.plot(color='black',markersize=0.1,alpha=0.1,ax=Border)
plt.show()
```

You are now aware of certain spatial operations that help you merge data and maps, and combine maps by layering. These are operations that novice user of spatial information will find easy. Going deeper into these matters will

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

require more formal training on these subjects. Let me now start using the numerical or categorical information in our maps.

7.5 Customizing Color and Size

Let me use the map `allZip`, which already has information from Washington State contributions for the year 2012. I will make a couple of maps using the money contributed. Let me remind you that the map `allZip` has some missing values in columns describing contributions:

```
> sum(is.na(allZip$total))
```

```
[1] 15
```

All those polygons made up the map `allZipNA` which was previously created. Let me subset `allZip` without missing data:

```
> allZip=allZip[complete.cases(allZip$total),]
```

The **Python** version is very similar:

```
allZip=allZip[~allZip.total.isnull()]
```

Let me prepare a couple of maps:

- The first map will represent the amount of money given to Democrat campaigns (see Figure 7.19 panel a) by color intensity:

```
> base =ggplot(data=waBorder) + geom_sf(fill='red') + theme_void()
> forDems =base + geom_sf(data=allZip,aes(fill=DEMOCRAT),color=NA)
> forDems =forDems + guides(fill=FALSE) # no legend
```

The **Python** code in *geopandas* looks like this:

```
Border= waBorder.plot(edgecolor='grey',color='red')
allZip.plot(column='DEMOCRAT',ax=Border)
```

The **Python** code using *geoplot* looks like this:

```
Border = gplt.polyplot(waBorder,
                       edgecolor='grey',
                       facecolor='red')
gplt.choropleth(allZip, hue='DEMOCRAT',ax=Border)
```

- The second map will represent the amount of money given to campaigns in general (see Figure 7.19 panel b) by color intensity:

```
> forALL = base + geom_sf(data=allZip,aes(fill=total),color=NA) +
+                 guides(fill=FALSE)
```

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

The *geopandas* version can be:

```
Border= waBorder.plot(edgecolor='grey',color='red')
allZip.plot(column='total',ax=Border)
```

A *geoplot* version follows:

```
Border = gplt.polyplot(waBorder,
                       edgecolor='grey',
                       facecolor='red')
gplt.choropleth(allZip, hue='total',ax=Border)
```

Figure 7.19 shows you the result for both of the previous objects.

Notice a couple of things when comparing Figure 7.19 panel a and 7.19 panel b. First, I have used the border map in red, so that you can identify the polygons with missing values; second, and most important, both maps are not very different. They show you "absolute" amounts that are clearly influenced by the size of the population. That is why the coloring of maps requires some thinking in order to produce a **choropleth** map, which represents relative values instead. Let me produce a map of the contributions to Democrats relative to the the total contribution in the zip code (see Figure 7.20):

```
> #new variable:
> allZip$DemChoro=allZip$DEMOCRAT/allZip$total
> #plotting new variable
> choro1 = base + geom_sf(data=allZip,aes(fill = DemChoro),color=NA)
```

The object `choro1` is shown in Figure 7.20. Notice I keep using the same base map, but the coloring is very different.

The code for the object `choro1`in *geopandas* should look familiar:

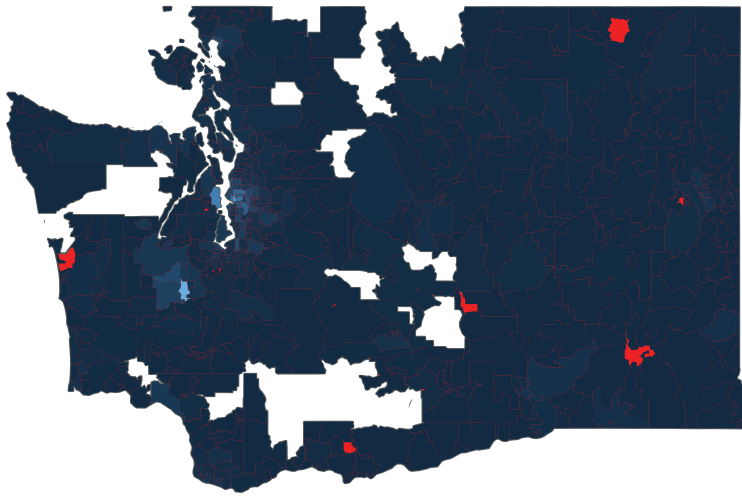
```
Border= waBorder.plot(edgecolor='grey',color='red')
allZip['DemChoro'] = allZip.DEMOCRAT / allZip.total
allZip.plot(column='DemChoro',legend=True,ax=Border,
            legend_kwds={'shrink': 0.6}) #shrink legend size
```

Notice that the legend for the continuous variable was located off the map. However, the size of the legend may be a little bigger, so I showed you how to resize it using the argument `'shrink'` inside the dictionary `legend_kwds`, which allows further settings for the legend. If you prefer to use *geoplot* in **Python**, you can do this (I will leave the default legend size):

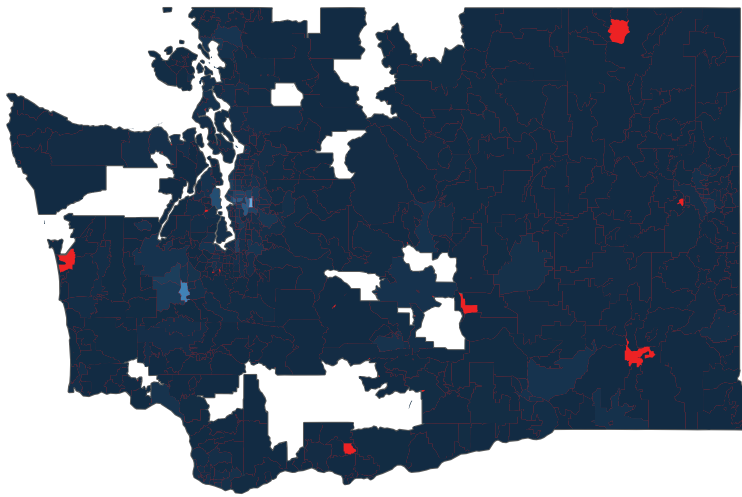
```
Border = gplt.polyplot(waBorder,
                       edgecolor='grey',
                       facecolor='red')
gplt.choropleth(allZip, hue='DemChoro',ax=Border, legend=True)
```

Notice that the process to represent missing values in **Python** was easier than when we tried this on page 207. Also, you should realize by now that while **R** can keep using the `Border` map, once *geopandas* or *geoplot* have used it you need to call it again.

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press



(a) Coloring contributions to democrats



(b) Coloring total contributions

Figure 7.19 Coloring polygons with absolute values

The total amount contributed in the zip code. Polygons with missing information are colored in *red*. The original shapefile was downloaded from the Office of Financial Management from Washington State; and the data on contributions were obtained from Washington State portal for open data

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

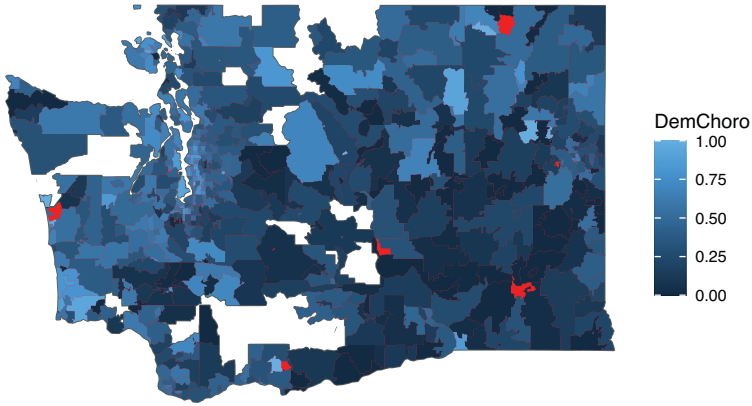


Figure 7.20 The choropleth map

Coloring polygons with relative values: the total amount contributed to Democrats by the total amount contributed in the zip code. Polygons with missing information are colored in *red*. The original shapefile was downloaded from the Office of Financial Management from Washington State; and the data on contributions were obtained from Washington State portal for open data.

7.5.1 Discretizing Values for Colors

In Figure 7.20, we used relative values to plot, but still you might be having a hard time displaying the information because you have so many polygons, with different sizes. This situation is similar to depicting five hundred bars in a plot (maybe harder). Let me take you to a simpler set of polygons at the county level. I am going to add some information to the current county map I have on COVID from wikipedia (Wikipedia, 2020), which I have saved in a csv file for this example (as the data structure could vary later).

```
> link3='raw/master/covidCountyWA.csv'
> LINK=paste0(link1,link3)
> #getting the data TABLE from the file in the cloud:
> covid=read.csv(file=url(LINK),stringsAsFactors = F)
> #first rows:
> head(covid)
```

```
   County Cases Deaths Recov Population CasesPer100k
1   Adams   134      0    45      19983        650.6
2  Asotin    21      2    NA      22582         93.0
3  Benton  1731     82    NA     204390        824.9
4  Chelan   330      6    NA      77200        408.0
5 Clallam    36      0    19      77331         46.6
6   Clark   731     30    NA     488241        142.8
```

As you see, the column candidate to be colored in the next choropleth is `CasesPer100k`. First let me do the merge:

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

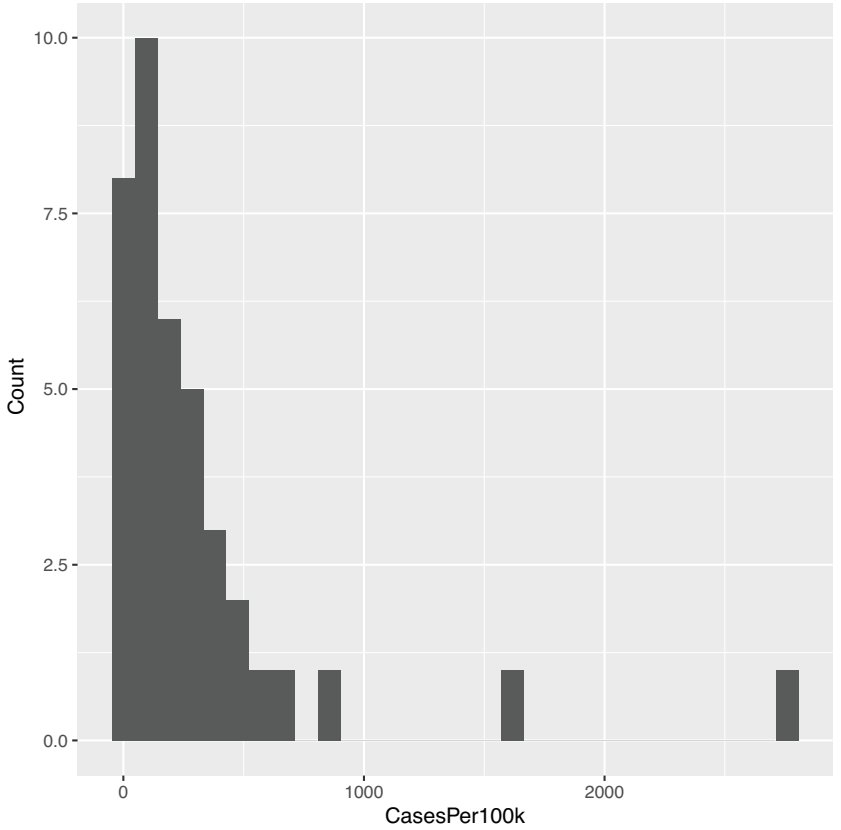


Figure 7.21 Exploring the distribution of cases per one hundred thousand people
This is needed to decide the way intervals will be created. The original data comes from (Wikipedia, 2020).

```
> covidMap=merge(waCounties[,c("JURISDIC_2")],covid,
+                by.x="JURISDIC_2",by.y="County")
```

When preparing Figure 7.20, I did not customize the colors or the amount of intervals. The first step in this process is to explore the variable to plot. With this simple code, I can produce a histogram.

```
> explore1=ggplot(covidMap,aes(CasesPer100k)) + geom_histogram()
```

The object `explore1` can be seen in Figure 7.21. Notice that it is positively skewed.

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

When the variable is shaped as a bell, you can follow traditional ways of creating intervals. When this does not work, you might need some modern approaches. **R** has a great library, *classInt* (Bivand et al., 2020), that will easily provide you with intervals. The library in **Python** for this same purpose is *mapclassify* (Rey et al., 2020). The question of how many intervals are needed is difficult to answer, as it depends on how the data are distributed (presence of skewness and data outliers), and also because you have a limit of intervals, considering that color palettes, like the one proposed in (Brewer, 1999), do not offer palettes with more than twelve color combinations. Apparently simple issues like these have been debated for a while (see (Brewer and Pickle, 2002)).

I always tend to use an odd number of intervals, and my usual choice is five colors, and if possible three. As you see, I prefer that readers do not get confused; at the end of the day, you need your audience to see the difference between the good, the bad, and the middle counties affected by a number of cases. Let me work in this section with four styles, trying to get five intervals:

- Equal intervals: This is a simple algorithm. You get intervals with the same width. This might be the easiest to understand.
- Quantiles: This might be a better option if your audience understands quantile statistics. The interval widths vary because the algorithm tries to put the same amount of cases into each interval.
- Jenks: This follows an optimization algorithm (see (Jenks and Caspall, 1971), or (Fisher, 1958)); that is, it tries to create an interval where it considers there is more density of cases, so it looks for an intrinsic grouping.
- Head-Tails: This is also an optimization algorithm, but it is more suitable for distributions with long tails (Jiang, 2013). In this style, you do not have to request the amount of intervals, the algorithm decides that.

Let me show you the steps for the *equal* style in **R**, which will be similar for the others:

1. Select variable:

```
> varToPlot=covidMap$CasesPer100k
```

2. Activate the library, and get the break points using the selected *style*. Notice that I need to request **brks** at the end here:

```
> library(classInt)
> cutEqual=classIntervals(varToPlot,n = 5,style = "equal")$brks
```

3. Create a categorical variable from numeric variable. The previous step will allow you to get the break points, which will be used to cut the numeric

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

variable. Notice I have selected, obviously for this case, that the output is an ordinal variable:

```
> covidMap$cases_Equal=cut(varToPlot, breaks = cutEqual,
+                          dig.lab=5, # digits to use in legend
+                          include.lowest = T,ordered_result = T)
```

4. By default, the function `cut` will write the values in each interval separated by a comma, here I change it to a dash and spaces:

```
> levels(covidMap$cases_Equal)=gsub(",","," - ", #substituting
+                                   levels(covidMap$cases_Equal))
```

5. The ordinal palettes can be divergent or sequential, as discussed in Section 3.1.3 (read from page 36). Let me choose a sequential one:

```
> colorPal="OrRd"
```

6. Prepare the object to plot:

```
> base= ggplot(covidMap) + theme_light()
> choro2=base + geom_sf(aes(fill=cases_Equal))
> choro2=choro2+scale_fill_brewer(palette = colorPal,direction=1)
```

The variable `rate_Equal` is ready. You will soon realize that, as it can happen in the *equal* style, one interval has no cases. Now let me follow the same steps to get the intervals using the quantile style:

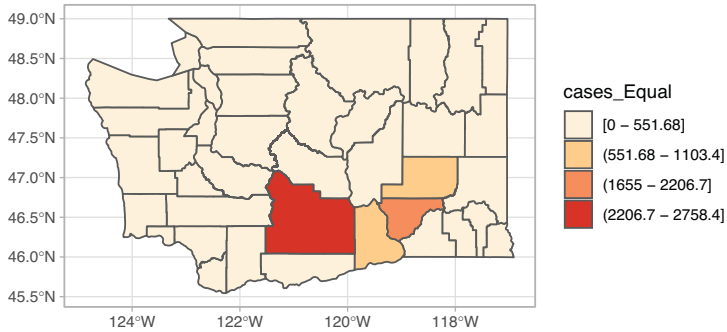
```
> # get intervals
> cutQuant=classIntervals(varToPlot,n = 5,style = "quantile")$brks
> #create variable
> covidMap$cases_Quant=cut(varToPlot, cutQuant,dig.lab=5,
+                          include.lowest = T,ordered_result = T)
> #change separator
> levels(covidMap$cases_Quant)=gsub(",","," - ",
+                                   levels(covidMap$cases_Quant))
> #prepare object
> base= ggplot(covidMap) + theme_light()
> choro3=base + geom_sf(aes(fill=cases_Quant))
> choro3=choro3+scale_fill_brewer(palette = colorPal,direction=1)
```

You can see the objects `choro2` and `choro3` in Figure 7.22.

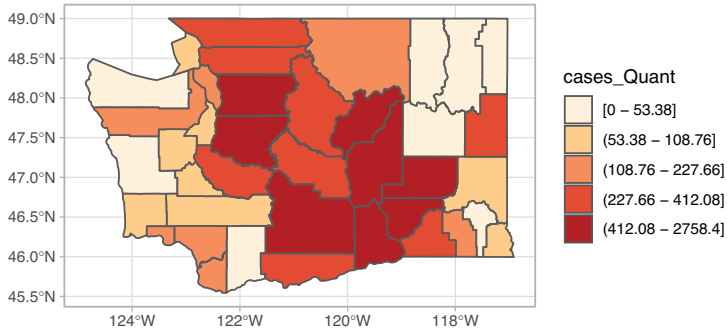
In **Python**, *geopandas* is connected to the `mapclassify` library, so once it is loaded, the steps are simple, as you just need to fill in the parameters requested by the function, without the need to create new columns as I did in **R**.[7]

[7] Notice that the legend requested appears on top of the plot, I will show you how to improve that in Subsection 7.5.2.

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press



(a) Coloring by *Equal* style



(b) Coloring by *Quantile* style

Figure 7.22 Coloring polygons with traditional styles

The variable used has been rate of Covid positives per 100,000 habitants. Five intervals were requested, but the **equal** option produced only four, because one interval had zero cases. The palette used is sequential and named as *OrRd*. The palette has been recommended (Brewer, 1999); the map was downloaded from the Washington Department of Natural Resource GIS Open Data, and includes the data on COVID from (Wikipedia, 2020).

```
import mapclassify as mc

#well-known styles
#geopandas for Equal intervals
covidMap.plot(column='CasesPer100k',
              scheme='EqualInterval',
              k=5,
              cmap='OrRd',
              legend=True)
#%%
#geopandas for Quantiles
covidMap.plot(column='CasesPer100k',
              scheme='Quantiles',
              k=5,
              cmap='OrRd',
              legend=True)
```

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

You can also use *geoplot*, which also reads the `mapclassify` output, but it does require you to call it explicitly (like using `mc.EqualInterval` where you also indicate how many cuts are needed).

```
#well-known styles
#geoplot for Equal intervals
gplt.choropleth(covidMap,
                hue='CasesPer100k',
                scheme=mc.EqualInterval(covidMap.CasesPer100k,k=5),
                cmap='OrRd',
                legend=True)
#%%
#geoplot for Quantiles
gplt.choropleth(covidMap,
                hue='CasesPer100k',
                scheme=mc.Quantiles(covidMap.CasesPer100k, k=5),
                cmap='OrRd',
                legend=True)
```

Remember that once you know which color palette is the right one, you can reverse the color if needed. In **R**, you need to change from `1` to `-1` in the `direction` argument of `scale_fill_brewer`; while in **Python**, you just append `_r`, for example, the palettes previously used can be reversed if you just write `OrRd_r` instead.

Let me prepare the plots with the intervals created using the optimization alternatives. Let me create the variables first:

```
> # get intervals
> cutJenks=classIntervals(varToPlot,n = 5,style = "jenks")$brks
> cutHT=classIntervals(varToPlot,style = "headtails")$brks
> #create variable
> covidMap$cases_Jenks=cut(varToPlot, cutJenks,dig.lab=5,
+                          include.lowest = T,ordered_result = T)
> covidMap$cases_HT=cut(varToPlot, cutHT,dig.lab=5,
+                          include.lowest = T,ordered_result = T)
> #change separator of intervals (better legend)
> levels(covidMap$cases_Jenks)=gsub(",","," - ",
+                                   levels(covidMap$cases_Jenks))
> levels(covidMap$cases_HT)=gsub(",","," - ",
+                                levels(covidMap$cases_HT))
```

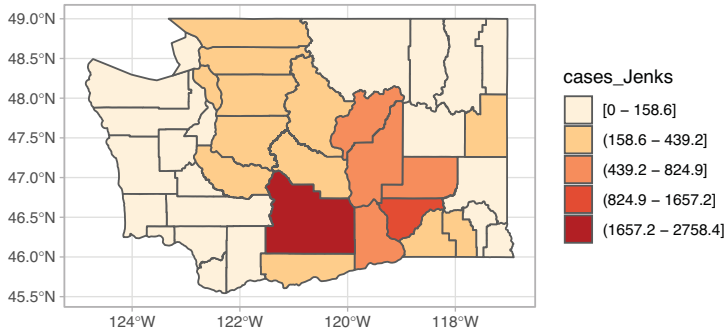
Now let me plot both variables (see the result in Figure 7.23):

```
> #prepare object to plot
> base= ggplot(covidMap) + theme_light()
> #for Jenks
> choro4=base + geom_sf(aes(fill=cases_Jenks))
> choro4=choro4+scale_fill_brewer(palette = colorPal,direction=1)
> #for Head-Tails
> choro5=base + geom_sf(aes(fill=cases_HT))
> choro5=choro5+scale_fill_brewer(palette = colorPal,direction=1)
```

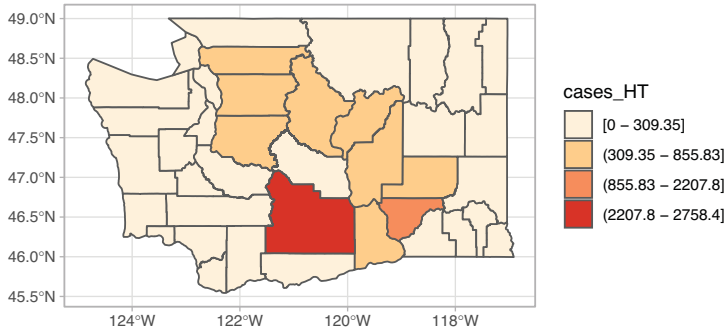
The **Python** alternative follows in *geopandas*, simply using *mapclassify* as before:

```
#optimization styles in geopandas
covidMap.plot(column='CasesPer100k',
```

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press



(a) Coloring by *Jenks* style



(b) Coloring by *Head-Tail* style

Figure 7.23 Coloring polygons with optimization styles

The variable used has been rate of Covid positives per 100,000 habitants. The palette used is sequential and named as *OrRd*. The palette has been recommended (Brewer, 1999); the map was downloaded from the Washington Department of Natural Resource GIS Open Data, and includes the data on COVID from (Wikipedia, 2020).

```
                scheme='FisherJenks',
                k=5,
                cmap='OrRd',
                legend=True)
#%%
covidMap.plot(column='CasesPer100k',
                scheme='HeadTailBreaks',
                cmap='OrRd',
                legend=True)
```

Similarly, this is the alternative version using *geoplot*:

```
#optimization styles in geoplot
gplt.choropleth(covidMap,
                  hue='CasesPer100k',
                  scheme=mc.FisherJenks(covidMap.CasesPer100k, k=5),
```

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

```
                cmap='OrRd',
                legend=True)
#%%
gplt.choropleth(covidMap,
                hue='CasesPer100k',
                scheme=mc.HeadTailBreaks(covidMap.CasesPer100k),
                cmap='OrRd',
                legend=True)
```

So far, I have used one variable. The next logical step would be a two-variable approach.

7.5.2 Color and Area

Color and area are not the best way to encode information. However, maps are a familiar visual whose readers are expect to see color, so the material covered in this section should be useful. Size is not a familiar attribute people expect, beyond experts familiar with this.

A first condition for resizing the polygon that might not seem obvious, is making sure you have a projected map. If your map is in latitude/longitude projection, it is actually unprojected. It is your task to look for a suitable projection to the map you have. In this case, I will turn my lon-lat into mercator again:

```
> library(sf)
> covidMap_reprj=st_transform(covidMap,3857)#mercator
```

Let me create a variable based on deaths and population:

```
> newDeathVals=100000*(covidMap_reprj$Deaths/covidMap_reprj$Population)
> covidMap_reprj$DeathsPer100k=newDeathVals
> varToCut=covidMap_reprj$DeathsPer100k
> cutEqualDeath=classIntervals(varToCut,
+                              n = 3,
+                              style = "equal")$brks
> #create variable
> covidMap_reprj$death_equal=cut(varToCut,
+                                cutEqualDeath,dig.lab=5,
+                                include.lowest = T,
+                                ordered_result = T)
> #change separator
> levels(covidMap_reprj$death_equal)=gsub(",","," - ",
+                                      levels(covidMap_reprj$death_equal))
```

In **Python**, I will simply create the new variable.

```
#new variable
valuesNew=100000*(covidMap.Deaths/covidMap.Population)
covidMap['DeathsPer100k']=valuesNew
```

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

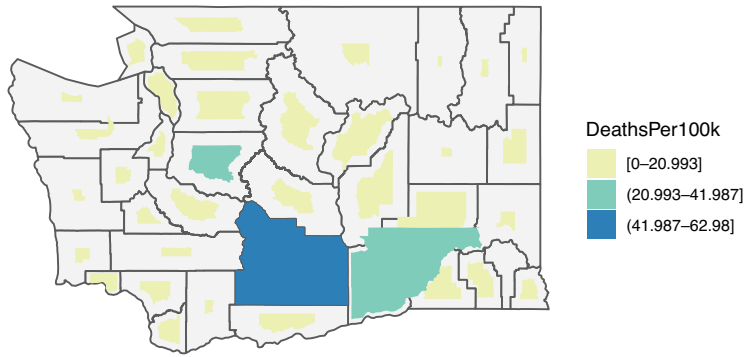


Figure 7.24 Changing polygon sizes via cartogram

The size of the polygon depends on the rate of cases per 100,000 habitants, and the color represents the rate of deaths per one-hundred thousand habitants. The original data comes from (Wikipedia, 2020).

Then, I will first use the variable `CasesPer100k` to alter the area of the polygon, while ensuring that the biggest value in the data should keep the same size with `k=1`. This new visual will be known as *cartogram*:

```
> library(cartogram)
> cartoCases=cartogram_ncont(covidMap_reprj,
+                            weight = "CasesPer100k",#var for resize
+                            k=1) #expansion limit
```

Let me use the object `covidMap_reprj` to plot my two variables, `DeathsPer100k` and `CasesPer100k`, represented by color and size, respectively:

```
> border=ggplot(covidMap_reprj) + geom_sf(fill="grey95") +
+         theme_void()
> cartogram=border+ geom_sf(data=cartoCases, #size
+                           aes(fill=death_equal),
+                           color=NA) #no border
> cartogram=cartogram+scale_fill_brewer(palette='YlGnBu',
+                                       direction=1,
+                                       name="DeathsPer100k")
```

The object `cartogram` will be shown in Figure 7.24.

In **Python**, *geoplot* can produce cartograms while reprojecting at the same time:

```
import geoplot.crs as gcrs #activating!

#border
border=gplt.polyplot(df=covidMap,
                     projection=gcrs.Mercator(),#reproject
```

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

```
                    edgecolor='gray', #border
                    facecolor='gainsboro') #fill of polygon
#area and color
gplt.cartogram(df=covidMap,#map
               scheme=mc.EqualInterval(covidMap.DeathsPer100k,k=3),
               cmap=plt.get_cmap('YlGnBu',3),#palette
               hue="DeathsPer100k", #var for color
               scale='CasesPer100k',#var for resize
               limits=(0.3, 1),  #limits cartogram polygons
               edgecolor='None', #no border
               legend=True,
               legend_var='hue', #legend of what
               legend_kwargs={'bbox_to_anchor': (0.1, 0.4),#location
                              'frameon': True, #with frame?
                              'markeredgecolor':'k',
                              'title':"DeathsPer100k"},
               ax=border)
```

The code in **Python** plots the cartogram on top of the basic map (unaltered). I am using more arguments for the legend this time so it looks better than in previous plots. When I requested legend for the plots with intervals, legends were drawn on the plot, covering some areas of the map. *Geoplot* has the argument `legend_kwargs` that allows you to move the legend around. For that, it uses a dictionary to set up several values; where the most important for this case is `bbox_to_anchor`. If you want the legend at the bottom to the left you use the tuple (0,0), and if you want the legend at the top to the left you use the tuple (1,1); of course, you can vary those values to move it around.

If you want to combine area and color, you do not need to use a cartogram but a simple option such as a symbol, like a dot, to represent size and color. I do not have a spatial point for my current map of counties, but I will get a point for each by calculating the *centroid* in each polygon. In **R**, you use `st_centroid()` to get the centroids. See the code:

```
> base=ggplot(covidMap_reprj) +
+      geom_sf(color='grey50') + theme_void()
> pointSize=base + geom_sf(data=st_centroid(covidMap_reprj),
+                          aes(fill=death_equal, #color
+                              size=CasesPer100k), #size
+                          pch=21) #shape of dot
> pointSize=pointSize+scale_color_brewer(palette='YlGnBu',
+                                        direction=1,
+                                        name="DeathsPer100k")
```

Figure 7.25 might be an easier visual to digest than a cartogram.

I can get a similar result in **Python**, but *geoplot* does not offer an option to plot two legends; you have to choose what the only legend possible represents using the argument `legend_var`. *Geoplot* needs two steps to plot the centroids. First, you need to create a column with the centroids, and then you set that column as the geometry of the map:

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

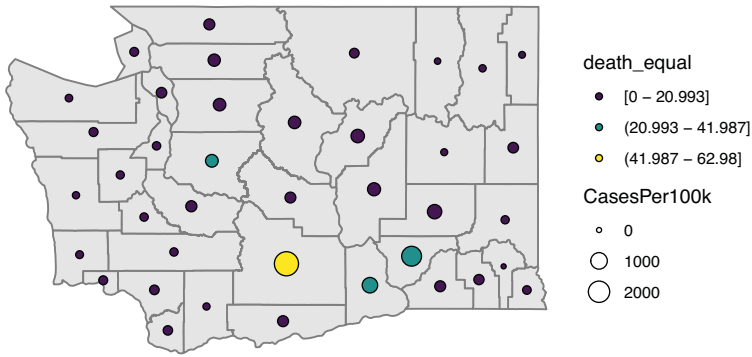


Figure 7.25 Changing polygon sizes via cartogram

Exploring the distribution on cases per one hundred thousand people. The original data comes from (Wikipedia, 2020).

```
import geoplot.crs as gcrs #activating!

#borders
countyBorders = gplt.polyplot(df=covidMap,
                              projection=gcrs.Mercator(),#HERE!
                              edgecolor='grey',
                              facecolor='gainsboro')

#points scaled
covidMap['centroid']=covidMap.centroid #compute centroids
gplt.pointplot(df=covidMap.set_geometry('centroid'), #set geometry
               scheme=mc.EqualInterval(covidMap.DeathsPer100k,k=3),
               cmap='YlGnBu',#palette
               hue="DeathsPer100k",
               scale='CasesPer100k', #sizes of points
               limits=(4, 40), #range for sizes of points
               legend=True,
               legend_var='hue',
               legend_kwargs={'bbox_to_anchor': (1, 1),
                              'frameon': True,
                              'markeredgecolor':'k',
                              'title':"DeathsPer100k"},
               extent = covidMap.total_bounds,
               ax=countyBorders)
plt.show()
```

Notice I have added the argument `extent`. This is important for this particular case to make sure both maps fit well together by setting the extent of the map with the value `total_bounds`, an attribute of `covidMap`.

You can also use color and shapes to identify where a set of points are denser than in other places, which also receives the name of *heatmap*; we have previously created those using frequency tables. Let me use the spatial points I have for the people contributing to campaigns to show you this. A simple and direct version can be:

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

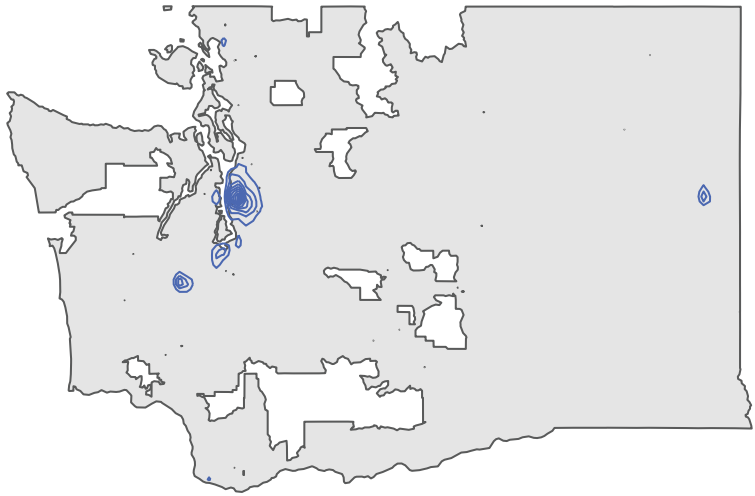


Figure 7.26 Basic spatial heatmap
Exploring the distribution of contributions to political campaign of the year 2012. The data on contributions was obtained from Washington State portal for open data.

```
> border=ggplot(data = waBorder) + geom_sf() + theme_void()
> heat=border+geom_density_2d(data = WApoints,
+                             aes(x=Lon,y=Lat))
```

The object `heat` is represented in Figure 7.26.

The code for **Python** to obtain a similar result may look like this:

```
counties=gplt.polyplot(waCounties,edgecolor= 'silver')

gplt.kdeplot(WApoints,
             shade=False,
             shade_lowest=False,
             ax=counties)
```

Let me try several things using my points on contributors. First, let me split my points by party:

```
> REPpoints=WApoints[WApoints$party=="REPUBLICAN",]
> DEMpoints=WApoints[WApoints$party=="DEMOCRAT",]
```

Let me prepare a similar plot to Figure 7.26, but only for the contributions to Republicans:

```
> base= ggplot(data = waCounties) + theme_void()
> counties = base + geom_sf(fill=NA)
> heatRep = counties +
```

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

```
+               stat_density2d(data = REPpoints,
+                              aes(x=Lon,
+                                  y=Lat,
+                                  fill = after_stat(level)),
+                                  geom="polygon")
```

Adding the arguments `fill` and `geom` will produce shaded areas, instead of only the borders as in Figure 7.26. Next, let me color those areas:

```
> library(RColorBrewer)
> kdpaletteRep=brewer.pal(7,"Reds")
> heatRep = heatRep +
+               scale_fill_gradientn(colours=kdpaletteRep) +
+                  guides(fill=FALSE)
>
```

My last step will be to zoom into King County, using the previous bounding box limits we used for Figure 7.18:

```
> heatRepKing = heatRep + coord_sf(xlim=forX, ylim = forY)
```

I can use the same steps to produce the map of contributors for the Democrat Party campaign:

```
> #colors
> kdpaletteDem=brewer.pal(7,"Blues")
> heatDem = counties +
+               stat_density2d(data = DEMpoints,
+                              aes(x=Lon,
+                                  y=Lat,
+                                  fill = after_stat(level)),
+                              geom="polygon")
> heatDem= heatDem +
+              scale_fill_gradientn(colours=kdpaletteDem) +
+              guides(fill=FALSE)
> heatDemKing = heatDem + coord_sf(xlim=forX, ylim = forY)
```

The maps `heatDemKing` and `heatRepKing` are shown in Figure 7.27.

The code to produce the `heatRepKing` in **Python** follows:

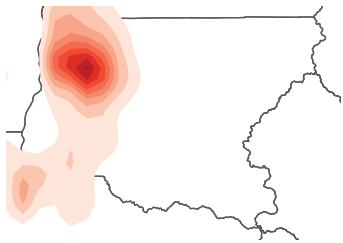
```
counties=gplt.polyplot(waCounties,edgecolor= 'black',
                       projection=gcrs.Mercator())#reproject)

gplt.kdeplot(WApoints[WApoints.party=='REPUBLICAN'], #subset
             shade=True,
             cmap='Reds',
             shade_lowest=False,
             ax=counties,
             extent=kingMap.total_bounds)
```

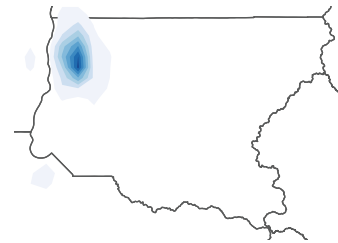
And, the code to produce the `heatDemKing` in **Python** follows:

```
counties=gplt.polyplot(waCounties,edgecolor= 'black',
                       projection=gcrs.Mercator())#reproject)
```

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press



(a) Heatplot-Contributions to Republicans.



(b) Heatplot-Contributions to Democrats.

Figure 7.27 Heatplots

```
gplt.kdeplot(WApoints[WApoints.party=='DEMOCRAT'], #subset
             shade=True,
             cmap='Blues',
             shade_lowest=False,
             ax=counties,
             extent=kingMap.total_bounds)
```

7.5.3 Final Touches

Let me pick the Figure 7.24. You have the map and the legend, but it is good practice to add some elements when mapping. First, you should consider adding caption explaining the projection you used:

```
> creditsText="EPSG:3857\nProj=Mercator"
> cartoCap=cartogram + labs(caption = creditsText)
```

Another element you should include is the *scalebar*. The easy way to add one is by using the library *ggspatial* (Dunnington, 2020).

```
> library(ggspatial)
> cartoCapSca=cartoCap +
+             annotation_scale(location = "bl", #'tr', etc.
+                              width_hint = 0.2,#size related ti plot
+                              plot_unit = 'mi',#or: 'km','ft', etc.
+                              unit_category='imperial', #or 'metric'
+                              style='ticks') # or 'bar'
```

I added familiar alternatives to the scale bar next to each argument of the function. Notice that for location you can combine *bottom* or *top* with *left* or *right* (no "middle" option) using the first letter of each word. A final element to include should be the *north arrow*, which is also facilitated by *ggspatial*:

```
> cartoCapScaNorth=cartoCapSca +
+                  annotation_north_arrow(location = "tl",
```

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

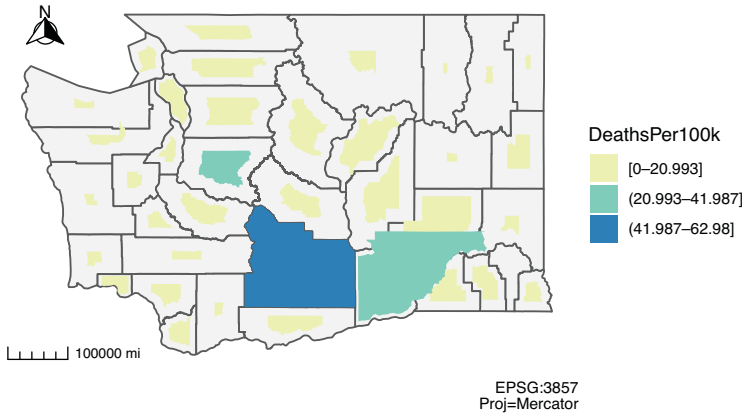


Figure 7.28 Map and its elements

Based on Figure 7.24, this plot is adding other elements: information about projections, scalebar, and north arrow. The elements were added using *ggspatial* (Dunnington, 2020). The original data comes from (Wikipedia, 2020).

```
+                          style = north_arrow_fancy_orienteering,
+                          height = unit(0.3, "in"))
```

The style of the north arrow requires a function to draw it, so do not write the style in quotations (the other basic option can be `north_arrow_minimal`; a more decorative can be `north_arrow_nautical`; you also have `north_arrow_orienteering`). You can see the final result, the object `cartoCapScaNorth`, in Figure 7.28.

In **Python**, neither *geopandas* nor *geoplot* offer simple ways to add the north arrow or the scale bar, but it is a feature suggested in their online forums. The discussion on how to do this just using *matplotlib* is directed to people with more expertise on geography. If you are an advanced user, you may try exploring how to add a scale bar by installing and using *matplotlib-scalebar* (Pinard, 2020), or by exploring the `AnchoredSizeBar` in *matplotlib*. For the north arrow, you can try the annotate function from *matplotlib*:

```
fig, ax = plt.subplots(figsize=(10, 10))
gplt.choropleth(covidMap,
                hue='CasesPer100k',
                scheme=mc.HeadTailBreaks(covidMap.CasesPer100k),
                cmap='OrRd',
                legend=True,
                legend_kwargs={'bbox_to_anchor': (1, 1),
                               'frameon': True,
                               'markeredgecolor':'k',
                               'title':"DeathsPer100k"},
                extent = covidMap.total_bounds,
```

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press

```
                ax=ax)

x, y, arrow_length = 0, 0.3, 0.3
ax.annotate('N', xy=(x, y),
            xytext=(x, y-arrow_length),
            arrowprops=dict(facecolor='black',
                            width=5,
                            headwidth=15),
            ha='center', va='center', fontsize=20,
            xycoords=ax.transAxes)
plt.show()
```

https://doi.org/10.1017/9781108625425.010 Published online by Cambridge University Press