# Outbreak of unknown origin, Berlin, Germany

*Alexander Spina, Patrick Keating, Amrish Baidjoe and Lutz Ehlkes*

This is a fictional case study about a respiratory illness of unknown source set in Berlin.

**Aims of case study:**
- To investigate the outbreak using geographic information system tools available in *R* packages
- Map cases and controls
- Apply spatial random sampling
- Visualise data
- Create a "Heat map"
- Generate hypotheses with regard to source of infection

## Prerequisites

Participants are expected to be familiar with data structures, management as well as data manipulation in R. A basic understanding of map theory is assumed. We have surpressed some of the messages and outputs in this document; so do not worry if you get messages on your console that are not shown in the document.

## Background

Excessive cases (n = 61) of a respiratory illness have been reported within the last few weeks.

Initial observations suggest the disease has high lethality, low contagiousness and the median age of affected individuals is 60 years. Cases have mostly been reported from a small area in central Berlin. The pathogen appears to be an unknown virus, potentially an Arenavirus. PCR-primers are currently being developed.

You decide to design a study in order to determine the source of the outbreak. Pharyngial swabs are being taken (consider what parameters you would choose for analysis).

You have been given a budget large enough for you to investigate 300 cases and controls.

Given the unknown nature and apparent spatial clustering, you want to select individuals for your study based on location.

## Installing and loading required packages

Several packages are required for different aspects of geospatial analysis with *R*. You will need to install these before starting. We will be using the following packages.

- *OpenStreetMap*: for downloading basemaps from OpenStreetMap
- *ggplot2*: for plotting maps and data
- *sp*: for spatial points and polygons classes as well as spatial random sampling
- *rgdal*: for reading and writing shapefiles
- *ggsn*: for adding scalebars to a plot
- *geosphere*: for calculating distance between two points

```
# Installing required packages for this case study
    # To avoid reinstalling already installed packages
required_packages <- c("OpenStreetMap", "ggplot2", "sp", "rgdal", "ggsn", "geosphere")
```

```
for(pkg in required_packages){
  if(!pkg %in% rownames(installed.packages())) {
    install.packages(pkg)
  }
}
```

Run the following code at the beginning of the case study to make sure that you have made available all the packages that you need. Be sure to include it in any scripts too.

```
# Loading required packages for this case study
required_packages <- c("OpenStreetMap", "ggplot2", "sp", "rgdal", "ggsn", "geosphere")

for (i in seq(along = required_packages)) {
  library(required_packages[i], character.only = TRUE)
}
```

## Downloading and plotting basemaps

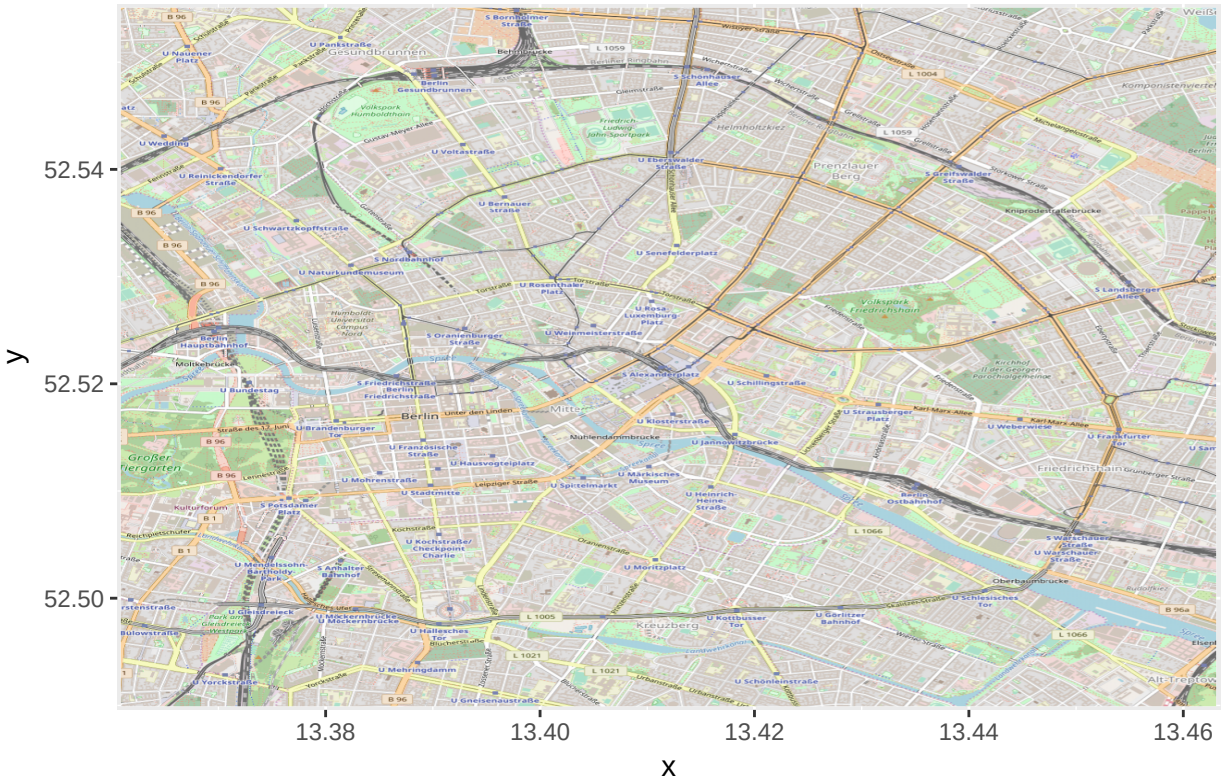
To get an idea of the area you want to map and its coordinates, you can use the "launchMapHelper()" function from the *OpenStreetMap* package. This opens a window where you can browse openstreetmap or other basemaps (e.g. bing satelite images, or ESRI). You can zoom in and out with your scroll wheel or move the map around using right click. Try selecting "Tile grid visible" from the menu bar on top to see how the map would be broken down in to small square images (called tiles). The coordinates given are sometimes a bit unreliable - so it may be better to check on the openstreetmap website itself.

```
#Opens a java window for you to browse maps interactively
launchMapHelper()
```

Once you have found the area that you want to plot, get the coordinates of the top left and bottom right corners of an imaginary box surrounding the area and put them in the *openmap* function from the OpenStreetMap package. Choose the map type you would like to load (see *?openmap* for details and the package website for examples). Choosing the minimum number of tiles requires a bit of fiddling around to get the best resolution. You then need to set the projection for your basemap using *openproj*.

Then you put that in to the *autoplot* function of the gpplot2 package. This then allows you to plot with all the functions of ggplot2 on top of your basemap.

```
#Download your area of interest and put it in object "basemap"
basemap <- openmap(upperLeft = c(lat = 52.555, lon = 13.361),
                   lowerRight = c(lat = 52.490, lon = 13.463),
                   type =  "osm",
                   minNumTiles = 12)

#Overwrite "basemap" with new projection
basemap <- openproj(basemap, projection = "+proj=longlat +ellps=WGS84 +init=epsg:4326")

#Plot this in ggplot2 using autoplot function and put in object "base"
base <- autoplot(basemap)

#View your map
base
```




*NB.* There are a lot of other packages that allow you to get basemaps from different servers. For example the *ggmap* package uses the google server. The *osmar* package allows you to download points and polygons from openstreetmap, meaning that you can then create shapefiles for buildings and roads. Another option is to use the *leaflet* package, where you can create interactive maps (see appendix).

For now we will just stick with the *OpenStreetMap* package as openstreetmap often has the most useful information and covers more areas in the world in detail. In addition this package seems to be the most stable in terms of functionality.

There will be lots of situations where you don't have a reliable internet connection. So it is a good idea to save your plotted map for later use offline. You can do this by saving and loading R-data files, using the base-R functions *save* and *load*.

```
#Save your plotted map as an R data file
save(base, file = "C:/Users/Spina/Desktop/MSF Geospatial/Berlin/base.rda")

#Reload your file at a later date
load(file = "C:/Users/Spina/Desktop/MSF Geospatial/Berlin/base.rda")

#plot your basemap
base
```

## Reading in and plotting shapefiles

Someone from the regional office has already decided on a study area and has sent you a shapefile delineating this area - they would now like you to draw a random sample within this area.

To read in shapefiles you need to use the *readOGR* function from the rgdal package. This function can actually be used to read in a range of different file types. In this case you just need to specify the folder path (called dsn) and the name of the layer; the function will then pull the various files together in to a single spatial polygon of the class "SpatialPolygonsDataFrame".

In *R*, you can also create your own polygons and save them as shapefiles; an explanation is given in the appendix.

```
#Read in your shapefile
studyarea <- readOGR(dsn = "C:/Users/Spina/Desktop/MSF Geospatial/Berlin", layer = "studyarea")
```

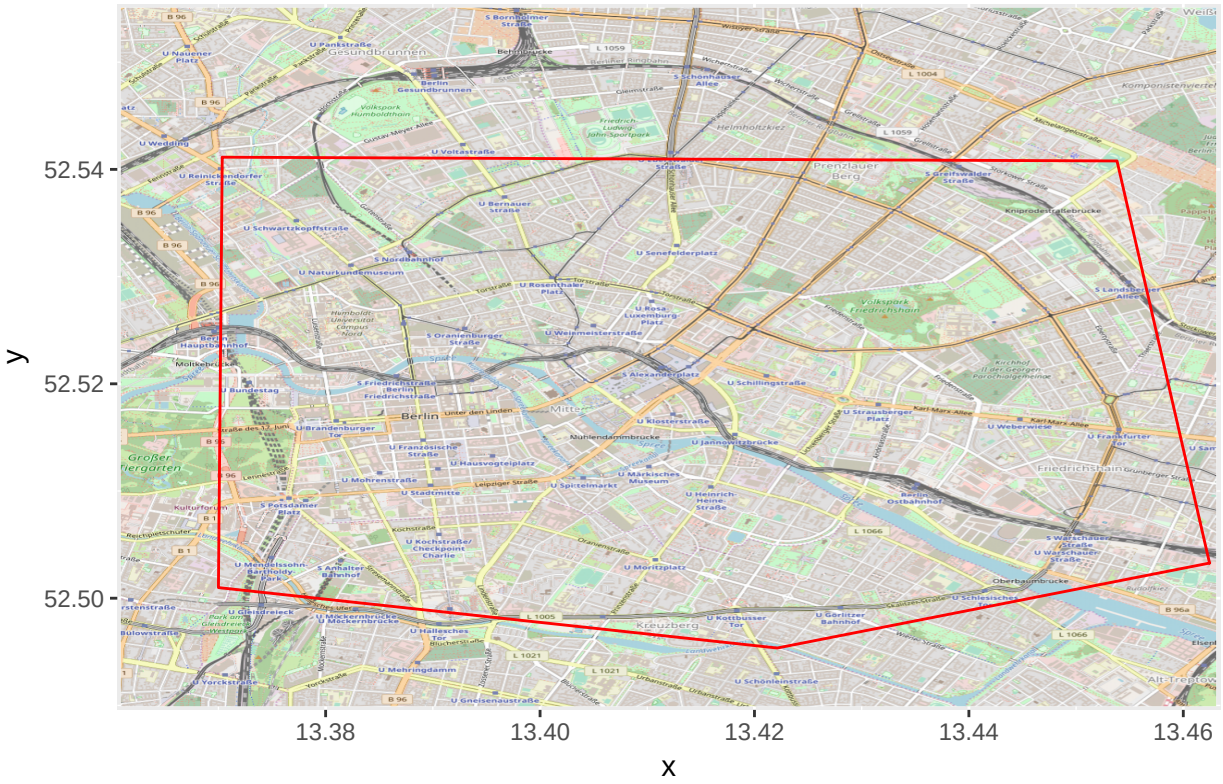
Take a look at what what is contained in your studyarea shapefile using the '@' (at-sign). Some lists allow you to access objects within them using the at-sign, rather than the normal way of using the dollar sign.

You can then plot your study area on top of your basemap. To do this you need to use the *geom_polygon* function in ggplot2. You can overwrite your already plotted "base" object, or call it something else if you want to create a new object.

Simply using the "+" after base allows you to plot on top, as you would for when plotting a normal graph in any other ggplot2.

Within *geom_polygon* your "studyarea" shapefile is treated as a dataframe. So you specify that "studyarea" is the data you want to plot, with the aesthetics (aes) you need to say what your x and y points are that you want to plot, as well as specifying which points go together in order to draw the polygon (group). You can then choose the fill (we want it empty) and the outline colour.

```
#Add a polygon on top of your basemap and overwrite the "base" object with this plot
base <- base +
    geom_polygon(data = studyarea,
                 aes(x = long, y = lat, group = group),
                 fill = NA, color = "red")

#Plot your new map
base
```




## Adding a scale bar

You may want to add a scale bar before taking a random sample in order to gauge what kind of area you will need to be covering.

To do this you need to go back to that imaginary box around your basemap. This imaginary box is called a bounding-box (bbox), and it's coordinates are saved within your downloaded (unplotted) "basemap" object. You can access it using basemap$bbox.

Your bbox is separated in two different lists, however ggplot2 needs these in a dataframe. To create a dataframe which can be used in ggplot2, you need to select and rearrange a bit.

Once you have it in the correct shape you can simply pass these points in to the *scalebar* function (see ?scalebar) of the ggsn package; which can then be integrated in a ggplot2 plot. You need to specify the distance (in our case 1 km), whether you would like it to be in kilometres, the projection system and finally where in your grid you would like to plot this scalebar (anchor).

```
#reformat your bbox to have it in a dataframe for the scalebar
bb2 <- data.frame(long = unlist(basemap$bbox)[c(1,3)],
                  lat = unlist(basemap$bbox)[c(2,4)])

#Add your scale bar to the ggplot with your basemap and polygon already plotted from above
base <- base +

    scalebar(data = bb2, dist = 1, dd2km = TRUE, model  = "WGS84",
           anchor = c(x = 13.46, y = 52.495))
```

## Spatial random sampling

Now you are ready to take your random sample. To do this simply use the *spsample* function from the sp package. Specify your polygon, the number of points you want and what method you would like to use (see ?spsample for options). In this case we sample 306 to have a few extra incase of non-responders. For an example of how to retrieve shapefiles of buildings from openstreetmap and use these for random sampling, see the appendix.

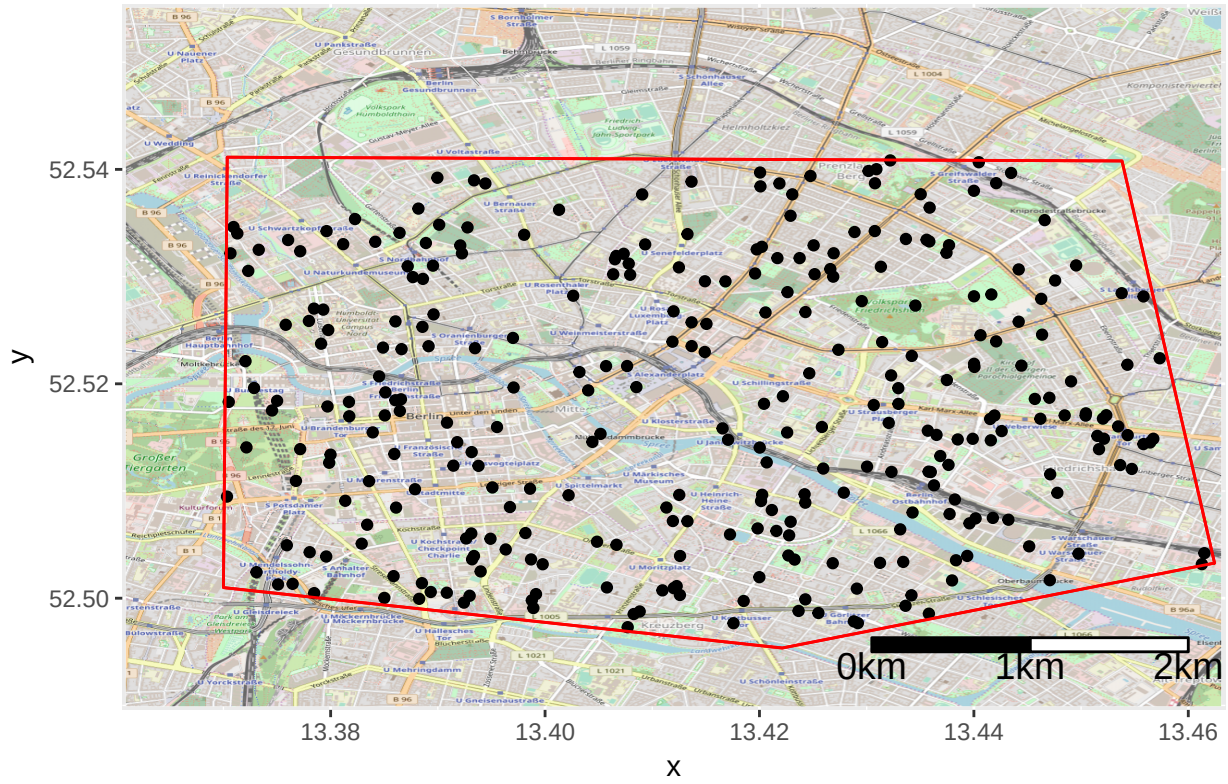
```
studypts <- spsample(studyarea, 306, "random")
```

Your randomly selected points are saved within a list in "studypts". You can specify the projection you would like these points to have (though in this case they will already be suitable for the projection we are working in).

Using the '@'(at-sign) and specifying "coords"", you can get to the lon/lat values for these points. Unlike the dollar-sign, the at-sign allows you to take both columns (longitude and latitude) at the same time.
You can then save these values in a dataframe and add them to your ggplot just to check if they are within the polygon.

```
#Specify the projection you would like to use
studypts <- SpatialPoints(studypts,
              proj4string = CRS("+proj=longlat +ellps=WGS84 +init=epsg:4326"))

#Overwrite your studypts object with the coords in a dataframe
studypts <- data.frame(studypts@coords)

#plot your points but dont assign them to an object just yet
base + geom_point(data = studypts, aes(x = x, y = y)) +
  geom_polygon(data = studyarea,
                 aes(x = long, y = lat, group = group),
                 fill = NA, color = "red")
```

```
## Regions defined for each Polygons
```




You can also export these points to a file (csv for example) which can then be loaded on to a phone, garmin or other device for field work.

```
write.csv(studypts, "C:/Users/Spina/Desktop/MSF Geospatial/Berlin/studypts.csv", row.names = F)
```

Alternatively if you wanted to export to a GPX file, you could use the following code. This GPX points can then be imported to the *osmand* mobile application for navigating to points.

```
#Create a waypoints (or counts variable)
studypts$a <- c(1:306)

#Change this to a "spatial points dataframe" with appropriate coordinate reference
latslongs <- SpatialPointsDataFrame(coords = studypts[,c(1,2)],
                                    data = studypts,proj4string =CRS("+proj=longlat + ellps=WGS84"))


#Write a GPX file with the writeOGR function from rgdal package
writeOGR(latslongs,
         dsn = "C:/Users/Spina/Desktop/MSF Geospatial/Berlin/gpxTEST.gpx",
         dataset_options = "GPX_USE_EXTENSIONS=yes",
         layer = "waypoints", driver="GPX", overwrite_layer = T)
```

## Plotting points

Teams have been round to houses, swabbed the ~300 people and asked questions. They send you some initial results (see Table 1) and a dataset, asking you to do some spatial analysis.

*Table 1:* Univariate logistic regression showing risk factors for infection

| Risk factor | Odds ratio | P-value | 95% Confidence interval |
|---|---|---|---|
| Smoker | 1.4 | 0.06 | 0.9-1.8 |
| Unemployed/retired | 3.0 | <0.001 | 2.8-4.0 |
| immunosurpressed | 8.6 | 0.37 | 0.8-12.6 |
| Male | 1.1 | 0.86 | 0.7-1.5 |
| Age category | | | |
| 0-20 | Ref | | |
| 21-40 | 0.6 | 0.03 | 0.4-0.9 |
| 41-60 | 1.3 | 0.24 | 0.8-2.0 |
| 60+ | 6.7 | <0.01 | 6.1-8.5 |

You are sent a CSV file, but because it was entered using a german computer the separator is a semi-colon instead of a comma. Using *read.csv* you can choose the separator value and thereby read in the case linelist.

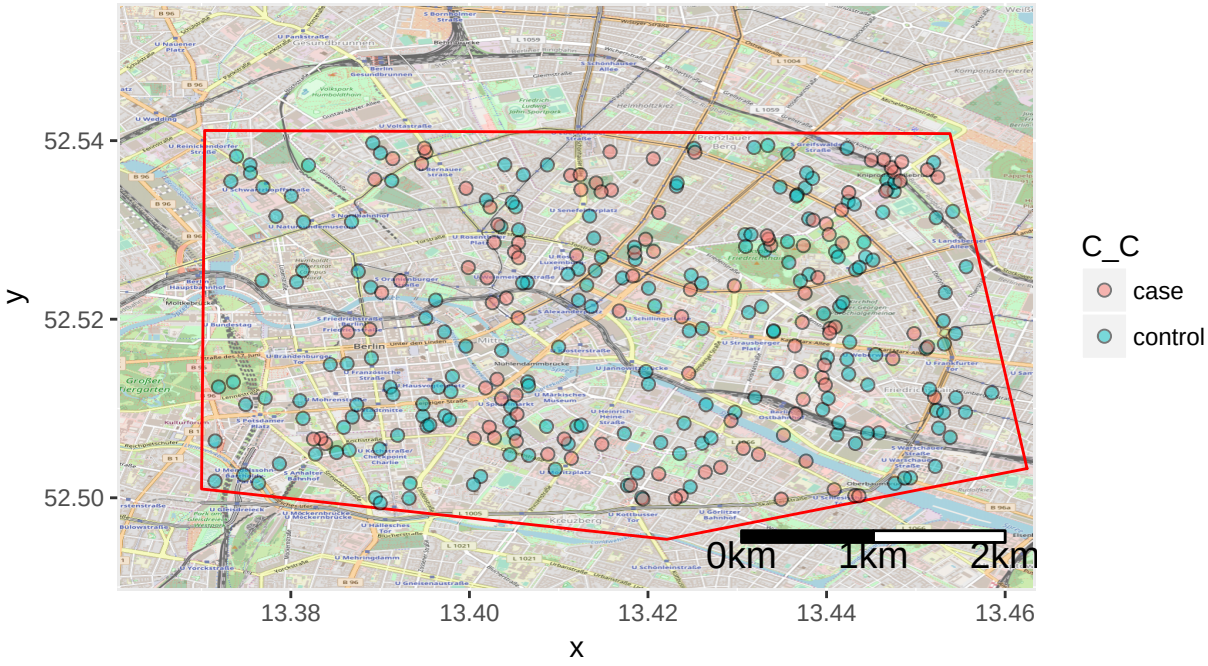
```
#Read in the linelist
cases <- read.csv("C:/Users/Spina/Desktop/MSF Geospatial/Berlin/Case_Control.csv", sep = ";")

#Change the case definition variable from binary to categorical
cases$C_C <- ifelse(cases$C_C == 1, "case", "control")
```

If the devices used to collect data had been in a different projection than what we are using, you could change them with the code below. Luckily this was not the case in our dataset so there is no need to do this, we can simply plot the values for longitude and latitude as they are.

```
#bind your two columns of longitude and latitude together
  #pass them through SpatialPoints and define the reference system
caseptslonglat <- SpatialPoints(cbind(lon = cases$Y ,
                                      lat = cases$X),
                                proj4string = CRS("+proj=longlat +ellps=WGS84 +init=epsg:4326"))

#bind your new SpatialPoints columns to your dataset
cases <- cbind(cases, caseptslonglat@coords)
```

You have already seen above how you would plot points using *geom_point*. Where you specify the dataset as well as the x and y coordinates. We will now try using *geom_jitter*, which slightly shifts overlapping points (called overplotting), so that you can see all the points on the map. We will also specify in aesthetics that we want to fill the points based on their category in the "C_C" variable. You can additionally specify the shape and size that you would like your point to be. See this website for options. In this case we specify shape 21 because it has a border and so can be filled in. You can also choose how see-through you would like your point to be using alpha; if you dont want it see-through at all, simply remove the argument. Finally, you can specify whether you would like to show the legend or not.

```
#add jittered points to your basemap and store in "dotmap"
dotmap <- base +
  geom_jitter(data = cases, aes(x = X, y = Y, fill = C_C),
              shape = 21,
              size = 2,
              alpha = 0.5,
              show.legend = TRUE)

#view your new plot
dotmap
```



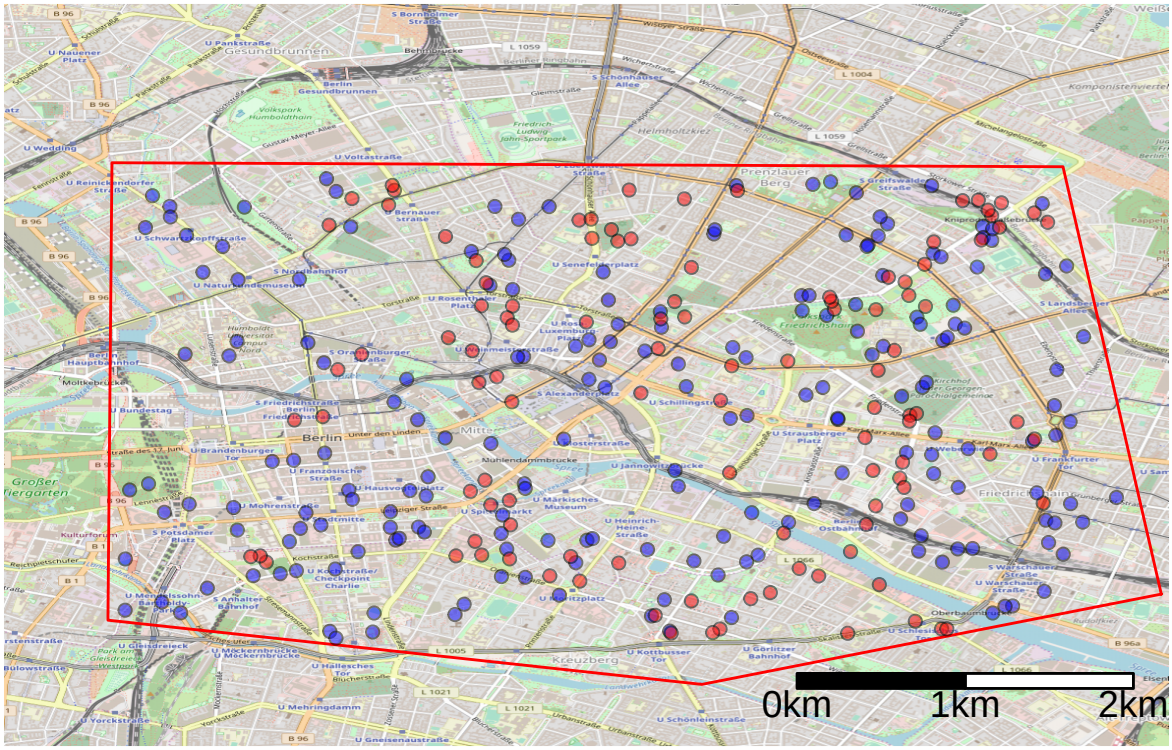

You can also choose the colours you would like to have have using *scale_fill_manual*. See an example of available colours and what they are called at this website.

```
#Change the colours you want in your dots
dotmap <- base +
  geom_jitter(data = cases, aes(x = X, y = Y, fill = C_C),
              shape = 21,
              size = 2,
              alpha = 0.5,
              show.legend = TRUE) +
  scale_fill_manual(values = c("Red", "Blue"))
```

Finally, you can edit parts of your plot using the *theme* function. You can for example get rid of the axes and labels (using *theme_void*) as well as edit parts of the legend. For example, you can remove the legend header and adjust the legend position or size of text.

The code below shows what your code would look like if you put all the code together rather than going step by step and overwriting "base".

```
#plot your basemap
dotmap <- autoplot(basemap) +

  #plot your study area shapefile
          geom_polygon(data = studyarea,
                 aes(x = long, y = lat, group = group),
                 fill = NA, color = "red") +

  #plot your scalebar
          scalebar(data = bb2, dist = 1, dd2km = TRUE, model  = "WGS84",
                   anchor = c(x = 13.46, y = 52.495)) +
  #plot your points
          geom_jitter(data = cases, aes(x = X, y = Y, fill = C_C),
                      shape = 21,
                      size = 2,
                      alpha = 0.5,
                      show.legend = T) +
  #choose the colour for your points
          scale_fill_manual(values = c("Red", "Blue")) +
  #remove axes and labels
          theme_void() +
  #adjust legend
          theme(legend.title = element_blank(), legend.key = element_blank(),
                legend.text = element_text(size = 12), legend.position = "bottom")


#view plot
dotmap
```




You can then also save your plot as an image file using *ggsave* - in this case we have saved it as a tiff file, however you can choose from a variety of different formats (see ?ggsave).

```
ggsave(dotmap, file = "C:/Users/Spina/Desktop/MSF Geospatial/Berlin/DotMap.tiff",
       device = "tiff", width = 6, height = 6)
```

## Plotting "Heatmaps"

You now have a meeting and would like to show a map to your colleagues that intuitively highlights the areas with increased case numbers.

A solution is to produce what is known as a "heat map" of cases. These are produced using kernel density estimates calculated using gaussian distributions. You can produce this type of map using the *stat_density_2d* function in ggplot2. You could just use it in a straight forward way, not specifying very many options; however this will not necessarily give you a very informative plot.

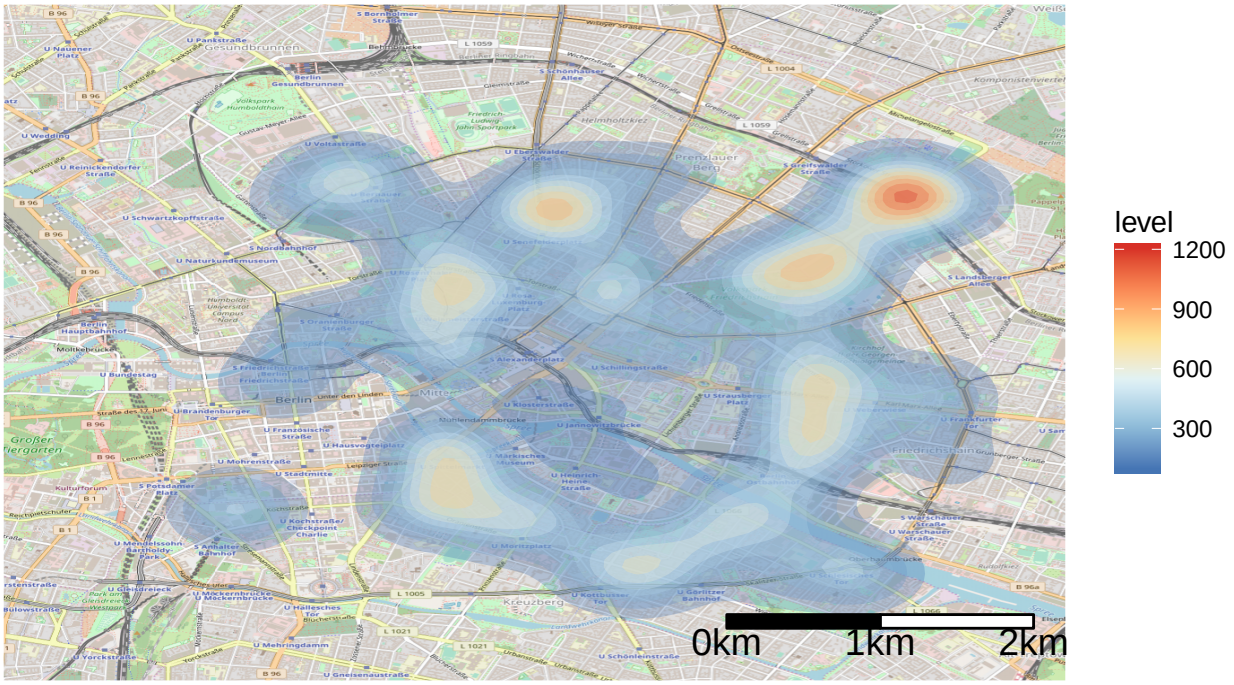
Here we demonstrate how you can specify that the diameter of the distribution used for kernel density estimation to be 1 kilometre. To do this we need to go back to our imaginary box around our basemap, the bounding box (bb2). To do this you need to calculate how many units of longitude and of latitude are equivalent to one kilometre. Then using those units in the *h* option for the *stat_density_2d* function.

To take a look at what combination of colours would be the most optimal, check out this website.

In this case you may want to surpress the legend - because it's units are not very intuitive.

```
#define your x and y minimum/maximum coordinates

xmin <- c( min(bb2$long), min(bb2$lat) )
xmax <- c( max(bb2$long), min(bb2$lat) )

ymin <- c( min(bb2$long), min(bb2$lat) )
ymax <- c( min(bb2$long), max(bb2$lat) )

#calculate the distance in metres on the axes of your basemap based on longitude and latitude
    #i.e. how many metres on your x axis and how many on your y
xdist <- distm(xmin, xmax )

ydist <- distm(ymin, ymax )

#divide the difference in latitude or longitude by the corresponding distance
    #multiply by 1000 to get how many units latitude and longitude correspond to one kilometre
xfact <- as.numeric( (max(bb2$long) - min(bb2$long) ) / xdist * 1000)
yfact <- as.numeric( (max(bb2$lat) - min(bb2$lat) ) / ydist * 1000)


#create subset of only cases
confirmedcases <- cases[which(cases$C_C == "case"), ]


#plot your basemap
densitymap <- autoplot(basemap) +

  #Statdensity creates your kernel density. Similar to other functions you specify data and X/Y
    #You then specify you want to fill with the level calculated from the kernel density
    #You specify you want a polygon shape
    #Then you pass your 1km worth of units for longitude and latitude (xfact and yfact) through h
              stat_density_2d(data = confirmedcases, aes(x = X, y = Y, fill = ..level..),
                              alpha = 0.3,  geom = "polygon", h = c(xfact, yfact),
                              show.legend = T) +

  #Select the colour combination you would like
              scale_fill_distiller(palette = "RdYlBu") +
```

```
  #add a scalebar
              scalebar(data = bb2, dist = 1, dd2km = TRUE, model  = "WGS84",
                       anchor = c(x = 13.46, y = 52.495)) +
  #remove axes and labels
             theme_void()

#view plot
densitymap
```




You can also save this map as an image if you need to.

```
ggsave(densitymap, file = "C:/Users/Spina/Desktop/MSF Geospatial/Berlin/DensityMap.tiff",
       device = "tiff", width = 6, height = 6)
```

## Conclusion

This outbreak appeared to be caused by a novel rodent-borne disease. Transmission was via bodily fluids and excretions. There was a short incubation period and high lethality. There appeared to be no person-to-person transmission. Based on the spatial analyses, it seemed as though the majority of cases were clustered near parks. Further investigation suggested that cases contracted the disease while conducting recreational activities in parks, such as barbequeing or feeding squirrels.

# Appendix

## Create your own polygon shapefile

```
#load the raster package
  #NOTE! The raster package stops scalebar from working
library(raster)


#plot your basemap (using the normal R plotting function - not ggplot2)
plot(basemap)

#drawPoly() allows you to click points on your plot outlining your polygon
  #when done, push escape and it will put your polygon in "mypolygon"
  #your polygon is of the class "SpatialPolygon"
mypolygon <- drawPoly()

#To save this as a shapefile, first convert to a spatial polygons dataframe
mypolygon <- as(mypolygon, "SpatialPolygonsDataFrame")

#Using writeOGR() from the rgdal package
  #Specify where you would like to save your shape file (dsn)
  #Then specify what you would like to call it (layer)
  #Also specify the driver in order to create a shapefile
library(rgdal)

writeOGR(obj = mypolygon, dsn = "C:/Users/Spina/Desktop/MSF Geospatial/Berlin",
         layer = "studyarea", driver = "ESRI Shapefile")
```

## Downloading buildings from OSM and sampling points over houses

Take a look at the walk-through available on this website.

In this situation, downloading the buildings can take a while because there are so many in central berlin, (aproximately 20,000). As it is a densley population city, these might not all be households; however if this was a rural area or a refugee camp, then you would be selecting only households.

```
#install package
install.packages("osmdata")

#load package to session
  #NOTE! The raster package stops scalebar from working
library("osmdata")
library("raster")

#choose which overpass server to connect to
  #(apparently can be quite unstable)

set_overpass_url('https://overpass.kumi.systems/api/interpreter')



#define your bounding box
```

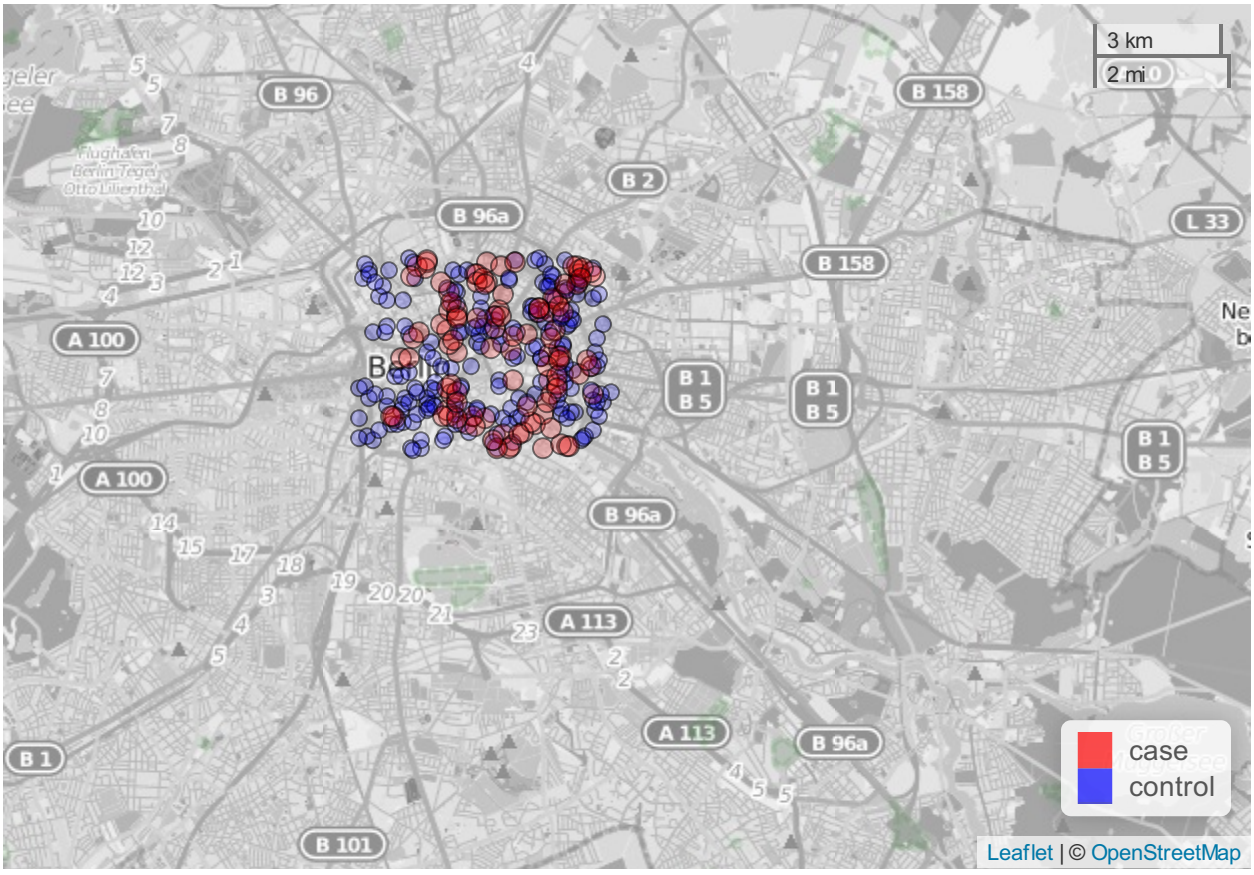
```
  #Nb. you could also just use bbox = "berlin" but would take forever
  #Bounding box is different to with basemap (here is: bottom left, top right)
overpassquery <- opq(bbox = studyarea@bbox)

#define what data you would like to download
dl <- add_osm_feature(overpassquery, key = "building")

#save as a spatialpolygon
buildings <- osmdata_sp(dl)

#Can take a look at what is in using buildings
  #We are interested in the polygons (which are a spatialpolygonsdataframe)
#View(buildings)

#extract only the polygons
buildings <- buildings$osm_polygons



#Make sure your studyarea has correct coordinate reference system
studyarea@proj4string <- CRS("+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs +towgs84=0,0,0")


#Restrict your buildings to those within the study area polygon
  #This is because the studyarea is not a perfect square!
      #important if you have a weirdly shaped study area
  #Using intersect function from raster package
restricter <- intersect(buildings, studyarea)


#Take your random sample of points from within building polygons
studypts <- spsample(restricter, 300, "random")
```

## Creating interactive maps with leaflet

With leaflet, adding a basemap is builtin to the package; you can add this using the *addProviderTiles*. After that you can then plot points using your cases dataframe.

```
#load necessary packages
  #for creating interactive maps
library(leaflet)
  #for saving images
library(mapview)
  #for saving HTML document (for sharing interactive map)
library(htmlwidgets)

#set your colours based on case definitions
fillpal <- colorFactor(c("Red", "Blue"), cases$C_C)

#start your plot by selecting your dataset
  #with options - remove the zoom bar (you can still zoom with your moouse)
leafmap <- leaflet(cases, options = leafletOptions(zoomControl = FALSE)) %>%
```

```
  #select your area of interest for plotting
            fitBounds(13.36, 52.56, 13.46, 52.49) %>%

  #select the background you want to plot with
            addProviderTiles("OpenStreetMap.BlackAndWhite", group = "Black&White") %>%

  #add points for your cases and controls
            addCircleMarkers(lng = jitter(cases$X, factor = 0.5),
                             lat = jitter(cases$Y, factor = 0.5),
                             radius = ~ifelse(C_C == "case", 5, 3.5),
                             color = "black",
                             fillColor = ~fillpal(C_C),
                             stroke = T, weight = 1,
                             fillOpacity = NULL) %>%
  #add a legend
            addLegend("bottomright",
                      pal = fillpal,
                      values = ~C_C,
                      title = "", opacity = NULL) %>%
  #add a scalebar
            addScaleBar()

#view your plot
leafmap
```




You can also save the map as either a static image or an interactive HTML widget with the code below.

```r
#save the interactive map to a static image file.
mapshot(leafmap, file = "sitesmap2.png")

# Save the interactive map to an HTML page.
saveWidget(leafmap, file = "sitesmap2.html", selfcontained = TRUE)
```