Spatial Analysis Notes

Francisco Rowe & Dani Arribas-Bel

2020-02-12

Contents

Chapter 1

Spatial Analysis Notes

This book contains computational illustrations on spatial analytical approaches using R.

Chapter 2

Introduction

This session[1] introduces R Notebooks, basic functions and data types. These are all important concepts that we will use during the module.

If you are already familiar with R, R notebooks and data types, you may want to jump to Section Read Data and start from there. This section describes how to read and manipulate data using `sf` and `tidyverse` functions, including `mutate()`, `%>%` (known as pipe operator), `select()`, `filter()` and specific packages and functions how to manipulate spatial data.

The content of this session is based on the following references:

- Grolemund and Wickham (2019), this book illustrates key libraries, including tidyverse, and functions for data manipulation in R
- Xie et al. (2019), excellent introduction to R markdown!
- Williamson (2018), some examples from the first lecture of ENVS450 are used to explain the various types of random variables.
- Lovelace et al. (2020), a really good book on handling spatial data and historical background of the evolution of R packages for spatial data analysis.

[1]This note is part of Spatial Analysis Notes Introduction – R Notebooks + Basic Functions + Data Types by Francisco Rowe is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

2.1 Dependencies

This tutorial uses the libraries below. Ensure they are installed on your machine[2] before loading them executing the following code chunk:

```
# Data manipulation, transformation and visualisation
library(tidyverse)
# Nice tables
library(kableExtra)
# Simple features (a standardised way to encode vector data ie. points, lines, polygon
library(sf)
# Spatial objects conversion
library(sp)
# Thematic maps
library(tmap)
# Colour palettes
library(RColorBrewer)
# More colour palettes
library(viridis) # nice colour schemes
```

2.2 Introducing R

R is a freely available language and environment for statistical computing and graphics which provides a wide variety of statistical and graphical techniques. It has gained widespread use in academia and industry. R offers a wider array of functionality than a traditional statistics package, such as SPSS and is composed of core (base) functionality, and is expandable through libraries hosted on CRAN. CRAN is a network of ftp and web servers around the world that store identical, up-to-date, versions of code and documentation for R.

Commands are sent to R using either the terminal / command line or the R Console which is installed with R on either Windows or OS X. On Linux, there is no equivalent of the console, however, third party solutions exist. On your own machine, R can be installed from here.

Normally RStudio is used to implement R coding. RStudio is an integrated development environment (IDE) for R and provides a more user-friendly front-end to R than the front-end provided with R.

To run R or RStudio, just double click on the R or RStudio icon. Throughout this module, we will be using RStudio:

[2] You can install package `mypackage` by running the command `install.packages("mypackage")` on the R prompt or through the `Tools --> Install Packages...` menu in RStudio.

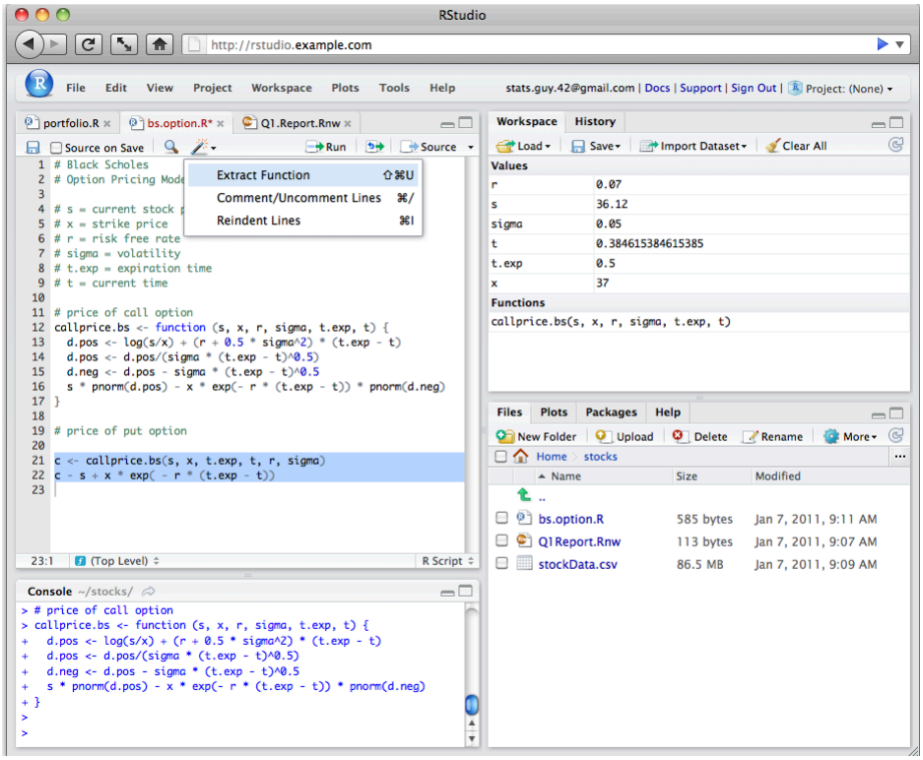


Figure 2.1: Fig. 1. RStudio features.

If you would like to know more about the various features of RStudio, watch this video

2.3 Setting the working directory

Before we start any analysis, ensure to set the path to the directory where we are working. We can easily do that with `setwd()`. Please replace in the following line the path to the folder where you have placed this file -and where the `data` folder lives.

```
#setwd('../data/sar.csv')
#setwd('.')
```

Note: It is good practice to not include spaces when naming folders and files. Use *underscores* or *dots*.

You can check your current working directory by typing:

```
getwd()
```

```
## [1] "/home/jovyan/work"
```

2.4 R Scripts and Notebooks

An *R script* is a series of commands that you can execute at one time and help you save time. So you don't repeat the same steps every time you want to execute the same process with different datasets. An R script is just a plain text file with R commands in it.

To create an R script in RStudio, you need to

- Open a new script file: *File* > *New File* > *R Script*
- Write some code on your new script window by typing eg. `mtcars`
- Run the script. Click anywhere on the line of code, then hit *Ctrl + Enter* (Windows) or *Cmd + Enter* (Mac) to run the command or select the code chunk and click *run* on the right-top corner of your script window. If do that, you should get:

```
mtcars
```

```
##                    mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
```

```
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
## Merc 450SLC         15.2   8 275.8 180 3.07 3.780 18.00  0  0    3    3
## Cadillac Fleetwood  10.4   8 472.0 205 2.93 5.250 17.98  0  0    3    4
## Lincoln Continental 10.4   8 460.0 215 3.00 5.424 17.82  0  0    3    4
## Chrysler Imperial   14.7   8 440.0 230 3.23 5.345 17.42  0  0    3    4
## Fiat 128            32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic         30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla      33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Toyota Corona       21.5   4 120.1  97 3.70 2.465 20.01  1  0    3    1
## Dodge Challenger    15.5   8 318.0 150 2.76 3.520 16.87  0  0    3    2
## AMC Javelin         15.2   8 304.0 150 3.15 3.435 17.30  0  0    3    2
## Camaro Z28          13.3   8 350.0 245 3.73 3.840 15.41  0  0    3    4
## Pontiac Firebird    19.2   8 400.0 175 3.08 3.845 17.05  0  0    3    2
## Fiat X1-9           27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2       26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa        30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Ford Pantera L      15.8   8 351.0 264 4.22 3.170 14.50  0  1    5    4
## Ferrari Dino        19.7   6 145.0 175 3.62 2.770 15.50  0  1    5    6
## Maserati Bora       15.0   8 301.0 335 3.54 3.570 14.60  0  1    5    8
## Volvo 142E          21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

- Save the script: *File* > *Save As*, select your required destination folder, and enter any filename that you like, provided that it ends with the file extension *.R*

An *R Notebook* is an R Markdown document with descriptive text and code chunks that can be executed independently and interactively, with output visible immediately beneath a code chunk - see Xie et al. (2019).

To create an R Notebook, you need to:

- Open a new script file: *File* > *New File* > *R Notebook*
- Insert code chunks, either:

1) use the *Insert* command on the editor toolbar;
2) use the keyboard shortcut *Ctrl + Alt + I* or *Cmd + Option + I* (Mac); or,
3) type the chunk delimiters ```` ```{r} ```` and ```` ``` ````

In a chunk code you can produce text output, tables, graphics and write code! You can control these outputs via chunk options which are provided inside the

```
---
title: "My Notebook"
output: html_notebook
---
```

Figure 2.2: Fig. 2. YAML metadata for notebooks.

curly brackets eg.

- Execute code: hit *"Run Current Chunk"*, *Ctrl + Shift + Enter* or *Cmd + Shift + Enter* (Mac)
- Save an R notebook: *File > Save As*. A notebook has a `*.Rmd` extension and when it is saved a `*.nb.html` file is automatically created. The latter is a self-contained HTML file which contains both a rendered copy of the notebook with all current chunk outputs and a copy of the *.Rmd file itself.

Rstudio also offers a *Preview* option on the toolbar which can be used to create pdf, html and word versions of the notebook. To do this, choose from the drop-down list menu `knit to ...`

2.5 Getting Help

You can use `help` or `?` to ask for details for a specific function:

```
help(sqrt) #or ?sqrt
```

And using `example` provides examples for said function:

```
example(sqrt)
```

```
##
## sqrt> require(stats) # for spline
##
## sqrt> require(graphics)
##
## sqrt> xx <- -9:9
##
## sqrt> plot(xx, sqrt(abs(xx)),  col = "red")
##
## sqrt> lines(spline(xx, sqrt(abs(xx)), n=101), col = "pink")
```

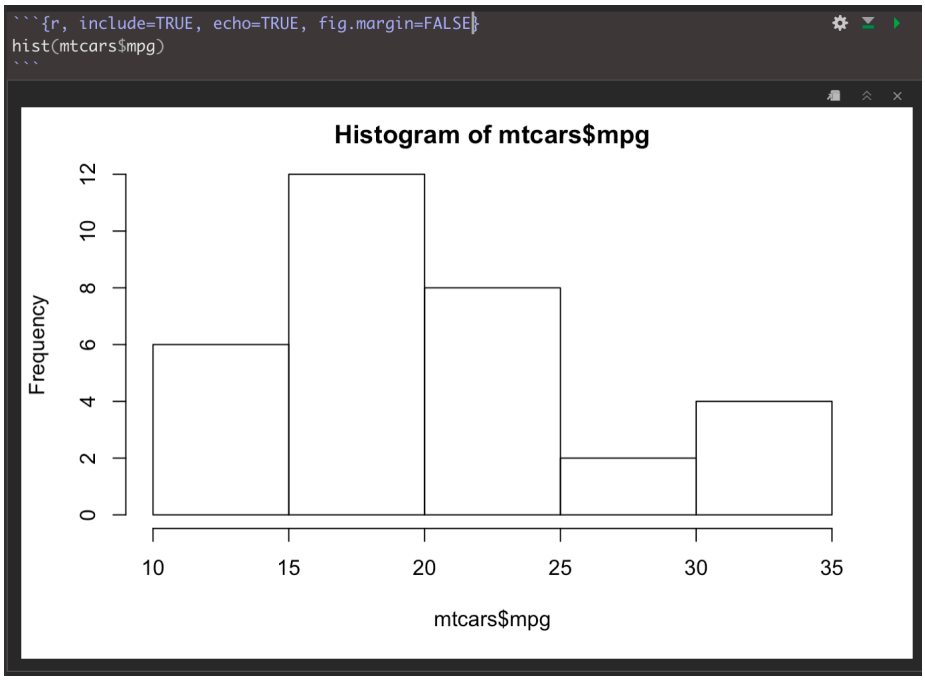


Figure 2.3: Fig. 3. Code chunk example. Details on the various options: https://rmarkdown.rstudio.com/lesson-3.html

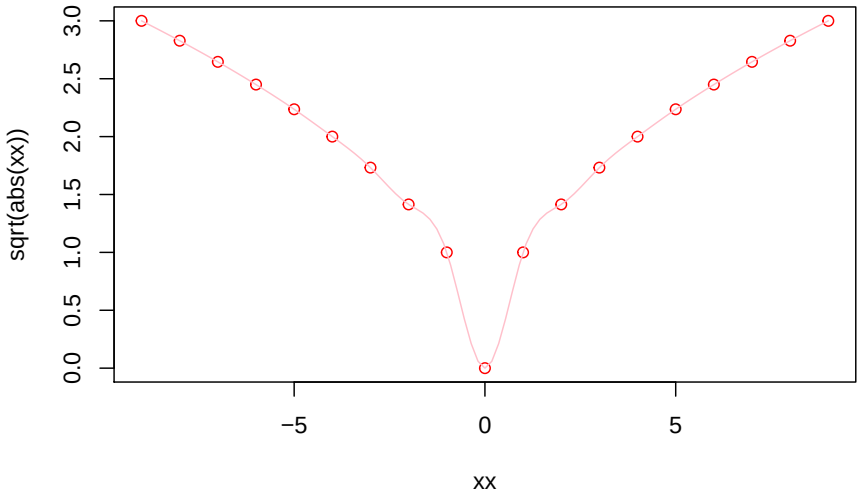


Figure 2.4: Example sqrt

2.6 Variables and objects

An *object* is a data structure having attributes and methods. In fact, everything in R is an object!

A *variable* is a type of data object. Data objects also include list, vector, matrices and text.

- Creating a data object

In R a variable can be created by using the symbol `<-` to assign a value to a variable name. The variable name is entered on the left `<-` and the value on the right. Note: Data objects can be given any name, provided that they start with a letter of the alphabet, and include only letters of the alphabet, numbers and the characters `.` and `_`. Hence AgeGroup, Age_Group and Age.Group are all valid names for an R data object. Note also that R is case-sensitive, to agegroup and AgeGroup would be treated as different data objects.

To save the value *28* to a variable (data object) labelled *age*, run the code:

```
age <- 28
```

- Inspecting a data object

To inspect the contents of the data object *age* run the following line of code:

```
age
```

```
## [1] 28
```

Find out what kind (class) of data object *age* is using:

```
class(age)
```

```
## [1] "numeric"
```

Inspect the structure of the *age* data object:

```
str(age)
```

```
##  num 28
```

- The *vector* data object

What if we have more than one response? We can use the `c( )` function to combine multiple values into one data vector object:

```
age <- c(28, 36, 25, 24, 32)
age
```

```
## [1] 28 36 25 24 32
```

```
class(age) #Still numeric..
```

```
## [1] "numeric"
```

```
str(age) #..but now a vector (set) of 5 separate values
```

```
##  num [1:5] 28 36 25 24 32
```

Note that on each line in the code above any text following the `#` character is ignored by R when executing the code. Instead, text following a `#` can be used to add comments to the code to make clear what the code is doing. Two marks of good code are a clear layout and clear commentary on the code.

2.6.1 Basic Data Types

There are a number of data types. Four are the most common. In R, **numeric** is the default type for numbers. It stores all numbers as floating-point numbers (numbers with decimals). This is because most statistical calculations deal with numbers with up to two decimals.

- Numeric

```
num <- 4.5 # Decimal values
class(num)
```

```
## [1] "numeric"
```

- Integer

```
int <- as.integer(4) # Natural numbers. Note integers are also numerics.
class(int)
```

```
## [1] "integer"
```

- Character

```
cha <- "are you enjoying this?" # text or string. You can also type `as.character("are
class(cha)
```

```
## [1] "character"
```

- Logical

```
log <- 2 < 1 # assigns TRUE or FALSE. In this case, FALSE as 2 is greater than 1
log
```

```
## [1] FALSE
```

```
class(log)
```

```
## [1] "logical"
```

2.6.2 Random Variables

In statistics, we differentiate between data to capture:

- *Qualitative attributes* categorise objects eg.gender, marital status. To measure these attributes, we use *Categorical* data which can be divided into:
  - *Nominal* data in categories that have no inherent order eg. gender
  - *Ordinal* data in categories that have an inherent order eg. income bands
- *Quantitative attributes*:
  - *Discrete* data: count objects of a certain category eg. number of kids, cars
  - *Continuous* data: precise numeric measures eg. weight, income, length.

In R these three types of random variables are represented by the following types of R data object:

variables	objects
nominal	factor
ordinal	ordered factor
discrete	numeric
continuous	numeric

We have already encountered the R data type *numeric*. The next section introduces the *factor* data type.

2.6.2.1 Factor

What is a factor?

A factor variable assigns a numeric code to each possible category (*level*) in a variable. Behind the scenes, R stores the variable using these numeric codes to save space and speed up computing. For example, compare the size of a list of `10,000` *males* and *females* to a list of `10,000` `1s` and `0s`. At the same time R also saves the category names associated with each numeric code (level). These are used for display purposes.

For example, the variable *gender*, converted to a factor, would be stored as a series of `1s` and `2s`, where `1 = female` and `2 = male`; but would be displayed in all outputs using their category labels of *female* and *male*.

Creating a factor

To convert a numeric or character vector into a factor use the `factor( )` function. For instance:

```
gender <- c("female","male","male","female","female") # create a gender variable
gender <- factor(gender) # replace character vector with a factor version
gender
```

```
## [1] female male   male   female female
## Levels: female male
```

```
class(gender)
```

```
## [1] "factor"
```

```
str(gender)
```

```
##  Factor w/ 2 levels "female","male": 1 2 2 1 1
```

Now *gender* is a factor and is stored as a series of `1s` and `2s`, with `1s` representing `females` and `2s` representing `males`. The function `levels( )` lists the levels (categories) associated with a given factor variable:

```
levels(gender)
```

```
## [1] "female" "male"
```

The categories are reported in the order that they have been numbered (starting from 1). Hence from the output we can infer that `females` are coded as 1, and `males` as 2.

2.7 Data Frames

R stores different types of data using different types of data structure. Data are normally stored as a *data.frame*. A data frames contain one row per observation (e.g. wards) and one column per attribute (eg. population and health).

We create three variables wards, population (`pop`) and people with good health (`ghealth`). We use 2011 census data counts for total population and good health for wards in Liverpool.

```
wards <- c("Allerton and Hunts Cross","Anfield","Belle Vale","Central","Childwall","Chu

pop <- c(14853,14510,15004,20340,13908,13974,15272,14045,14503,
                14561,14782,16786,16132,15377,16115,13312,13816,15047,
                16461,17009,17104,18422,12991,20300,16489,16481,14772,
                14382,12921,16746)

ghealth <- c(7274,6124,6129,11925,7219,7461,6403,5930,7094,6992,
                 5517,7879,8990,6495,6662,5981,7322,6529,7192,7953,
                 7636,9001,6450,8973,7302,7521,7268,7013,6025,7717)
```

Note that `pop` and `ghealth` and `wards` contains characters.

2.7.1 Creating A Data Frame

We can create a data frame and examine its structure:

```
df <- data.frame(wards, pop, ghealth)
df # or use view(data)
```

```
##                       wards   pop ghealth
## 1  Allerton and Hunts Cross 14853    7274
## 2                   Anfield 14510    6124
## 3                Belle Vale 15004    6129
## 4                   Central 20340   11925
## 5                 Childwall 13908    7219
## 6                    Church 13974    7461
## 7                  Clubmoor 15272    6403
## 8                    County 14045    5930
## 9                Cressington 14503    7094
## 10                  Croxteth 14561    6992
```

```
## 11                  Everton 14782    5517
## 12               Fazakerley 16786    7879
## 13                Greenbank 16132    8990
## 14 Kensington and Fairfield 15377    6495
## 15                 Kirkdale 16115    6662
## 16               Knotty Ash 13312    5981
## 17             Mossley Hill 13816    7322
## 18             Norris Green 15047    6529
## 19                 Old Swan 16461    7192
## 20                   Picton 17009    7953
## 21             Princes Park 17104    7636
## 22                Riverside 18422    9001
## 23             St Michael's 12991    6450
## 24            Speke-Garston 20300    8973
## 25 Tuebrook and Stoneycroft 16489    7302
## 26                 Warbreck 16481    7521
## 27                Wavertree 14772    7268
## 28               West Derby 14382    7013
## 29                  Woolton 12921    6025
## 30                  Yew Tree 16746    7717
```

```
str(df) # or use glimpse(data)
```

```
## 'data.frame':    30 obs. of  3 variables:
##  $ wards  : Factor w/ 30 levels "Allerton and Hunts Cross",..: 1 2 3 4 5 6 7 8 9 10 ...
##  $ pop    : num  14853 14510 15004 20340 13908 ...
##  $ ghealth: num  7274 6124 6129 11925 7219 ...
```

2.7.2 Referencing Data Frames

Throughout this module, you will need to refer to particular parts of a dataframe - perhaps a particular column (an area attribute); or a particular subset of respondents. Hence it is worth spending some time now mastering this particular skill.

The relevant R function, `[ ]`, has the format `[row,col]` or, more generally, `[set of rows, set of cols]`.

Run the following commands to get a feel of how to extract different slices of the data:

```
df # whole data.frame
df[1, 1] # contents of first row and column
df[2, 2:3] # contents of the second row, second and third columns
df[1, ] # first row, ALL columns [the default if no columns specified]
df[ ,1:2] # ALL rows; first and second columns
df[c(1,3,5), ] # rows 1,3,5; ALL columns
```

```
df[ , 2] # ALL rows; second column (by default results containing only
              #one column are converted back into a vector)
df[ , 2, drop=FALSE] # ALL rows; second column (returned as a data.frame)
```

In the above, note that we have used two other R functions:

- `1:3` The colon operator tells R to produce a list of numbers including the named start and end points.
- `c(1,3,5)` Tells R to combine the contents within the brackets into one list of objects

Run both of these fuctions on their own to get a better understanding of what they do.

Three other methods for referencing the contents of a data.frame make direct use of the variable names within the data.frame, which tends to make for easier to read/understand code:

```
df[,"pop"] # variable name in quotes inside the square brackets
df$pop # variable name prefixed with $ and appended to the data.frame name
# or you can use attach
attach(df)
pop # but be careful if you already have an age variable in your local workspace
```

Want to check the variables available, use the `names( )`:

```
names(df)
```

```
## [1] "wards"   "pop"     "ghealth"
```

2.8 Read Data

Ensure your memory is clear

```
rm(list=ls()) # rm for targeted deletion / ls for listing all existing objects
```

There are many commands to read / load data onto R. The command to use will depend upon the format they have been saved. Normally they are saved in *csv* format from Excel or other software packages. So we use either:

- `df <- read.table("path/file_name.csv", header = FALSE, sep =",")`
- `df <- read("path/file_name.csv", header = FALSE)`
- `df <- read.csv2("path/file_name.csv", header = FALSE)`

To read files in other formats, refer to this useful DataCamp tutorial

```
census <- read.csv("data/census/census_data.csv")
head(census)
```

```
##        code                     ward pop16_74 higher_managerial   pop
## 1 E05000886 Allerton and Hunts Cross    10930              1103 14853
## 2 E05000887                  Anfield    10712               312 14510
## 3 E05000888                Belle Vale    10987               432 15004
## 4 E05000889                   Central    19174              1346 20340
## 5 E05000890                 Childwall    10410              1123 13908
## 6 E05000891                    Church    10569              1843 13974
##   ghealth
## 1    7274
## 2    6124
## 3    6129
## 4   11925
## 5    7219
## 6    7461
```

```
# NOTE: always ensure your are setting the correct directory leading to the data.
# It may differ from your existing working directory
```

2.8.1 Quickly inspect the data

1. What class?
2. What R data types?
3. What data types?

```
# 1
class(census)
# 2 & 3
str(census)
```

Just interested in the variable names:

```
names(census)
```

```
## [1] "code"              "ward"              "pop16_74"
## [4] "higher_managerial" "pop"               "ghealth"
```

or want to view the data:

```
View(census)
```

2.9 Manipulation Data

2.9.1 Adding New Variables

Usually you want to add / create new variables to your data frame using existing variables eg. computing percentages by dividing two variables. There are many ways in which you can do this i.e. referecing a data frame as we have done above, or using `$` (e.g. `census$pop`). For this module, we'll use `tidyverse`:

```
census <- census %>% mutate(per_ghealth = ghealth / pop)
```

Note we used a *pipe operator* `%>%`, which helps make the code more efficient and readable - more details, see Grolemund and Wickham (2019). When using the pipe operator, recall to first indicate the data frame before `%>%`.

Note also the use a variable name before the `=` sign in brackets to indicate the name of the new variable after `mutate`.

2.9.2 Selecting Variables

Usually you want to select a subset of variables for your analysis as storing to large data sets in your R memory can reduce the processing speed of your machine. A selection of data can be achieved by using the `select` function:

```
ndf <- census %>% select(ward, pop16_74, per_ghealth)
```

Again first indicate the data frame and then the variable you want to select to build a new data frame. Note the code chunk above has created a new data frame called `ndf`. Explore it.

2.9.3 Filtering Data

You may also want to filter values based on defined conditions. You may want to filter observations greater than a certain threshold or only areas within a certain region. For example, you may want to select areas with a percentage of good health population over 50%:

```
ndf2 <- census %>% filter(per_ghealth < 0.5)
```

You can use more than one variables to set conditions. Use "`,`" to add a condition.

2.9.4 Joining Data Drames

When working with spatial data, we often need to join data. To this end, you need a common unique `id variable`. Let's say, we want to add a data frame containing census data on households for Liverpool, and join the new attributes to one of the existing data frames in the workspace. First we will read the data frame we want to join (ie. `census_data2.csv`).

```
# read data
census2 <- read.csv("data/census/census_data2.csv")
# visualise data structure
str(census2)
```

```
## 'data.frame':    30 obs. of  3 variables:
##  $ geo_code              : Factor w/ 30 levels "E05000886","E05000887",..: 1 2 3 4 5 6 7 8 9
##  $ households            : int  6359 6622 6622 7139 5391 5884 6576 6745 6317 6024 ...
##  $ socialrented_households: int  827 1508 2818 1311 374 178 2859 1564 1023 1558 ...
```

The variable `geo_code` in this data frame corresponds to the `code` in the existing data frame and they are unique so they can be automatically matched by using the `merge()` function. The `merge()` function uses two arguments: `x` and `y`. The former refers to data frame 1 and the latter to data frame 2. Both of these two data frames must have a `id` variable containing the same information. Note they can have different names. Another key argument to include is `all.x=TRUE` which tells the function to keep all the records in `x`, but only those in `y` that match in case there are discrepancies in the `id` variable.

```
# join data frames
join_dfs <- merge(census, census2, by.x="code", by.y="geo_code", all.x = TRUE)
# check data
head(join_dfs)
```

```
##        code                      ward pop16_74 higher_managerial   pop
## 1 E05000886 Allerton and Hunts Cross    10930              1103 14853
## 2 E05000887                  Anfield    10712               312 14510
## 3 E05000888               Belle Vale    10987               432 15004
## 4 E05000889                  Central    19174              1346 20340
## 5 E05000890                Childwall    10410              1123 13908
## 6 E05000891                   Church    10569              1843 13974
##   ghealth per_ghealth households socialrented_households
## 1    7274   0.4897327       6359                     827
## 2    6124   0.4220538       6622                    1508
## 3    6129   0.4084911       6622                    2818
## 4   11925   0.5862832       7139                    1311
## 5    7219   0.5190538       5391                     374
## 6    7461   0.5339201       5884                     178
```

2.9.5 Saving Data

It may also be convinient to save your R projects. They contains all the objects that you have created in your workspace by using the `save.image( )` function:

```
save.image("week1_envs453.RData")
```

This creates a file labelled "week1_envs453.RData" in your working directory. You can load this at a later stage using the `load( )` function.

```
load("week1_envs453.RData")
```

Alternatively you can save / export your data into a `csv` file. The first argument in the function is the object name, and the second: the name of the csv we want to create.

```
write.csv(join_dfs, "join_censusdfs.csv")
```

2.10 Using Spatial Data Frames

A core area of this module is learning to work with spatial data in R. R has various purposedly designed **packages** for manipulation of spatial data and spatial analysis techniques. Various R packages exist in CRAN eg. `spatial`, `sgeostat`, `splancs`, `maptools`, `tmap`, `rgdal`, `spand` and more recent development of `sf` - see Lovelace et al. (2020) for a great description and historical context for some of these packages.

During this session, we will use `sf`.

We first need to import our spatial data. We will use a shapefile containing data at Output Area (OA) level for Liverpool. These data illustrates the hierarchical structure of spatial data.

2.10.1 Read Spatial Data

```
oa_shp <- st_read("data/census/Liverpool_OA.shp")
```

```
## Reading layer `Liverpool_OA' from data source `/home/jovyan/work/data/census/Liverpo
## Simple feature collection with 1584 features and 18 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: 332390.2 ymin: 379748.5 xmax: 345636 ymax: 397980.1
## epsg (SRID):    NA
## proj4string:    +proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=400000 +y_0=-1
```

Examine the input data. A spatial data frame stores a range of attributes derived from a shapefile including the **geometry** of features (e.g. polygon shape and location), **attributes** for each feature (stored in the .dbf), projection and coordinates of the shapefile's bounding box - for details, execute:

```
?st_read
```

You can employ the usual functions to visualise the content of the created data frame:

```
# visualise variable names
names(oa_shp)
```

```
##  [1] "OA_CD"    "LSOA_CD"  "MSOA_CD"  "LAD_CD"   "pop"      "H_Vbad"
##  [7] "H_bad"    "H_fair"   "H_good"   "H_Vgood"  "age_men"  "age_med"
## [13] "age_60"   "S_Rent"   "Ethnic"   "illness"  "unemp"    "males"
## [19] "geometry"
```

```
# data structure
str(oa_shp)
```

```
## Classes 'sf' and 'data.frame':   1584 obs. of  19 variables:
##  $ OA_CD   : Factor w/ 1584 levels "E00032987","E00032988",..: 1547 485 143 1567 980 1186 1122
##  $ LSOA_CD : Factor w/ 298 levels "E01006512","E01006513",..: 291 96 33 124 187 231 220 175 17
##  $ MSOA_CD : Factor w/ 61 levels "E02001347","E02001348",..: 59 12 19 23 29 20 31 14 30 10 ...
##  $ LAD_CD  : Factor w/ 1 level "E08000012": 1 1 1 1 1 1 1 1 1 1 ...
##  $ pop     : int  185 281 208 200 321 187 395 320 316 214 ...
##  $ H_Vbad  : int  1 2 3 7 4 4 5 9 5 4 ...
##  $ H_bad   : int  2 20 10 8 10 25 19 22 25 17 ...
##  $ H_fair  : int  9 47 22 17 32 70 42 53 55 39 ...
##  $ H_good  : int  53 111 71 52 112 57 131 104 104 53 ...
##  $ H_Vgood : int  120 101 102 116 163 31 198 132 127 101 ...
##  $ age_men : num  27.9 37.7 37.1 33.7 34.2 ...
##  $ age_med : num  25 36 32 29 34 53 23 30 34 29 ...
##  $ age_60  : num  0.0108 0.1637 0.1971 0.1 0.1402 ...
##  $ S_Rent  : num  0.0526 0.176 0.0235 0.2222 0.0222 ...
##  $ Ethnic  : num  0.3514 0.0463 0.0192 0.215 0.0779 ...
##  $ illness : int  185 281 208 200 321 187 395 320 316 214 ...
##  $ unemp   : num  0.0438 0.121 0.1121 0.036 0.0743 ...
##  $ males   : int  122 128 95 120 158 123 207 164 157 94 ...
##  $ geometry:sfc_MULTIPOLYGON of length 1584; first list element: List of 1
##   ..$ :List of 1
##   .. ..$ : num [1:14, 1:2] 335106 335130 335164 335173 335185 ...
##   ..- attr(*, "class")= chr  "XY" "MULTIPOLYGON" "sfg"
##  - attr(*, "sf_column")= chr "geometry"
##  - attr(*, "agr")= Factor w/ 3 levels "constant","aggregate",..: NA NA NA NA NA NA NA NA NA NA
##   ..- attr(*, "names")= chr  "OA_CD" "LSOA_CD" "MSOA_CD" "LAD_CD" ...
```

```
# see first few observations
head(oa_shp)
```

```
## Simple feature collection with 6 features and 18 fields
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: 335071.6 ymin: 389876.7 xmax: 339426.9 ymax: 394479
## epsg (SRID):    NA
## proj4string:    +proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=400000 +y_0=-1
##       OA_CD   LSOA_CD   MSOA_CD    LAD_CD pop H_Vbad H_bad H_fair H_good
## 1 E00176737 E01033761 E02006932 E08000012 185      1     2      9     53
## 2 E00033515 E01006614 E02001358 E08000012 281      2    20     47    111
## 3 E00033141 E01006546 E02001365 E08000012 208      3    10     22     71
## 4 E00176757 E01006646 E02001369 E08000012 200      7     8     17     52
## 5 E00034050 E01006712 E02001375 E08000012 321      4    10     32    112
## 6 E00034280 E01006761 E02001366 E08000012 187      4    25     70     57
##   H_Vgood  age_men age_med     age_60     S_Rent     Ethnic illness
## 1     120 27.94054      25 0.01081081 0.05263158 0.35135135     185
## 2     101 37.71174      36 0.16370107 0.17600000 0.04626335     281
## 3     102 37.08173      32 0.19711538 0.02352941 0.01923077     208
## 4     116 33.73000      29 0.10000000 0.22222222 0.21500000     200
## 5     163 34.19003      34 0.14018692 0.02222222 0.07788162     321
## 6      31 56.09091      53 0.44919786 0.88524590 0.11764706     187
##        unemp males                       geometry
## 1 0.04379562   122 MULTIPOLYGON (((335106.3 38...
## 2 0.12101911   128 MULTIPOLYGON (((335810.5 39...
## 3 0.11214953    95 MULTIPOLYGON (((336738 3931...
## 4 0.03597122   120 MULTIPOLYGON (((335914.5 39...
## 5 0.07428571   158 MULTIPOLYGON (((339325 3914...
## 6 0.44615385   123 MULTIPOLYGON (((338198.1 39...
```

TASK:

- What are the geographical hierarchy in these data?
- What is the smallest geography?
- What is the largest geography?

2.10.2 Basic Mapping

Again, many functions exist in CRAN for creating maps:

- `plot` to create static maps
- `tmap` to create static and interactive maps
- `leaflet` to create interactive maps
- `mapview` to create interactive maps
- `ggplot2` to create data visualisations, including static maps

Figure 2.5: OAs of Livepool

- `shiny` to create web applications, including maps

Here this notebook demonstrates the use of `plot` and `tmap`. First `plot` is used to map the spatial distribution of non-British-born population in Liverpool. First we only map the geometries on the right,

2.10.2.1 Using `plot`

```
# mapping geometry
plot(st_geometry(oa_shp))
```

and then:

```
# map attributes, adding intervals
plot(oa_shp["Ethnic"], key.pos = 4, axes = TRUE, key.width = lcm(1.3), key.length = 1.,
     breaks = "jenks", lwd = 0.1, border = 'grey')
```

TASK:

- What is the key pattern emerging from this map?

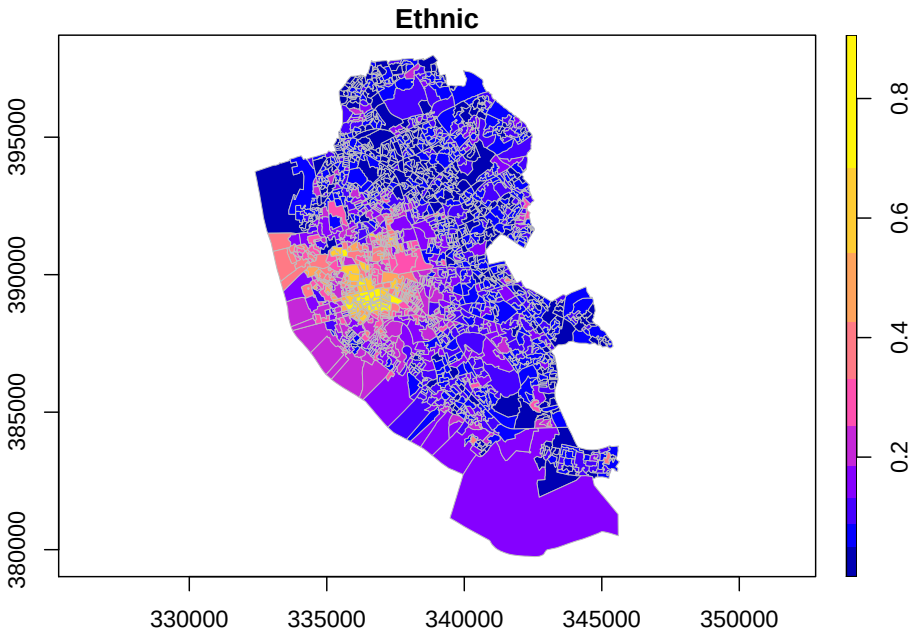


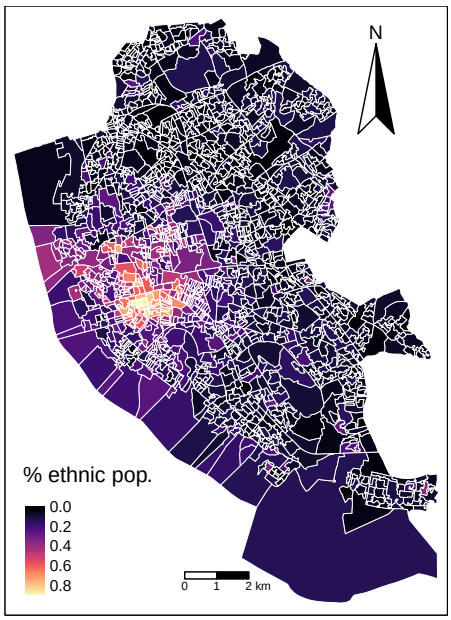
Figure 2.6: Spatial distribution of ethnic groups, Liverpool

2.10.2.2 Using `tmap`

Similar to `ggplot2`, `tmap` is based on the idea of a 'grammar of graphics' which involves a separation between the input data and aesthetics (i.e. the way data are visualised). Each data set can be mapped in various different ways, including location as defined by its geometry, colour and other features. The basic building block is `tm_shape()` (which defines input data), followed by one or more layer elements such as `tm_fill()` and `tm_dots()`.

```
# ensure geometry is valid
oa_shp = lwgeom::st_make_valid(oa_shp)

# map
legend_title = expression("% ethnic pop.")
map_oa = tm_shape(oa_shp) +
  tm_fill(col = "Ethnic", title = legend_title, palette = magma(256), style = "cont") +
  tm_borders(col = "white", lwd = .01)  + # add borders
  tm_compass(type = "arrow", position = c("right", "top") , size = 4) + # add compass
  tm_scale_bar(breaks = c(0,1,2), text.size = 0.5, position =  c("center", "bottom")) +
map_oa
```



Note that the operation + is used to add new layers. You can set style themes by `tm_style`. To visualise the existing styles use `tmap_style_catalogue()`, and you can also evaluate the code chunk below if you would like to create an interactive map.

```
tmap_mode("view")
map_oa
```

TASK:

- Try mapping other variables in the spatial data frame. Where do population aged 60 and over concentrate?

2.10.3 Comparing geographies

If you recall, one of the key issues of working with spatial data is the modifiable area unit problem (MAUP) - see lecture notes. To get a sense of the effects of MAUP, we analyse differences in the spatial patterns of the ethnic population in Liverpool between Middle Layer Super Output Areas (MSOAs) and OAs. So we map these geographies together.

```
# read data at the msoa level
msoa_shp <- st_read("data/census/Liverpool_MSOA.shp")
```

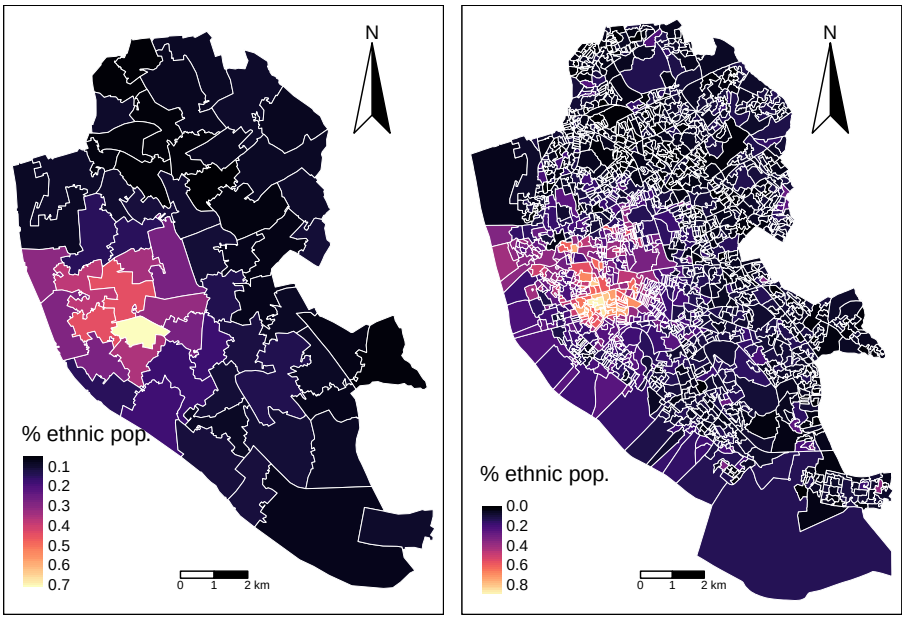
```
## Reading layer `Liverpool_MSOA' from data source `/home/jovyan/work/data/census/Liverpool_MSOA.
## Simple feature collection with 61 features and 16 fields
```

```
## geometry type:  MULTIPOLYGON
## dimension:      XY
## bbox:           xmin: 333086.1 ymin: 381426.3 xmax: 345636 ymax: 397980.1
## epsg (SRID):    NA
## proj4string:    +proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=400000 +y_0=-10
```

```r
# ensure geometry is valid
msoa_shp = lwgeom::st_make_valid(msoa_shp)

# create a map
map_msoa = tm_shape(msoa_shp) +
  tm_fill(col = "Ethnic", title = legend_title, palette = magma(256), style = "cont") +
  tm_borders(col = "white", lwd = .01)  +
  tm_compass(type = "arrow", position = c("right", "top") , size = 4) +
  tm_scale_bar(breaks = c(0,1,2), text.size = 0.5, position =  c("center", "bottom"))

# arrange maps
tmap_arrange(map_msoa, map_oa)
```



TASK:

- What differences do you see between OAs and MSOAs?
- Can you identify areas of spatial clustering? Where are they?

2.11 Useful Functions

Function	Description
read.csv()	read csv files into data frames
str()	inspect data structure
mutate()	create a new variable
filter()	filter observations based on variable values
%>%	pipe operator - chain operations
select()	select variables
merge()	join dat frames
st_read	read spatial data (ie. shapefiles)
plot()	create a map based a spatial data set
tm_shape(), tm_fill(), tm_borders()	create a map using tmap functions
tm_arrange	display multiple maps in a single “metaplot”

Chapter 3

Points

This chapter[1] is based on the following references, which are great follow-up's on the topic:

- Lovelace and Cheshire (2014) is a great introduction.
- Chapter 6 of Brunsdon and Comber (2015), in particular subsections 6.3 and 6.7.
- Bivand et al. (2013) provides an in-depth treatment of spatial data in R.

This chapter is part of Spatial Analysis Notes, a compilation hosted as a GitHub repository that you can access it in a few ways:

- As a download of a `.zip` file that contains all the materials.
- As an html website.
- As a pdf document
- As a GitHub repository.

3.1 Dependencies

This tutorial relies on the following libraries that you will need to have installed on your machine to be able to interactively follow along[2]. Once installed, load them up with the following commands:

```
# Layout
library(tufte)
```

[1] This chapter is part of Spatial Analysis Notes Points – Kernel Density Estimation and Spatial interpolation by Dani Arribas-Bel is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

[2] You can install package `mypackage` by running the command `install.packages("mypackage")` on the R prompt or through the `Tools --> Install Packages...` menu in RStudio.

```
# For pretty table
library(knitr)
# Spatial Data management
library(rgdal)
```

```
## Loading required package: sp
```

```
## rgdal: version: 1.4-4, (SVN revision 833)
##  Geospatial Data Abstraction Library extensions to R successfully loaded
##  Loaded GDAL runtime: GDAL 2.2.3, released 2017/11/20
##  Path to GDAL shared files: /usr/share/gdal/2.2
##  GDAL binary built with GEOS: TRUE
##  Loaded PROJ.4 runtime: Rel. 4.9.3, 15 August 2016, [PJ_VERSION: 493]
##  Path to PROJ.4 shared files: (autodetected)
##  Linking to sp version: 1.3-1
```

```
# Pretty graphics
library(ggplot2)
# Thematic maps
library(tmap)
# Pretty maps
library(ggmap)
```

```
## Google's Terms of Service: https://cloud.google.com/maps-platform/terms/.
```

```
## Please cite ggmap if you use it! See citation("ggmap") for details.
```

```
# Various GIS utilities
library(GISTools)
```

```
## Loading required package: maptools
```

```
## Checking rgeos availability: TRUE
```

```
## Loading required package: RColorBrewer
```

```
## Loading required package: MASS
```

```
## Loading required package: rgeos
```

```
## rgeos version: 0.5-1, (SVN revision 614)
##  GEOS runtime version: 3.6.2-CAPI-1.10.2
##  Linking to sp version: 1.3-1
##  Polygon checking: TRUE
```

```
# For all your interpolation needs
library(gstat)
```

```
## Registered S3 method overwritten by 'xts':
##   method     from
##   as.zoo.xts zoo
```

```
# For data manipulation
library(plyr)
```

Before we start any analysis, let us set the path to the directory where we are working. We can easily do that with `setwd()`. Please replace in the following line the path to the folder where you have placed this file -and where the `house_transactions` folder with the data lives.

```
#setwd('/media/dani/baul/AAA/Documents/teaching/u-lvl/2016/envs453/code')
setwd('.')
```

3.2 Data

For this session, we will use a subset of residential property transaction data for the city of Liverpool. These are provided by the Land Registry (as part of their Price Paid Data) but have been cleaned and re-packaged by Dani Arribas-Bel.

Let us start by reading the data, which comes in a shapefile:

```
db <- readOGR(dsn = 'data/house_transactions', layer = 'liv_house_trans')
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "/home/jovyan/work/data/house_transactions", layer: "liv_house_trans"
## with 6324 features
## It has 18 fields
## Integer64 fields read as strings:  price
```

Before we forget, let us make sure `price` is considered a number, not a factor:

```
db@data$price <- as.numeric(as.character((db@data$price)))
```

The dataset spans the year 2014:

```
# Format dates
dts <- as.Date(db@data$trans_date)
# Set up summary table
tab <- summary(dts)
tab
```

```
##         Min.      1st Qu.       Median         Mean      3rd Qu.
## "2014-01-02" "2014-04-11" "2014-07-09" "2014-07-08" "2014-10-03"
##         Max.
## "2014-12-30"
```

We can then examine the elements of the object with the `summary` method:

```
summary(db)
```

```
## Object of class SpatialPointsDataFrame
```

```
## Coordinates:
##              min    max
## coords.x1 333536 345449
## coords.x2 382684 397833
## Is projected: TRUE
## proj4string :
## [+proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=400000
## +y_0=-100000 +datum=OSGB36 +units=m +no_defs +ellps=airy
## +towgs84=446.448,-125.157,542.060,0.1502,0.2470,0.8421,-20.4894]
## Number of points: 6324
## Data attributes:
##       pcds                                          id
##  L1 6LS : 126   {00029226-80EF-4280-9809-109B8509656A}:   1
##  L8 5TE :  63   {00041BD2-4A07-4D41-A5AE-6459CD5FD37C}:   1
##  L1 5AQ :  34   {0005AE67-9150-41D4-8D56-6BFC868EECA3}:   1
##  L24 1WA:  31   {00183CD7-EE48-434B-8A1A-C94B30A93691}:   1
##  L17 6BT:  26   {003EA3A5-F804-458D-A66F-447E27569456}:   1
##  L3 1EE :  24   {00411304-DD5B-4F11-9748-93789D6A000E}:   1
##  (Other):6020   (Other)                                :6318
##      price                    trans_date   type     new      duration
##  Min.   :    1000   2014-06-27 00:00: 109   D: 505   N:5495   F:3927
##  1st Qu.:   70000   2014-12-19 00:00: 109   F:1371   Y: 829   L:2397
##  Median :  110000   2014-02-28 00:00: 105   O: 119
##  Mean   :  144310   2014-10-31 00:00:  95   S:1478
##  3rd Qu.:  160000   2014-03-28 00:00:  94   T:2851
##  Max.   :26615720   2014-11-28 00:00:  94
##                     (Other)         :5718
##       paon               saon                  street
##  3      : 203   FLAT 2     :  25   CROSSHALL STREET: 133
##  11     : 151   FLAT 3     :  25   STANHOPE STREET :  63
##  14     : 148   FLAT 1     :  24   PALL MALL       :  47
##  5      : 146   APARTMENT 4:  23   DUKE STREET     :  41
##  4      : 140   APARTMENT 2:  21   MANN ISLAND     :  41
##  8      : 128   (Other)    : 893   OLD HALL STREET :  39
##  (Other):5408   NA's       :5313   (Other)         :5960
##           locality           town           district           county
##  WAVERTREE   : 126   LIVERPOOL:6324   KNOWSLEY :  12   MERSEYSIDE:6324
##  MOSSLEY HILL: 102                    LIVERPOOL:6311
##  WALTON      :  88                    WIRRAL   :   1
##  WEST DERBY  :  71
##  WOOLTON     :  66
##  (Other)     : 548
##  NA's        :5323
##  ppd_cat  status         lsoa11          LSOA11CD
##  A:5393   A:6324   E01033762: 144   E01033762: 144
##  B: 931            E01033756:  98   E01033756:  98
```

```
##                    E01033752:  93   E01033752:  93
##                    E01033750:  71   E01033750:  71
##                    E01006518:  68   E01006518:  68
##                    E01033755:  65   E01033755:  65
##                    (Other)   :5785   (Other)   :5785
```

See how it contains several pieces, some relating to the spatial information, some relating to the tabular data attached to it. We can access each of the separately if we need it. For example, to pull out the names of the columns in the `data.frame`, we can use the `@data` appendix:

```
colnames(db@data)
```

```
##  [1] "pcds"       "id"         "price"      "trans_date" "type"
##  [6] "new"        "duration"   "paon"       "saon"       "street"
## [11] "locality"   "town"       "district"   "county"     "ppd_cat"
## [16] "status"     "lsoa11"     "LSOA11CD"
```

The rest of this session will focus on two main elements of the shapefile: the spatial dimension (as stored in the point coordinates), and the house price values contained in the `price` column. To get a sense of what they look like first, let us plot both. We can get a quick look at the non-spatial distribution of house values with the following commands:

```
# Create the histogram
hist <- qplot(data=db@data,x=price)
hist
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

This basically shows there is a lot of values concentrated around the lower end of the distribution but a few very large ones. A usual transformation to *shrink* these differences is to take logarithms:

```
# Create log and add it to the table
logpr <- log(as.numeric(db@data$price))
db@data['logpr'] <- logpr
# Create the histogram
hist <- qplot(x=logpr)
hist
```

```
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

To obtain the spatial distribution of these houses, we need to turn away from the `@data` component of `db`. The easiest, quickest (and also dirtiest) way to get a sense of what the data look like over space is using `plot`:

```
plot(db)
```

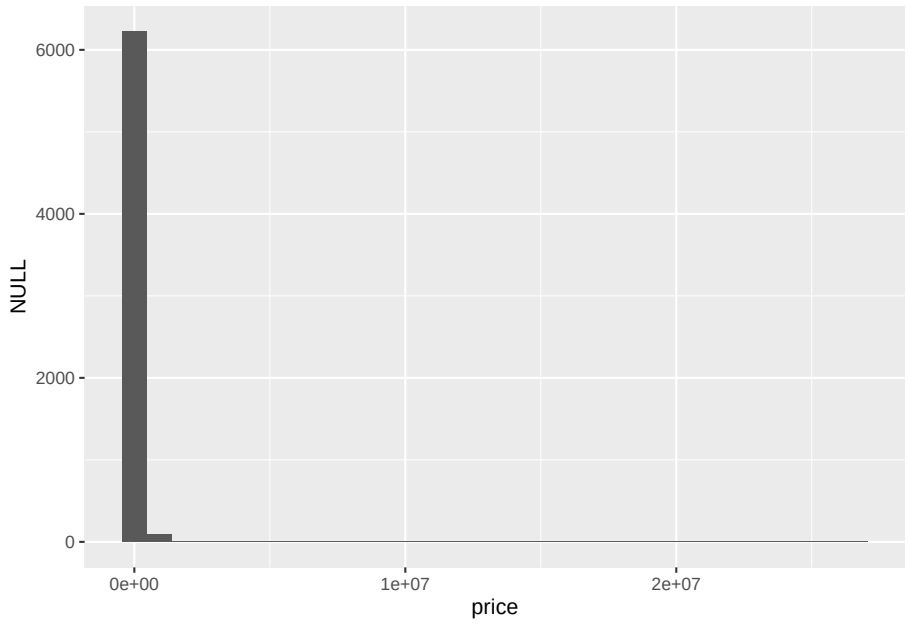


Figure 3.1: Raw house prices in Liverpool

3.3 KDE

Kernel Density Estimation (KDE) is a technique that creates a *continuous* representation of the distribution of a given variable, such as house prices. Although theoretically it can be applied to any dimension, usually, KDE is applied to either one or two dimensions.

3.3.1 One-dimensional KDE

KDE over a single dimension is essentially a contiguous version of a histogram. We can see that by overlaying a KDE on top of the histogram of logs that we have created before:

```
# Create the base
base <- ggplot(db@data, aes(x=logpr))
# Histogram
hist <- base +
  geom_histogram(bins=50, aes(y=..density..))
# Overlay density plot
kde <- hist +
```

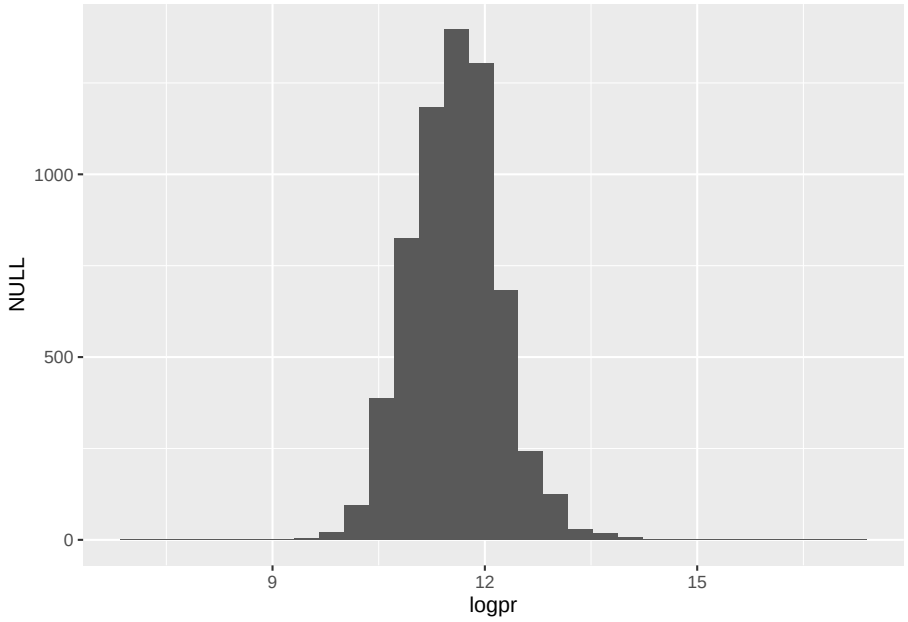


Figure 3.2: Log of house price in Liverpool

```
  geom_density(fill="#FF6666", alpha=0.5, colour="#FF6666")
kde
```

The key idea is that we are smoothing out the discrete binning that the histogram involves. Note how the histogram is exactly the same as above shapewise, but it has been rescalend on the Y axis to reflect probabilities rather than counts.

3.3.2 Two-dimensional (spatial) KDE

Geography, at the end of the day, is usually represented as a two-dimensional space where we locate objects using a system of dual coordinates, `X` and `Y` (or latitude and longitude). Thanks to that, we can use the same technique as above to obtain a smooth representation of the distribution of a two-dimensional variable. The crucial difference is that, instead of obtaining a curve as the output, we will create a *surface*, where intensity will be represented with a color gradient, rather than with the second dimension, as it is the case in the figure above.

To create a spatial KDE in R, there are several ways. If you do not want to necessarily acknowledge the spatial nature of your data, or you they are not

Figure 3.3: Spatial distribution of house transactions in Liverpool

stored in a spatial format, you can plot them using `ggplot2`. Note we first need to convert the coordinates (stored in the spatial part of `db`) into columns of X and Y coordinates, then we can plot them:

```
# Attach XY coordinates
db@data['X'] <- db@coords[, 1]
db@data['Y'] <- db@coords[, 2]
# Set up base layer
base <- ggplot(data=db@data, aes(x=X, y=Y))
# Create the KDE surface
kde <- base + stat_density2d(aes(x = X, y = Y, alpha = ..level..),
               size = 0.01, bins = 16, geom = 'polygon') +
            scale_fill_gradient()
kde
```

Or, we can use a package such as the `GISTools`, which allows to pass a spatial object directly:

```
# Compute the KDE
kde <- kde.points(db)
# Plot the KDE
level.plot(kde)
```

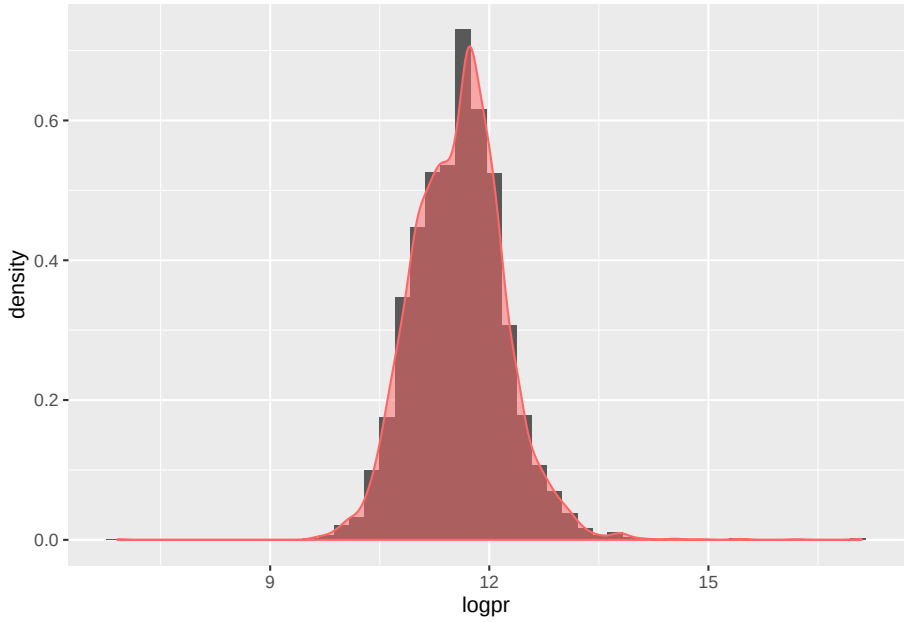


Figure 3.4: Histogram and KDE of the log of house prices in Liverpool

Either of these approaches generate a surface that represents the density of dots, that is an estimation of the probability of finding a house transaction at a given coordinate. However, without any further information, they are hard to interpret and link with previous knowledge of the area. To bring such context to the figure, we can plot an underlying basemap, using a cloud provider such as Google Maps or, as in this case, OpenStreetMap. To do it, we will leverage the library `ggmap`, which is designed to play nicely with the `ggplot2` family (hence the seemingly counterintuitive example above). Before we can plot them with the online map, we need to reproject them though.

```
# Reproject coordinates
wgs84 <- CRS("+proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0")
db_wgs84 <- spTransform(db, wgs84)
db_wgs84@data['lon'] <- db_wgs84@coords[, 1]
db_wgs84@data['lat'] <- db_wgs84@coords[, 2]
xys <- db_wgs84@data[c('lon', 'lat')]
# Bounding box
liv <- c(left = min(xys$lon), bottom = min(xys$lat),
         right = max(xys$lon), top = max(xys$lat))
# Download map tiles
basemap <- get_stamenmap(liv, zoom = 12,
```

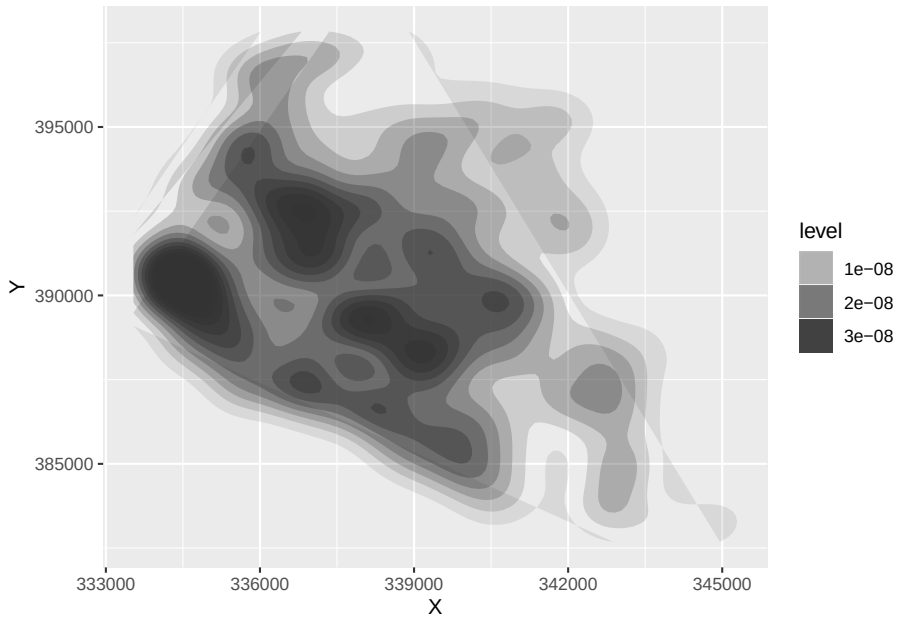


Figure 3.5: KDE of house transactions in Liverpool

```
                          maptype = "toner-lite")
```

```
## Source : http://tile.stamen.com/toner-lite/12/2013/1325.png
## Source : http://tile.stamen.com/toner-lite/12/2014/1325.png
## Source : http://tile.stamen.com/toner-lite/12/2015/1325.png
## Source : http://tile.stamen.com/toner-lite/12/2013/1326.png
## Source : http://tile.stamen.com/toner-lite/12/2014/1326.png
## Source : http://tile.stamen.com/toner-lite/12/2015/1326.png
## Source : http://tile.stamen.com/toner-lite/12/2013/1327.png
## Source : http://tile.stamen.com/toner-lite/12/2014/1327.png
## Source : http://tile.stamen.com/toner-lite/12/2015/1327.png
```

```
# Overlay KDE
final <- ggmap(basemap, extent = "device",
               maprange=FALSE) +
  stat_density2d(data = db_wgs84@data,
                 aes(x = lon, y = lat,
```

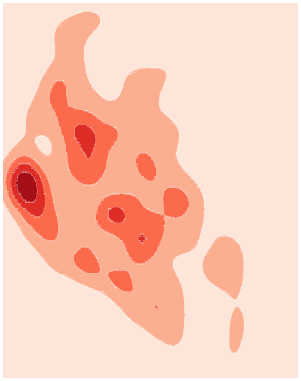


Figure 3.6: KDE of house transactions in Liverpool

```
                    alpha=..level..,
                    fill = ..level..),
                size = 0.01, bins = 16,
                geom = 'polygon',
                show.legend = FALSE) +
  scale_fill_gradient2("Transaction\nDensity",
                       low = "#fffff8",
                       high = "#8da0cb")
final
```

The plot above[3] allows us to not only see the distribution of house transactions, but to relate it to what we know about Liverpool, allowing us to establish many more connections than we were previously able. Mainly, we can easily see that the area with a highest volume of houses being sold is the city centre, with a "hole" around it that displays very few to no transactions and then several pockets further away.

[3] **EXERCISE** The map above uses the Stamen map `toner-lite`. Explore additional tile styles on their website and try to recreate the plot above.

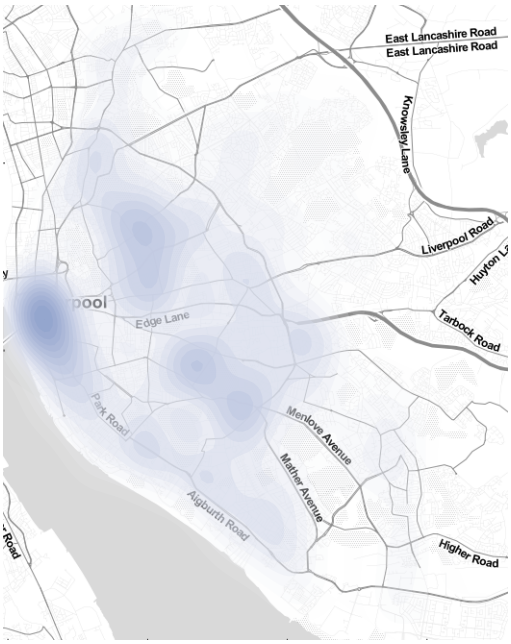


Figure 3.7: KDE of house transactions in Liverpool

3.4 Spatial Interpolation

The previous section demonstrates how to visualize the distribution of a set of spatial objects represented as points. In particular, given a bunch of house transactions, it shows how one can effectively visualize their distribution over space and get a sense of the density of occurrences. Such visualization, because it is based on KDE, is based on a smooth continuum, rather than on a discrete approach (as a choropleth would do, for example).

Many times however, we are not particularly interested in learning about the density of occurrences, but about the distribution of a given value attached to each location. Think for example of weather stations and temperature: the location of the stations is no secret and rarely changes, so it is not of particular interest to visualize the density of stations; what we are usually interested instead is to know how temperature is distributed over space, given we only measure it in a few places. One could argue the example we have been working with so far, house price transactions, fits into this category as well: although where house are sold may be of relevance, more often we are interested in finding out what the "surface of price" looks like. Rather than *where are most houses being sold?* we usually want to know *where the most expensive or most affordable* houses are located.

In cases where we are interested in creating a surface of a given value, rather than a simple density surface of occurrences, KDE cannot help us. In these cases, what we are interested in is *spatial interpolation*, a family of techniques that aim at exactly that: creating continuous surfaces for a particular phenomenon (e.g. temperature, house prices) given only a finite sample of observations. Spatial interpolation is a large field of research that is still being actively developed and that can involve a substantial amount of mathematical complexity in order to obtain the most accurate estimates possible[4]. In this session, we will introduce the simplest possible way of interpolating values, hoping this will give you a general understanding of the methodology and, if you are interested, you can check out further literature. For example, Banerjee et al. (2014) or Cressie (2015) are hard but good overviews.

3.4.1 Inverse Distance Weight (IDW) interpolation

The technique we will cover here is called *Inverse Distance Weighting*, or IDW for convenience. Brunsdon and Comber (2015) offer a good description:

> In the *inverse distance weighting* (IDW) approach to interpolation, to estimate the value of z at location x a weighted mean of nearby observations is taken [...]. To accommodate the idea that observations of z at points closer to x should be given more importance in the interpolation, greater weight is given to these points [...]
>
> — Page 204

The math[5] is not particularly complicated and may be found in detail elsewhere (the reference above is a good starting point), so we will not spend too much time on it. More relevant in this context is the intuition behind. Essentially, the idea is that we will create a surface of house price by smoothing many values arranged along a regular grid and obtained by interpolating from the known locations to the regular grid locations. This will give us full and equal coverage to soundly perform the smoothing.

Enough chat, let's code.

From what we have just mentioned, there are a few steps to perform an IDW spatial interpolation:

1. Create a regular grid over the area where we have house transactions.

[4]There is also an important economic incentive to do this: some of the most popular applications are in the oil and gas or mining industries. In fact, the very creator of this technique, Danie G. Krige, was a mining engineer. His name is usually used to nickname spatial interpolation as *kriging*.

[5]Essentially, for any point x in space, the IDW estimate for value z is equivalent to $\hat{z}(x) = \frac{\sum_i w_i z_i}{\sum_i w_i}$ where i are the observations for which we do have a value, and w_i is a weight given to location i based on its distance to x.

2. Obtain IDW estimates for each point in the grid, based on the values of k nearest neighbors.
3. Plot a smoothed version of the grid, effectively representing the surface of house prices.

Let us go in detail into each of them[6]. First, let us set up a grid:

```
liv.grid <- spsample(db, type='regular', n=25000)
```

That's it, we're done! The function `spsample` hugely simplifies the task by taking a spatial object and returning the grid we need. Not a couple of additional arguments we pass: `type` allows us to get a set of points that are *uniformly* distributed over space, which is handy for the later smoothing; `n` controls how many points we want to create in that grid.

On to the IDW. Again, this is hugely simplified by `gstat`:

```
idw.hp <- idw(price ~ 1, locations=db, newdata=liv.grid)
```

```
## [inverse distance weighted interpolation]
```

Boom! We've got it. Let us pause for a second to see how we just did it. First, we pass `price ~ 1`. This specifies the formula we are using to model house prices. The name on the left of `~` represents the variable we want to explain, while everything to its right captures the *explanatory* variables. Since we are considering the simplest possible case, we do not have further variables to add, so we simply write `1`. Then we specify the original locations for which we do have house prices (our original `db` object), and the points where we want to interpolate the house prices (the `liv.grid` object we just created above). One more note: by default, `idw.hp` uses all the available observations, weighted by distance, to provide an estimate for a given point. If you want to modify that and restrict the maximum number of neighbors to consider, you need to tweak the argument `nmax`, as we do above by using the 150 neares observations to each point[7].

The object we get from `idw` is another spatial table, just as `db`, containing the interpolated values. As such, we can inspect it just as with any other of its kind. For example, to check out the top of the estimated table:

```
head(idw.hp@data)
```

```
##   var1.pred var1.var
## 1  158101.5       NA
## 2  158212.4       NA
## 3  158326.2       NA
## 4  158442.9       NA
## 5  158562.6       NA
```

[6] For the relevant calculations, we will be using the `gstat` library.
[7] Have a play with this because the results do change significantly. Can you reason why?

```
## 6  158685.6       NA
```

The column we will pay attention to is `var1.pred`. And to see the locations for which those correspond:

```
head(idw.hp@coords)
```

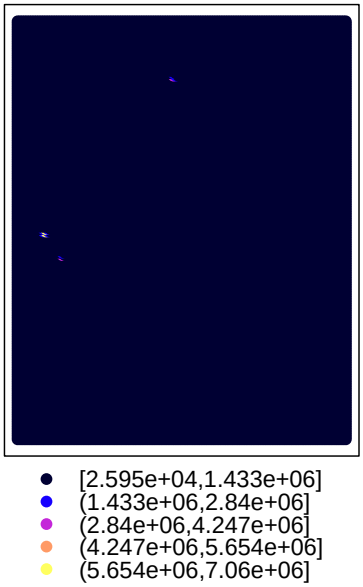
```
##            x1       x2
## [1,] 333598.3 382719.3
## [2,] 333683.3 382719.3
## [3,] 333768.2 382719.3
## [4,] 333853.2 382719.3
## [5,] 333938.2 382719.3
## [6,] 334023.1 382719.3
```

So, for a hypothetical house sold at the location in the first row of `idw.hp@coords` (expressed in the OSGB coordinate system), the price we would guess it would cost, based on the price of houses sold nearby, is the first element of column `var1.pred` in `idw.hp@data`.

3.4.2 A surface of housing prices

Once we have the IDW object computed, we can plot it to explore the distribution, not of house transactions in this case, but of house price over the geography of Liverpool. The easiest way to do this is by quickly calling the command `spplot`:

```
spplot(idw.hp['var1.pred'])
```



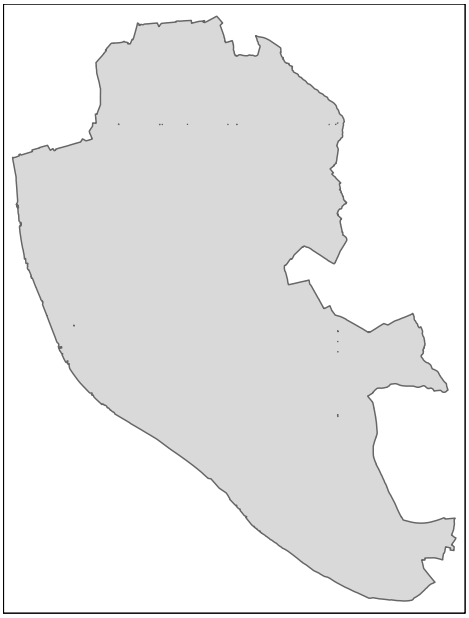
However, this is not entirely satisfactory for a number of reasons. Let us get an equivalen plot with the package `tmap`, which streamlines some of this and makes more aesthetically pleasant maps easier to build as it follows a "ggplot-y" approach.

```
# Load up the layer
liv.otl <- readOGR('data/house_transactions', 'liv_outline')
```

```
## OGR data source with driver: ESRI Shapefile
## Source: "/home/jovyan/work/data/house_transactions", layer: "liv_outline"
## with 1 features
## It has 1 fields
```

The shape we will overlay looks like this:

```
qtm(liv.otl)
```



Now let's give it a first go!

```
#
p = tm_shape(liv.otl) + tm_fill(col='black', alpha=1) +
  tm_shape(idw.hp) +
  tm_symbols(col='var1.pred', size=0.1, alpha=0.25,
             border.lwd=0., palette='YlGn')
p
```

The last two plots, however, are not really a surface, but a representation of the points we have just estimated. To create a surface, we need to do an interim transformation to convert the spatial object `idw.hp` into a table that a "surface plotter" can understand.

```
xyz <- data.frame(x=coordinates(idw.hp)[, 1],
                  y=coordinates(idw.hp)[, 2],
                  z=idw.hp$var1.pred)
```

Now we are ready to plot the surface as a contour:

```
base <- ggplot(data=xyz, aes(x=x, y=y))
surface <- base + geom_contour(aes(z=z))
surface
```

Which can also be shown as a filled contour:

```
base <- ggplot(data=xyz, aes(x=x, y=y))
surface <- base + geom_raster(aes(fill=z))
surface
```

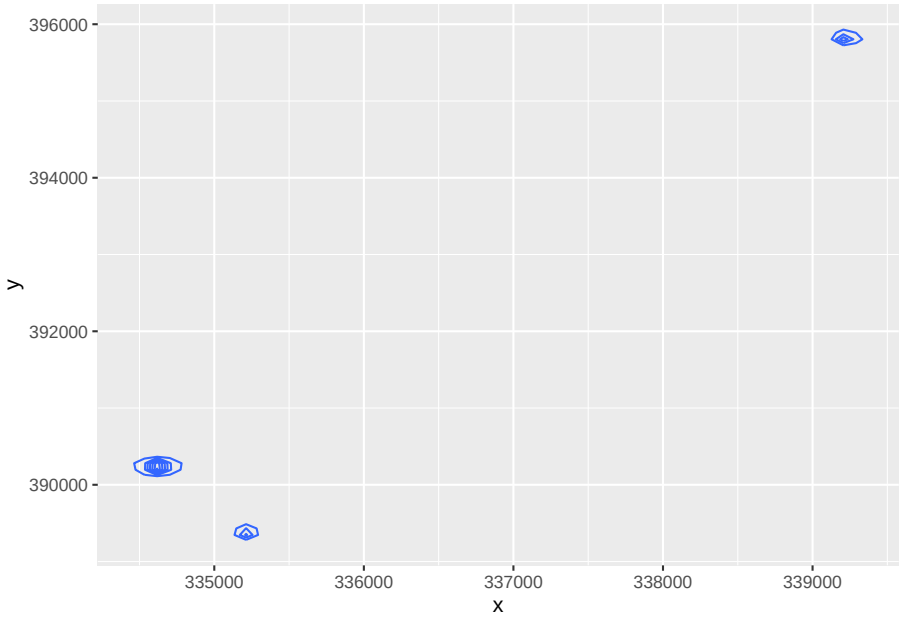
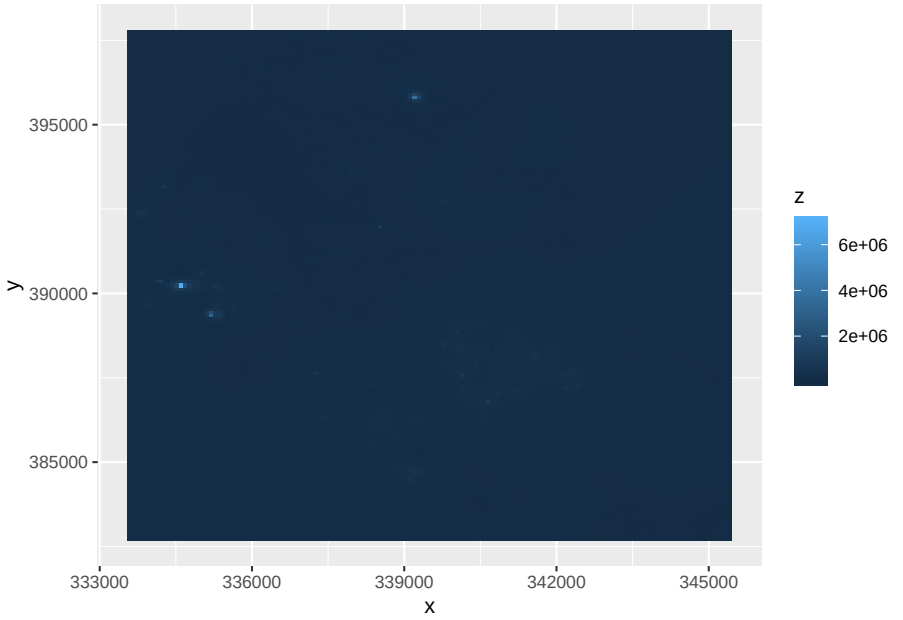


Figure 3.8: Contour of prices in Liverpool



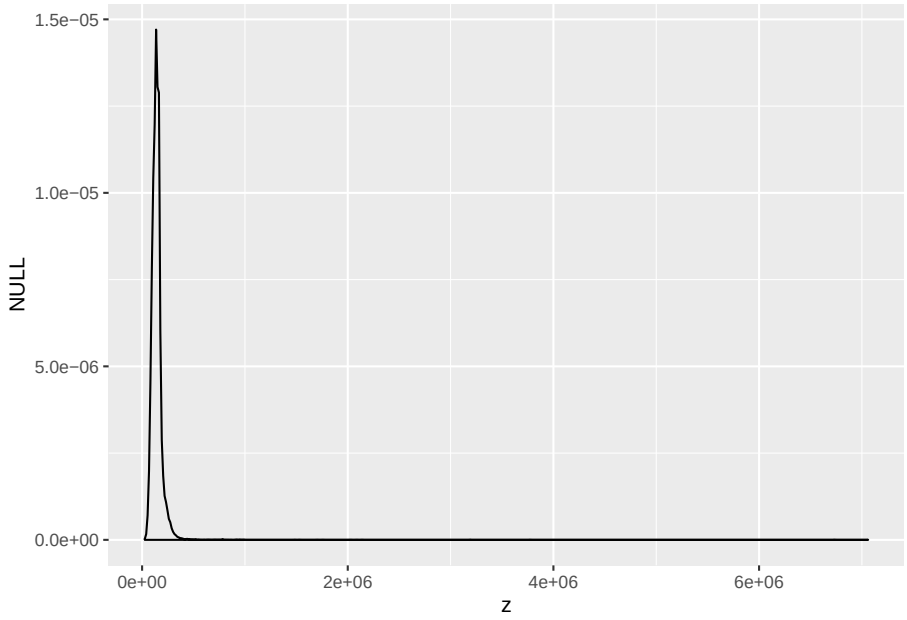


Figure 3.9: Skewness of prices in Liverpool

The problem here, when compared to the KDE above for example, is that a few values are extremely large:

```
qplot(data=xyz, x=z, geom='density')
```

Let us then take the logarithm before we plot the surface:

```
xyz['lz'] <- log(xyz$z)
base <- ggplot(data=xyz, aes(x=x, y=y))
surface <- base +
           geom_raster(aes(fill=lz),
                       show.legend = F)
surface
```

Now this looks better. We can start to tell some patterns. To bring in context, it would be great to be able to add a basemap layer, as we did for the KDE. This is conceptually very similar to what we did above, starting by reprojecting the points and continuing by overlaying them on top of the basemap. However, technically speaking it is not possible because `ggmap` –the library we have been using to display tiles from cloud providers– does not play well with our own rasters (i.e. the price surface). At the moment, it is surprisingly tricky to get this to work, so we will park it for now. However, developments such as the `sf`

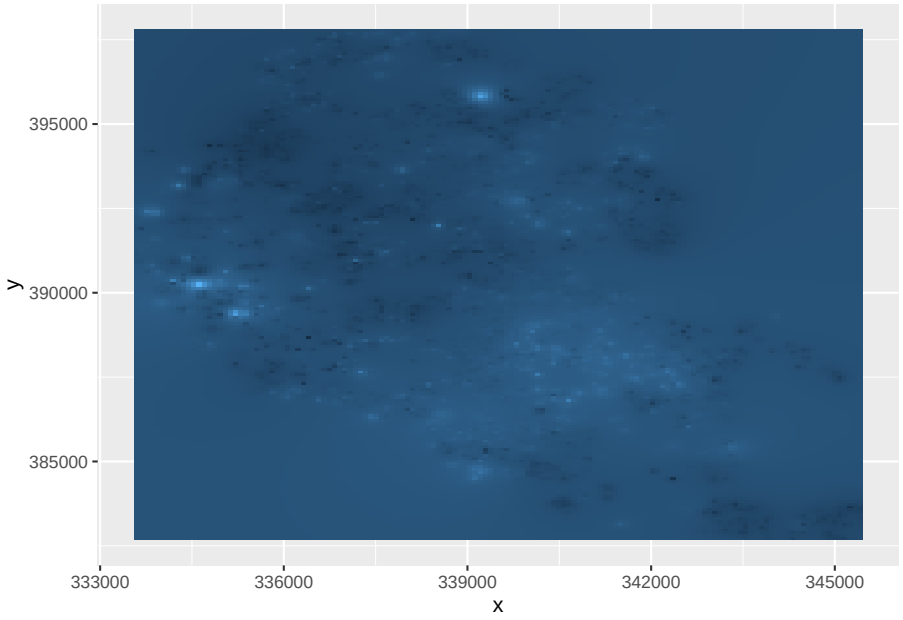


Figure 3.10: Surface of log-prices in Liverpool

project promise to make this easier in the future[8].

3.4.3 *"What should the next house's price be?"*

The last bit we will explore in this session relates to prediction for new values. Imagine you are a real state data scientist and your boss asks you to give an estimate of how much a new house going into the market should cost. The only information you have to make such a guess is the location of the house. In this case, the IDW model we have just fitted can help you. The trick is realizing that, instead of creating an entire grid, all we need is to obtain an estimate of a single location.

Let us say, the house is located on the coordinates `x=340000, y=390000` as expressed in the GB National Grid coordinate system. In that case, we can do as follows:

```
pt <- SpatialPoints(cbind(x=340000, y=390000),
                    proj4string = db@proj4string)
idw.one <- idw(price ~ 1, locations=db, newdata=pt)
```

[8] **BONUS** if you can figure out a way to do it yourself!

```
## [inverse distance weighted interpolation]
```

```
idw.one
```

```
## class       : SpatialPointsDataFrame
## features    : 1
## extent      : 340000, 340000, 390000, 390000  (xmin, xmax, ymin, ymax)
## crs         : +proj=tmerc +lat_0=49 +lon_0=-2 +k=0.9996012717 +x_0=400000 +y_0=-100
## variables   : 2
## names       :        var1.pred, var1.var
## value       : 157099.029513871,       NA
```

And, as show above, the estimated value is GBP157,099[9].

Using this predictive logic, and taking advantage of Google Maps and its geocoding capabilities, it is possible to devise a function that takes an arbitrary address in Liverpool and, based on the transactions occurred throughout 2014, provides an estimate of what the price for a property in that location could be.

```
how.much.is <- function(address, print.message=TRUE){
  # Convert the address into Lon/Lat coordinates
  # NOTE: this now requires an API key
  # https://github.com/dkahle/ggmap#google-maps-and-credentials
  ll.pt <- geocode(address)
  # Process as spatial table
  wgs84 <- CRS("+proj=longlat +datum=WGS84 +ellps=WGS84 +towgs84=0,0,0")
  ll.pt <- SpatialPoints(cbind(x=ll.pt$lon, y=ll.pt$lat),
                         proj4string = wgs84)
  # Transform Lon/Lat into OSGB
  pt <- spTransform(ll.pt, db@proj4string)
  # Obtain prediction
  idw.one <- idw(price ~ 1, locations=db, newdata=pt)
  price <- idw.one@data$var1.pred
  # Return predicted price
  if(print.message==T){
    writeLines(paste("\n\nBased on what surrounding properties were sold",
                     "for in 2014 a house located at", address, "would",
                     "cost",  paste("GBP", round(price), ".", sep=''), "\n\n"))
  }
  return(price)
}
```

Ready to test!

```
address <- "74 Bedford St S, Liverpool, L69 7ZT, UK"
#p <- how.much.is(address)
```

[9] **PRO QUESTION** Is that house expensive or cheap, as compared to the other houses sold in this dataset? Can you figure out where the house is?

Chapter 4

Flows

This chapter[1] covers spatial interaction flows. Using open data from the city of San Francisco about trips on its bikeshare system, we will estimate spatial interaction models that try to capture and explain the variation in the amount of trips on each given route. After visualizing the dataset, we begin with a very simple model and then build complexity progressively by augmenting it with more information, refined measurements, and better modeling approaches. Throughout the chapter, we explore different ways to grasp the predictive performance of each model. We finish with a prediction example that illustrates how these models can be deployed in a real-world application.

Content is based on the following references, which are great follow-up's on the topic:

- Singleton (2017), an online short course on R for Geographic Data Science and Urban Analytics. In particular, the section on mapping flows is specially relevant here.
- The predictive checks section draws heavily from Gelman and Hill (2006), in particular Chapters 6 and 7.

This tutorial is part of Spatial Analysis Notes, a compilation hosted as a GitHub repository that you can access in a few ways:

- As a download of a `.zip` file that contains all the materials.
- As an html website.
- As a pdf document
- As a GitHub repository.

[1] This chapter is part of Spatial Analysis Notes Flows – Exploring flows visually and through spatial interaction by Dani Arribas-Bel is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

Chapter 5

Dependencies

This tutorial relies on the following libraries that you will need to have installed on your machine to be able to interactively follow along[1]. Once installed, load them up with the following commands:

```
# Layout
library(tufte)
# Spatial Data management
library(rgdal)
```

```
## Loading required package: sp
```

```
## rgdal: version: 1.4-4, (SVN revision 833)
##  Geospatial Data Abstraction Library extensions to R successfully loaded
##  Loaded GDAL runtime: GDAL 2.2.3, released 2017/11/20
##  Path to GDAL shared files: /usr/share/gdal/2.2
##  GDAL binary built with GEOS: TRUE
##  Loaded PROJ.4 runtime: Rel. 4.9.3, 15 August 2016, [PJ_VERSION: 493]
##  Path to PROJ.4 shared files: (autodetected)
##  Linking to sp version: 1.3-1
```

```
# Pretty graphics
library(ggplot2)
# Thematic maps
library(tmap)
# Pretty maps
library(ggmap)
```

```
## Google's Terms of Service: https://cloud.google.com/maps-platform/terms/.
```

[1] You can install package `mypackage` by running the command `install.packages("mypackage")` on the R prompt or through the `Tools --> Install Packages...` menu in RStudio.

```
## Please cite ggmap if you use it! See citation("ggmap") for details.
```

```
# Simulation methods
library(arm)
```

```
## Loading required package: MASS
```

```
## Loading required package: Matrix
```

```
## Loading required package: lme4
```

```
##
## arm (Version 1.10-1, built: 2018-4-12)
```

```
## Working directory is /home/jovyan/work
```

Before we start any analysis, let us set the path to the directory where we are working. We can easily do that with `setwd()`. Please replace in the following line the path to the folder where you have placed this file -and where the `sf_bikes` folder with the data lives.

```
setwd('.')
```

Chapter 6

Data

In this note, we will use data from the city of San Francisco representing bike trips on their public bike share system. The original source is the SF Open Data portal (link) and the dataset comprises both the location of each station in the Bay Area as well as information on trips (station of origin to station of destination) undertaken in the system from September 2014 to August 2015 and the following year. Since this note is about modeling and not data preparation, a cleanly reshaped version of the data, together with some additional information, has been created and placed in the `sf_bikes` folder. The data file is named `flows.geojson` and, in case you are interested, the (Python) code required to created from the original files in the SF Data Portal is also available on the `flows_prep.ipynb` notebook [url], also in the same folder.

Let us then directly load the file with all the information necessary:

```
db <- readOGR('./data/sf_bikes/flows.geojson')
```

```
## OGR data source with driver: GeoJSON
## Source: "/home/jovyan/work/data/sf_bikes/flows.geojson", layer: "flows"
## with 1722 features
## It has 9 fields
```

```
rownames(db@data) <- db$flow_id
db@data$flow_id <- NULL
```

Note how the interface is slightly different since we are reading a `GeoJSON` file instead of a shapefile.

The data contains the geometries of the flows, as calculated from the Google Maps API, as well as a series of columns with characteristics of each flow:

```
head(db@data)
```

```
##       dest orig straight_dist street_dist total_down total_up trips15
```

```
## 39-41   41   39      1452.201   1804.1150  11.205753 4.698162      68
## 39-42   42   39      1734.861   2069.1557  10.290236 2.897886      23
## 39-45   45   39      1255.349   1747.9928  11.015596 4.593927      83
## 39-46   46   39      1323.303   1490.8361   3.511543 5.038044     258
## 39-47   47   39       715.689    769.9189   0.000000 3.282495     127
## 39-48   48   39      1996.778   2740.1290  11.375186 3.841296      81
##       trips16
## 39-41      68
## 39-42      29
## 39-45      50
## 39-46     163
## 39-47      73
## 39-48      56
```

where `orig` and `dest` are the station IDs of the origin and destination, `street/straight_dist` is the distance in metres between stations measured along the street network or as-the-crow-flies, `total_down/up` is the total downhil and climb in the trip, and `tripsXX` contains the amount of trips undertaken in the years of study.

Chapter 7

"*Seeing*" flows

The easiest way to get a quick preview of what the data looks like spatially is to make a simple plot:

```
plot(db)
```

Equally, if we want to visualize a single route, we can simply subset the table. For example, to get the shape of the trip from station `39` to station `48`, we can:

```
one39to48 <- db[ which(
          db@data$orig == 39 & db@data$dest == 48
          ) , ]
plot(one39to48)
```

or, for the most popular route, we can:

```
most_pop <- db[ which(
          db@data$trips15 == max(db@data$trips15)
          ) , ]
plot(most_pop)
```

These however do not reveal a lot: there is no geographical context (*why are there so many routes along the NE?*) and no sense of how volumes of bikers are allocated along different routes. Let us fix those two.

The easiest way to bring in geographical context is by overlaying the routes on top of a background map of tiles downloaded from the internet. Let us download this using `ggmap`:

```
sf_bb <- c(left=db@bbox['x', 'min'],
           right=db@bbox['x', 'max'],
           bottom=db@bbox['y', 'min'],
           top=db@bbox['y', 'max'])
```

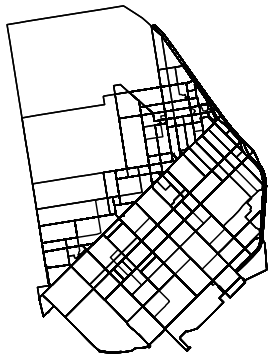


Figure 7.1: Potential routes

```
SanFran <- get_stamenmap(sf_bb,
                         zoom = 14,
                         maptype = "toner-lite")
```

```
## Source : http://tile.stamen.com/toner-lite/14/2620/6330.png
## Source : http://tile.stamen.com/toner-lite/14/2621/6330.png
## Source : http://tile.stamen.com/toner-lite/14/2622/6330.png
## Source : http://tile.stamen.com/toner-lite/14/2620/6331.png
## Source : http://tile.stamen.com/toner-lite/14/2621/6331.png
## Source : http://tile.stamen.com/toner-lite/14/2622/6331.png
## Source : http://tile.stamen.com/toner-lite/14/2620/6332.png
## Source : http://tile.stamen.com/toner-lite/14/2621/6332.png
## Source : http://tile.stamen.com/toner-lite/14/2622/6332.png
## Source : http://tile.stamen.com/toner-lite/14/2620/6333.png
## Source : http://tile.stamen.com/toner-lite/14/2621/6333.png
```

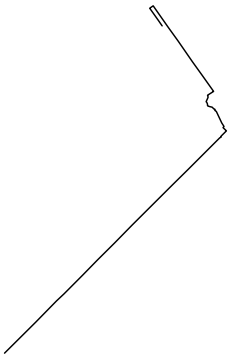


Figure 7.2: Trip from station 39 to 48

```
## Source : http://tile.stamen.com/toner-lite/14/2622/6333.png
```

and make sure it looks like we intend it to look:

```
ggmap(SanFran)
```

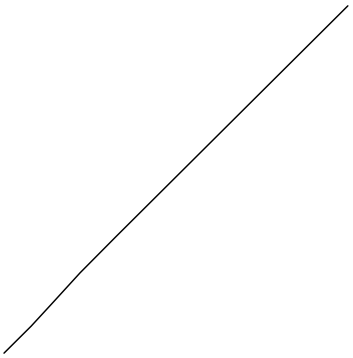


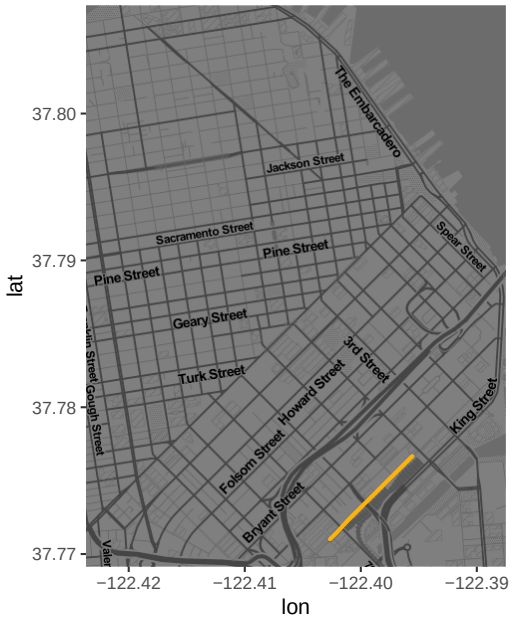
Figure 7.3: Most popular trip

Now to combine tiles and routes, we need to pull out the coordinates that make up each line. For the route example above, this would be:

```
xys1 <- as.data.frame(coordinates(most_pop))
```

Now we can plot the route[1] (note we also dim down the background to focus the attention on flows):

```
ggmap(SanFran, darken=0.5) +
  geom_path(aes(x=X1, y=X2),
            data=xys1,
            size=1,
            color=rgb(0.996078431372549, 0.7019607843137254, 0.03137254901960784),
            lineend='round')
```



Now we can plot all of the lines by using a short `for` loop to build up the table:

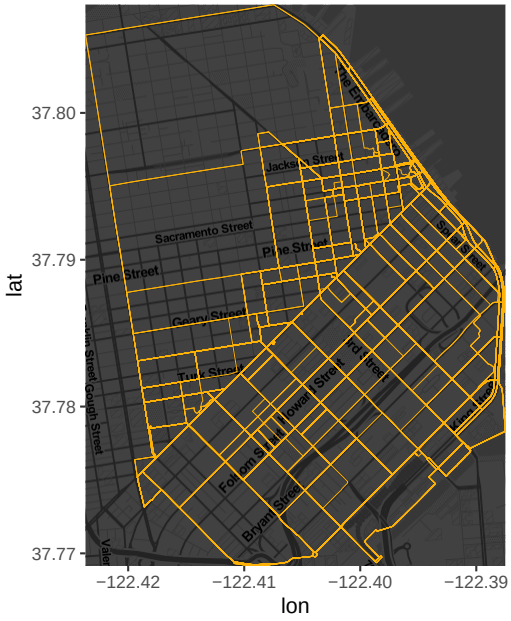
```
# Set up shell data.frame
lines <- data.frame(lat = numeric(0),
                    lon = numeric(0),
                    trips = numeric(0),
                    id = numeric(0)
                    )
# Run loop
for(x in 1:nrow(db)){
  # Pull out row
```

[1] **EXERCISE**: *can you plot the route for the largest climb?*

```
  r <- db[x, ]
  # Extract lon/lat coords
  xys <- as.data.frame(coordinates(r))
  names(xys) <- c('lon', 'lat')
  # Insert trips and id
  xys['trips'] <- r@data$trips15
  xys['id'] <- x
  # Append them to `lines`
  lines <- rbind(lines, xys)
}
```

Now we can go on and plot all of them:

```
ggmap(SanFran, darken=0.75) +
  geom_path(aes(
                x=lon,
                y=lat,
                group=id
                ),
            data=lines,
            size=0.1,
            color=rgb(0.996078431372549, 0.7019607843137254, 0.03137254901960784),
            lineend='round')
```



Finally, we can get a sense of the distribution of the flows by associating a color gradient to each flow based on its number of trips:

```
ggmap(SanFran, darken=0.75) +
  geom_path(aes(
                x=lon,
                y=lat,
                group=id,
                colour=trips
                ),
            data=lines,
            size=log1p(lines$trips / max(lines$trips)),
            lineend='round') +
  scale_colour_gradient(low='grey',
                        high='#07eda0') +
  theme(axis.text.x = element_blank(),
      axis.text.y = element_blank(),
      axis.ticks = element_blank()
      )
```

Note how we transform the size so it's a proportion of the largest trip and then it is compressed with a logarithm.

Chapter 8

Modelling flows

Now we have an idea of the spatial distribution of flows, we can begin to think about modeling them. The core idea in this section is to fit a model that can capture the particular characteristics of our variable of interest (the volume of trips) using a set of predictors that describe the nature of a given flow. We will start from the simplest model and then progressively build complexity until we get to a satisfying point. Along the way, we will be exploring each model using concepts from Gelman and Hill (2006) such as predictive performance checks[1] (PPC)

Before we start running regressions, let us first standardize the predictors so we can interpret the intercept as the average flow when all the predictors take the average value, and so we can interpret the model coefficients as changes in standard deviation units:

```
# Scale all the table
db_std <- as.data.frame(scale(db@data))
# Reset trips as we want the original version
db_std$trips15 <- db@data$trips15
db_std$trips16 <- db@data$trips16
# Reset origin and destination station and express them as factors
db_std$orig <- as.factor(db@data$orig)
db_std$dest <- as.factor(db@data$dest)
```

8.1 Baseline model

One of the simplest possible models we can fit in this context is a linear model that explains the number of trips as a function of the straight distance between

[1] For a more elaborate introduction to PPC, have a look at Chapters 7 and 8.

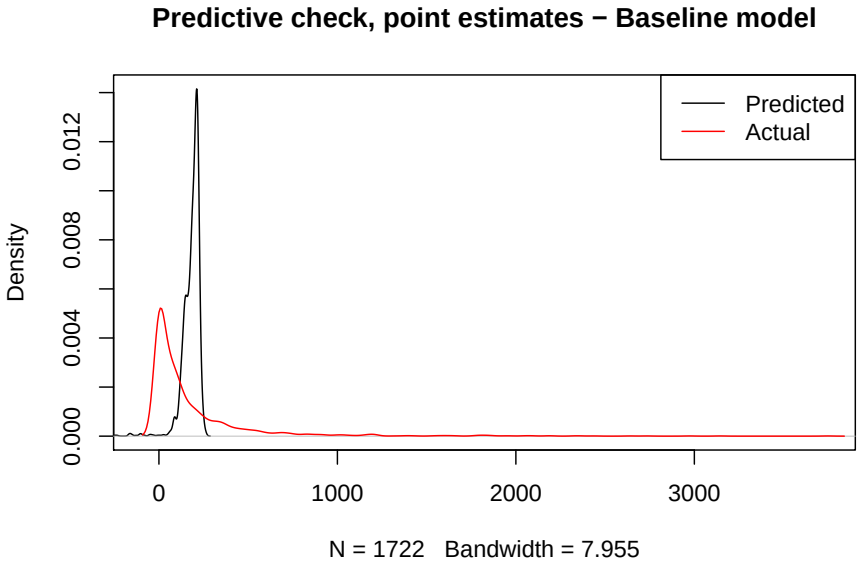
the two stations and total amount of climb and downhill. We will take this as the baseline on which we can further build later:

```
m1 <- lm('trips15 ~ straight_dist + total_up + total_down', data=db_std)
summary(m1)
```

```
##
## Call:
## lm(formula = "trips15 ~ straight_dist + total_up + total_down",
##     data = db_std)
##
## Residuals:
##    Min     1Q Median     3Q    Max
## -261.9 -168.3 -102.4   30.8 3527.4
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)    182.070      8.110  22.451  < 2e-16 ***
## straight_dist   17.906      9.108   1.966   0.0495 *
## total_up       -44.100      9.353  -4.715 2.61e-06 ***
## total_down     -20.241      9.229  -2.193   0.0284 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 336.5 on 1718 degrees of freedom
## Multiple R-squared:  0.02196,    Adjusted R-squared:  0.02025
## F-statistic: 12.86 on 3 and 1718 DF,  p-value: 2.625e-08
```

To explore how good this model is, we will be comparing the predictions the model makes about the number of trips each flow should have with the actual number of trips. A first approach is to simply plot the distribution of both variables:

```
plot(density(m1$fitted.values),
     xlim=c(-100, max(db_std$trips15)),
     main='')
lines(density(db_std$trips15),
      col='red',
      main='')
legend('topright',
       c('Predicted', 'Actual'),
       col=c('black', 'red'),
       lwd=1)
title(main="Predictive check, point estimates - Baseline model")
```



The plot makes pretty obvious that our initial model captures very few aspects of the distribution we want to explain. However, we should not get too attached to this plot just yet. What it is showing is the distribution of predicted *point* estimates from our model. Since our model is not deterministic but inferential, there is a certain degree of uncertainty attached to its predictions, and that is completely absent from this plot.

Generally speaking, a given model has two sources of uncertainty: *predictive*, and *inferential*. The former relates to the fact that the equation we fit does not capture all the elements or in the exact form they enter the true data generating process; the latter has to do with the fact that we never get to know the true value of the model parameters only guesses (estimates) subject to error and uncertainty. If you think of our linear model above as

$$T_{ij} = X_{ij}\beta + \epsilon_{ij}$$

where T_{ij} represents the number of trips undertaken between station i and j, X_{ij} is the set of explanatory variables (length, climb, descent, etc.), and ϵ_{ij} is an error term assumed to be distributed as a normal distribution $N(0, \sigma)$; then predictive uncertainty comes from the fact that there are elements to some extent relevant for y that are not accounted for and thus subsummed into ϵ_{ij}. Inferential uncertainty comes from the fact that we never get to know β but only an estimate of it which is also subject to uncertainty itself.

Taking these two sources into consideration means that the black line in the plot above represents only the behaviour of our model we expect if the error term is

absent (no predictive uncertainty) and the coefficients are the true estimates (no inferential uncertainty). However, this is not necessarily the case as our estimate for the uncertainty of the error term is certainly not zero, and our estimates for each parameter are also subject to a great deal of inferential variability. we do not know to what extent other outcomes would be just as likely. Predictive checking relates to simulating several feasible scenarios under our model and use those to assess uncertainty and to get a better grasp of the quality of our predictions.

Technically speaking, to do this, we need to build a mechanism to obtain a possible draw from our model and then repeat it several times. The first part of those two steps can be elegantly dealt with by writing a short function that takes a given model and a set of predictors, and produces a possible random draw from such model:

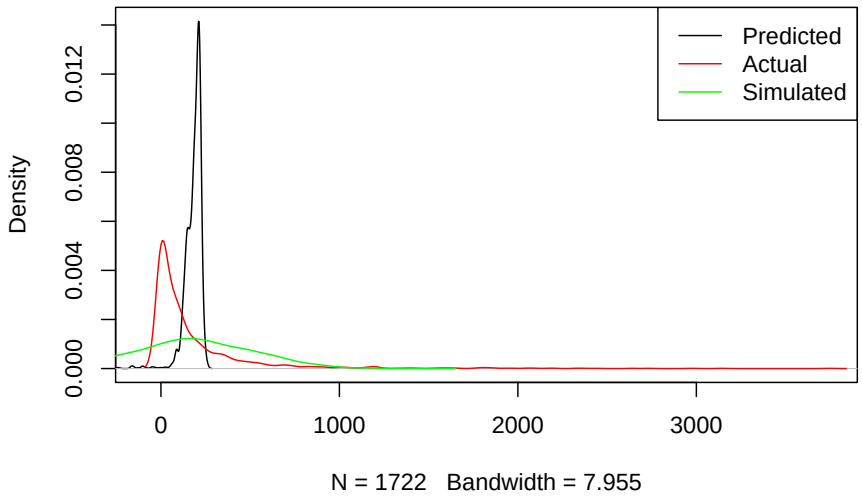
```
generate_draw <- function(m){
  # Set up predictors matrix
  x <- model.matrix(m)
  # Obtain draws of parameters (inferential uncertainty)
  sim_bs <- sim(m, 1)
  # Predicted value
  mu <- x %*% sim_bs@coef[1, ]
  # Draw
  n <- length(mu)
  y_hat <- rnorm(n, mu, sim_bs@sigma[1])
  return(y_hat)
}
```

This function takes a model `m` and the set of covariates `x` used and returns a random realization of predictions from the model. To get a sense of how this works, we can get and plot a realization of the model, compared to the expected one and the actual values:

```
new_y <- generate_draw(m1)

plot(density(m1$fitted.values),
     xlim=c(-100, max(db_std$trips15)),
     ylim=c(0, max(c(
                      max(density(m1$fitted.values)$y),
                      max(density(db_std$trips15)$y)
                      )
                    )
            ),
     col='black',
     main='')
lines(density(db_std$trips15),
      col='red',
```

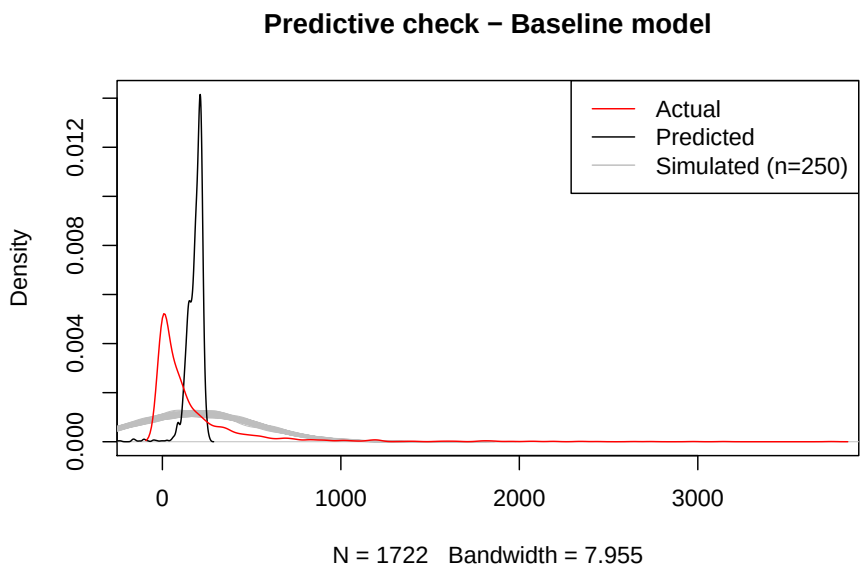
```
      main='')
lines(density(new_y),
      col='green',
      main='')
legend('topright',
       c('Predicted', 'Actual', 'Simulated'),
       col=c('black', 'red', 'green'),
       lwd=1)
```



Once we have this "draw engine", we can set it to work as many times as we want using a simple `for` loop. In fact, we can directly plot these lines as compared to the expected one and the trip count:

```
plot(density(m1$fitted.values),
     xlim=c(-100, max(db_std$trips15)),
     ylim=c(0, max(c(
                  max(density(m1$fitted.values)$y),
                  max(density(db_std$trips15)$y)
                  )
                )
        ),
     col='white',
     main='')
# Loop for realizations
for(i in 1:250){
```

```
  tmp_y <- generate_draw(m1)
  lines(density(tmp_y),
        col='grey',
        lwd=0.1
        )
}
#
lines(density(m1$fitted.values),
      col='black',
      main='')
lines(density(db_std$trips15),
      col='red',
      main='')
legend('topright',
       c('Actual', 'Predicted', 'Simulated (n=250)'),
       col=c('red', 'black', 'grey'),
       lwd=1)
title(main="Predictive check - Baseline model")
```



The plot shows there is a significant mismatch between the fitted values, which are much more concentrated around small positive values, and the realizations of our "inferential engine", which depict a much less concentrated distribution of values. This is likely due to the combination of two different reasons: on the one hand, the accuracy of our estimates may be poor, causing them to jump around a wide range of potential values and hence resulting in very diverse

predictions (inferential uncertainty); on the other hand, it may be that the amount of variation we are not able to account for in the model[2] is so large that the degree of uncertainty contained in the error term of the model is very large, hence resulting in such a flat predictive distribution.

It is important to keep in mind that the issues discussed in the paragraph above relate only to the uncertainty behind our model, not to the point predictions derived from them, which are a mechanistic result of the minimization of the squared residuals and hence are not subject to probability or inference. That allows them in this case to provide a fitted distribution much more accurate apparently (black line above). However, the lesson to take from this model is that, even if the point predictions (fitted values) are artificially accurate[3], our capabilities to infer about the more general underlying process are fairly limited.

8.2 Improving the model

The bad news from the previous section is that our initial model is not great at explaining bike trips. The good news is there are several ways in which we can improve this. In this section we will cover three main extensions that exemplify three different routes you can take when enriching and augmenting models in general, and spatial interaction ones in particular[4]. These three routes are aligned around the following principles:

1. Use better approximations to model your dependent variable.
2. Recognize the structure of your data.
3. Get better predictors.

- **Use better approximations to model your dependent variable**

Standard OLS regression assumes that the error term and, since the predictors are deterministic, the dependent variable are distributed following a normal (gaussian) distribution. This is usually a good approximation for several phenomena of interest, but maybe not the best one for trips along routes: for one, we know trips cannot be negative, which the normal distribution does not account for[5]; more subtly, their distribution is not really symmetric but skewed with a very long tail on the right. This is common in variables that represent counts and that is why usually it is more appropriate to fit a model that relies on a distribution different from the normal.

One of the most common distributions for this cases is the Poisson, which can

[2] The R^2 of our model is around 2%

[3] which they are not really, in light of the comparison between the black and red lines.

[4] These principles are general and can be applied to pretty much any modeling exercise you run into. The specific approaches we take in this note relate to spatial interaction models

[5] For an illustration of this, consider the amount of probability mass to the left of zero in the predictive checks above.

be incorporated through a general linear model (or GLM). The underlying assumption here is that instead of $T_{ij} \sim N(\mu_{ij}, \sigma)$, our model now follows:

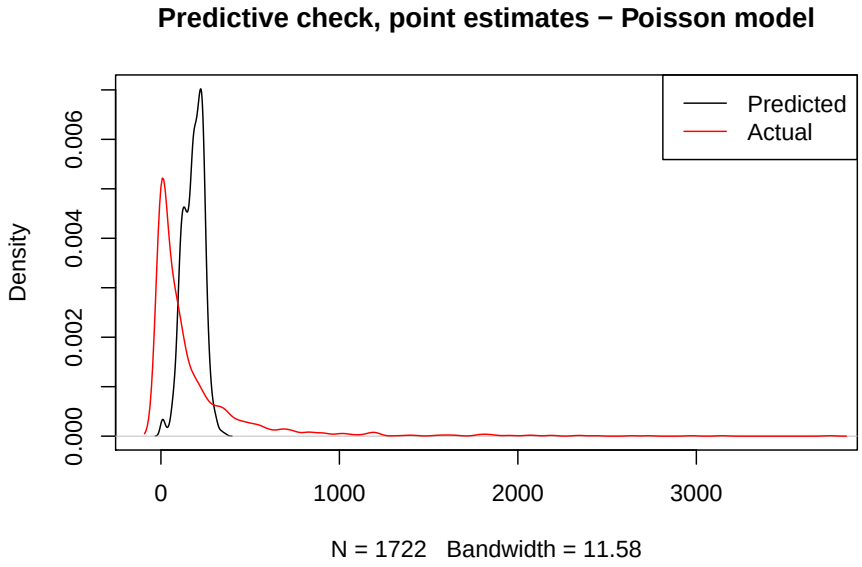
$$T_{ij} \sim Poisson(\exp^{X_{ij}\beta})$$

As usual, such a model is easy to run in R:

```
m2 <- glm('trips15 ~ straight_dist + total_up + total_down',
          data=db_std,
          family=poisson,
          )
```

Now let's see how much better, if any, this approach is. To get a quick overview, we can simply plot the point predictions:

```
plot(density(m2$fitted.values),
     xlim=c(-100, max(db_std$trips15)),
     ylim=c(0, max(c(
                  max(density(m2$fitted.values)$y),
                  max(density(db_std$trips15)$y)
                  )
                )
      ),
     col='black',
     main='')
lines(density(db_std$trips15),
      col='red',
      main='')
legend('topright',
       c('Predicted', 'Actual'),
       col=c('black', 'red'),
       lwd=1)
title(main="Predictive check, point estimates - Poisson model")
```



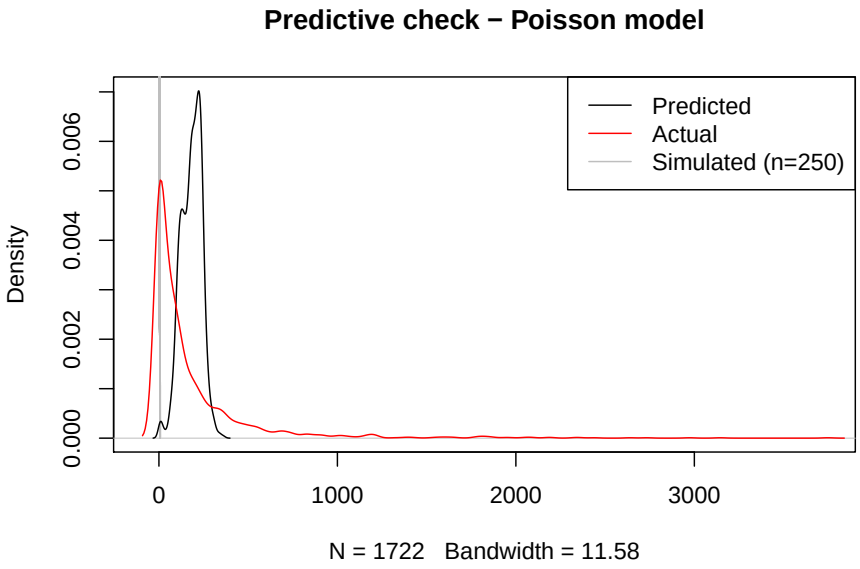
To incorporate uncertainty to these predictions, we need to tweak our `generate_draw` function so it accommodates the fact that our model is not linear anymore.

```
generate_draw_poi <- function(m){
  # Set up predictors matrix
  x <- model.matrix(m)
  # Obtain draws of parameters (inferential uncertainty)
  sim_bs <- sim(m, 1)
  # Predicted value
  xb <- x %*% sim_bs@coef[1, ]
  #xb <- x %*% m$coefficients
  # Transform using the link function
  mu <- exp(xb)
  # Obtain a random realization
  y_hat <- rpois(n=length(mu), lambda=mu)
  return(y_hat)
}
```

And then we can examine both point predictions an uncertainty around them:

```
plot(density(m2$fitted.values),
     xlim=c(-100, max(db_std$trips15)),
     ylim=c(0, max(c(
                  max(density(m2$fitted.values)$y),
                  max(density(db_std$trips15)$y)
```

```
                  )
                )
      ),
     col='white',
     main='')
# Loop for realizations
for(i in 1:250){
  tmp_y <- generate_draw_poi(m2)
  lines(density(tmp_y),
        col='grey',
        lwd=0.1
        )
}
#
lines(density(m2$fitted.values),
      col='black',
      main='')
lines(density(db_std$trips15),
      col='red',
      main='')
legend('topright',
       c('Predicted', 'Actual', 'Simulated (n=250)'),
       col=c('black', 'red', 'grey'),
       lwd=1)
title(main="Predictive check - Poisson model")
```



Voila! Although the curve is still a bit off, centered too much to the right of the actual data, our predictive simulation leaves the fitted values right in the middle. This speaks to a better fit of the model to the actual distribution othe original data follow.

- **Recognize the structure of your data**

So far, we've treated our dataset as if it was flat (i.e. comprise of fully independent realizations) when in fact it is not. Most crucially, our baseline model does not account for the fact that every observation in the dataset pertains to a trip between two stations. This means that all the trips from or to the same station probably share elements which likely help explain how many trips are undertaken between stations. For example, think of trips to an from a station located in the famous Embarcadero, a popular tourist spot. Every route to and from there probably has more trips due to the popularity of the area and we are currently not acknowledging it in the model.

A simple way to incorporate these effects into the model is through origin and destination fixed effects. This approach shares elements with both spatial fixed effects and multilevel modeling and essentially consists of including a binary variable for every origin and destination station. In mathematical notation, this equates to:

$$T_{ij} = X_{ij}\beta + \delta_i + \delta_j + \epsilon_{ij}$$

where δ_i and δ_j are origin and destination station fixed effects[6], and the rest is as above. This strategy accounts for all the unobserved heterogeneity associated with the location of the station. Technically speaking, we simply need to introduce `orig` and `dest` in the the model:

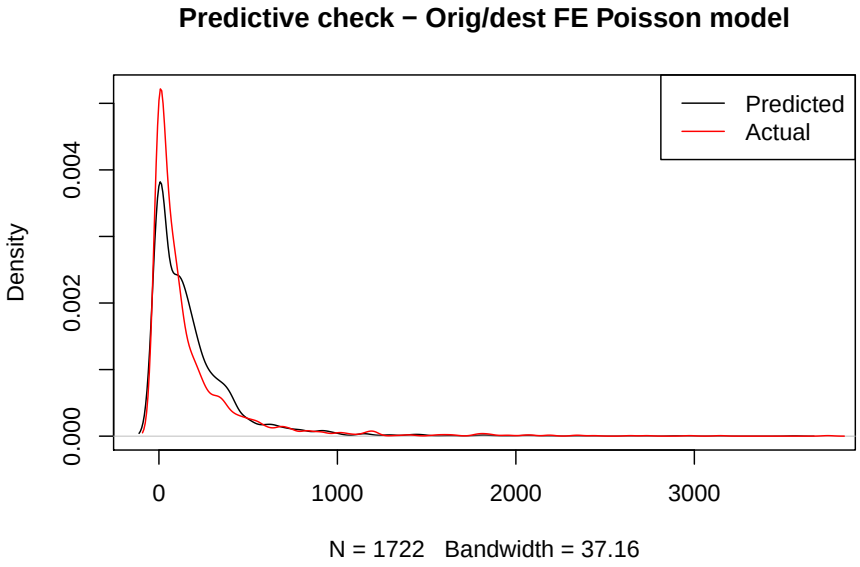
```
m3 <- glm('trips15 ~ straight_dist + total_up + total_down + orig + dest',
          data=db_std,
          family=poisson)
```

And with our new model, we can have a look at how well it does at predicting the overall number of trips[7]:

```
plot(density(m3$fitted.values),
     xlim=c(-100, max(db_std$trips15)),
     ylim=c(0, max(c(
                  max(density(m3$fitted.values)$y),
                  max(density(db_std$trips15)$y)
                  )
                )
      ),
     col='black',
     main='')
lines(density(db_std$trips15),
      col='red',
      main='')
legend('topright',
       c('Predicted', 'Actual'),
       col=c('black', 'red'),
       lwd=1)
title(main="Predictive check - Orig/dest FE Poisson model")
```

[6] In this session, δ_i and δ_j are estimated as independent variables so their estimates are similar to interpret to those in β. An alternative approach could be to model them as random effects in a multilevel framework.

[7] Although, theoretically, we could also include simulations of the model in the plot to get a better sense of the uncertainty behind our model, in practice this seems troublesome. The problems most likely arise from the fact that many of the origin and destination binary variable coefficients are estimated with a great deal of uncertainty. This causes some of the simulation to generate extreme values that, when passed through the exponential term of the Poisson link function, cause problems. If anything, this is testimony of how a simple fixed effect model can sometimes lack accuracy and generate very uncertain estimates. A potential extension to work around these problems could be to fit a multilevel model with two specific levels beyond the trip-level: one for origin and another one for destination stations.



That looks significantly better, doesn't it? In fact, our model now better accounts for the long tail where a few routes take a lot of trips. This is likely because the distribution of trips is far from random across stations and our origin and destination fixed effects do a decent job at accounting for that structure. However our model is still notably underpredicting less popular routes and overpredicting routes with above average number of trips. Maybe we should think about moving beyond a simple linear model.

- **Get better predictors**

The final extension is, in principle, always available but, in practice, it can be tricky to implement. The core idea is that your baseline model might not have the best measurement of the phenomena you want to account for. In our example, we can think of the distance between stations. So far, we have been including the distance measured "as the crow flies" between stations. Although in some cases this is a good approximation (particularly when distances are long and likely route taken is as close to straight as possible), in some cases like ours, where the street layout and the presence of elevation probably matter more than the actual final distance pedalled, this is not necessarily a safe assumption.

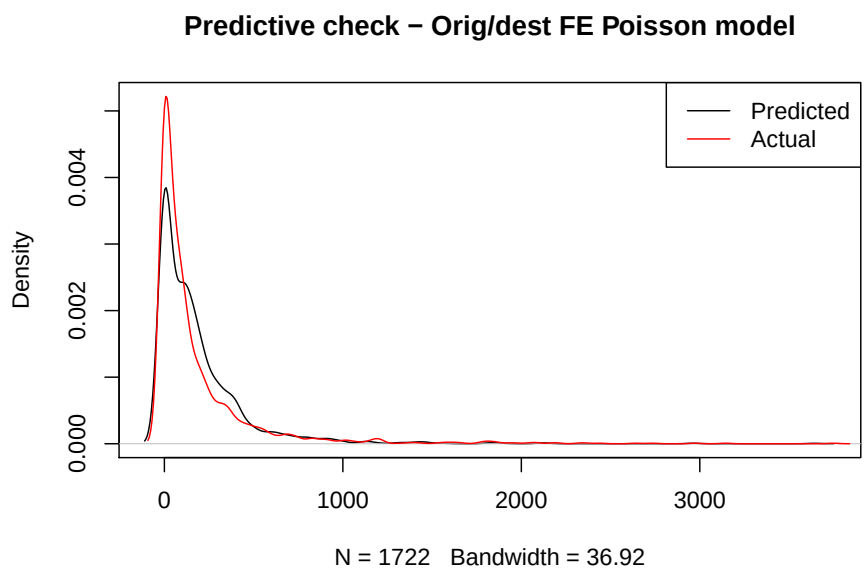
As an exampe of this approach, we can replace the straight distance measurements for more refined ones based on the Google Maps API routes. This is very easy as all we need to do (once the distances have been calculated!) is to swap `straight_dist` for `street_dist`:

```
m4 <- glm('trips15 ~ street_dist + total_up + total_down + orig + dest',
          data=db_std,
```

```
          family=poisson)
```

And we can similarly get a sense of our predictive fitting with:

```
plot(density(m4$fitted.values),
     xlim=c(-100, max(db_std$trips15)),
     ylim=c(0, max(c(
                  max(density(m4$fitted.values)$y),
                  max(density(db_std$trips15)$y)
                  )
                )
      ),
     col='black',
     main='')
lines(density(db_std$trips15),
      col='red',
      main='')
legend('topright',
       c('Predicted', 'Actual'),
       col=c('black', 'red'),
       lwd=1)
title(main="Predictive check - Orig/dest FE Poisson model")
```



Hard to tell any noticeable difference, right? To see if there is any, we can have a look at the estimates obtained:

```
summary(m4)$coefficients['street_dist', ]
```

```
##       Estimate     Std. Error        z value       Pr(>|z|)
##  -9.961619e-02   2.688731e-03  -3.704952e+01  1.828096e-300
```

And compare this to that of the straight distances in the previous model:

```
summary(m3)$coefficients['straight_dist', ]
```

```
##       Estimate     Std. Error        z value       Pr(>|z|)
##  -7.820014e-02   2.683052e-03  -2.914596e+01  9.399407e-187
```

As we can see, the differences exist but ar not massive. Let's use this example to learn how to interpret coefficients in a Poisson model[8]. Effectively, these estimates can be understood as multiplicative effects. Since our model fits

$$T_{ij} \sim Poisson(\exp^{X_{ij}\beta})$$

we need to transform β through an exponential in order to get a sense of the effect of distance on the number of trips. This means that for the street distance, our original estimate is $\beta_{street} = -0.0996$, but this needs to be translated through the exponential into $e^{-0.0996} = 0.906$. In other words, since distance is expressed in standard deviations[9], we can expect a 10% decrease in the number of trips for an increase of one standard deviation (about 1Km) in the distance between the stations. This can be compared with $e^{-0.0782} = 0.925$ for the straight distances, or a reduction of about 8% the number of trips for every increase of a standard deviation (about 720m).

[8] See section 6.2 of Gelman and Hill (2006) for a similar treatment of these.
[9] Remember the transformation at the very beginning.

Chapter 9

Predicting flows

So far we have put all of our modeling efforts in understanding the model we fit and improving such model so it fits our data as closely as possible. This is essential in any modelling exercise but should be far from a stopping point. Once we're confident our model is a decent representation of the data generating process, we can start exploiting it. In this section, we will cover one specific case that showcases how a fitted model can help: out-of-sample forecasts.

It is August 2015, and you have just started working as a data scientist for the bikeshare company that runs the San Francisco system. You join them as they're planning for the next academic year and, in order to plan their operations (re-allocating vans, station maintenance, etc.), they need to get a sense of how many people are going to be pedalling across the city and, crucially, *where* they are going to be pedalling through. What can you do to help them?

The easiest approach is to say "well, a good guess for how many people will be going between two given stations this coming year is how many went through last year, isn't it?". This is one prediction approach. However, you could see how, even if the same process governs over both datasets (2015 and 2016), each year will probably have some idiosyncracies and thus looking too closely into one year might not give the best possible answer for the next one. Ideally, you want a good stylized synthesis that captures the bits that stay constant over time and thus can be applied in the future and that ignores those aspects that are too particular to a given point in time. That is the rationale behind using a fitted model to obtain predictions.

However good any theory though, the truth is in the pudding. So, to see if a modeling approach is better at producing forecasts than just using the counts from last year, we can put them to a test. The way this is done when evaluating the predictive performance of a model (as this is called in the literature) relies on two basic steps: a) obtain predictions from a given model and b) compare those to the actual values (in our case, with the counts for 2016 in `trips16`)

and get a sense of "how off" they are. We have essentially covered a) above; for b), there are several measures to use. We will use one of the most common ones, the root mean squared error (RMSE), which roughly gives a sense of the average difference between a predicted vector and the real deal:

$$RMSE = \sqrt{\sum_{ij} (\hat{T_{ij}} - T_{ij})^2}$$

where $\hat{T_{ij}}$ is the predicted amount of trips between stations i and j. RMSE is straightforward in R and, since we will use it a couple of times, let's write a short function to make our lives easier:

```
rmse <- function(t, p){
  se <- (t - p)^2
  mse <- mean(se)
  rmse <- sqrt(mse)
  return(rmse)
}
```

where `t` stands for the vector of true values, and `p` is the vector of predictions. Let's give it a spin to make sure it works:

```
rmse_m4 <- rmse(db_std$trips16, m4$fitted.values)
rmse_m4
```

```
## [1] 256.2197
```

That means that, on average, predictions in our best model `m4` are 256 trips off. Is this good? Bad? Worse? It's hard to say but, being practical, what we can say is whether this better than our alternative. Let us have a look at the RMSE of the other models as well as that of simply plugging in last year's counts:[1]

```
rmses <- data.frame(model=c('OLS', 'Poisson', 'Poisson + FE',
                            'Poisson + FE + street dist.',
                            'Trips-2015'
                            ),
                    RMSE=c(rmse(db_std$trips16,
                               m1$fitted.values),
                           rmse(db_std$trips16,
                               m2$fitted.values),
                           rmse(db_std$trips16,
                               m3$fitted.values),
                           rmse(db_std$trips16,
                               m4$fitted.values),
                           rmse(db_std$trips16,
```

[1] **EXERCISE**: can you create a single plot that displays the distribution of the predicted values of the five different ways to predict trips in 2016 and the actual counts of trips?

```
                                db_std$trips15)
                            )
                        )
rmses
```

```
##                           model     RMSE
## 1                           OLS 323.6135
## 2                       Poisson 320.8962
## 3                  Poisson + FE 254.4468
## 4 Poisson + FE + street dist. 256.2197
## 5                    Trips-2015 131.0228
```

The table is both encouraging and disheartning at the same time. On the one hand, all the modeling techniques covered above behave as we would expect: the baseline model displays the worst predicting power of all, and every improvement (except the street distances!) results in notable decreases of the RMSE. This is good news. However, on the other hand, all of our modelling efforts fall short of given a better guess than simply using the previous year's counts. *Why? Does this mean that we should not pay attention to modeling and inference?* Not really. Generally speaking, a model is as good at predicting as it is able to mimic the underlying process that gave rise to the data in the first place. The results above point to a case where our model is not picking up all the factors that determine the amount of trips undertaken in a give route. This could be improved by enriching the model with more/better predictors, as we have seen above. Also, the example above seems to point to a case where those idiosyncracies in 2015 that the model does not pick up seem to be at work in 2016 as well. This is great news for our prediction efforts this time, but we have no idea why this is the case and, for all that matters, it could change the coming year. Besides the elegant quantification of uncertainty, the true advantage of a modeling approach in this context is that, if well fit, it is able to pick up the fundamentals that apply over and over. This means that, if next year we're not as lucky as this one and previous counts are not good predictors but the variables we used in our model continue to have a role in determining the outcome, the data scientist should be luckier and hit a better prediction.

Chapter 10

References

Chapter 11

Spatial Econometrics

DA-B to fill in

Chapter 12

Multilevel Models (Pt. I)

FR to fill in

Chapter 13

Multilevel Models (Pt. II)

FR to fill in

Chapter 14

GWR

FR

Chapter 15

Space-Time Analysis

FR

Bibliography

Banerjee, S., Carlin, B. P., and Gelfand, A. E. (2014). *Hierarchical modeling and analysis for spatial data*. Crc Press.

Bivand, R. S., Pebesma, E., and Gómez-Rubio, V. (2013). *Applied Spatial Data Analysis with R*. Springer New York.

Brunsdon, C. and Comber, L. (2015). *An introduction to R for spatial analysis & mapping*. Sage.

Cressie, N. (2015). *Statistics for spatial data*. John Wiley & Sons.

Gelman, A. and Hill, J. (2006). *Data analysis using regression and multilevel/hierarchical models*. Cambridge University Press.

Grolemund, G. and Wickham, H. (2019). *R for Data Science*. O'Reilly, US.

Lovelace, R. and Cheshire, J. (2014). Introduction to visualising spatial data in R. *National Centre for Research Methods Working Papers*, 14(03).

Lovelace, R., Nowosad, J., and Muenchow, J. (2020). *An introduction to statistical learning*, volume 112. CRC Press, R Series.

Singleton, A. (2017). Geographic data science for urban analytics. Online course.

Williamson, P. (2018). Survey analysis.

Xie, Y., Allaire, J., and Grolemund, G. (2019). *R Markdown: The Definitive Guide*. CRC Press, Taylor & Francis, Chapman & Hall Book.