Linking R to the Web (Mapping and Analysis)

Alex Singleton

3 November 2015

Learning Objectives

1. Learn how R can be used for web analysis
2. Have the skills to utilize web mapping servers to provide contextual detail for maps within R
3. Use geocoding and routing functionality to plot locations and shortest paths
4. Understanding of loops

Introduction

This practical introduces a method for using online cartography to provide context for maps created within R. These functions are within the `ggmap()` package which has a syntax very similar to `ggplot()` used in the previous practical. After exploring the plotting functions, you will also utilize web analysis functionality that enable geocoding of address and routing along a street network.

First we will install and load the necessary packages.

```
install.packages("ggmap", depend = TRUE)
install.packages("RColorBrewer", depend = TRUE)
install.packages("rgdal", depend = TRUE)
install.packages("jsonlite", depend = TRUE)
install.packages("httr", depend = TRUE)
```

```
library("ggmap")
library("RColorBrewer")
library("rgdal")
library("RCurl")
library("jsonlite")
library("httr")
```

For this practical we will be using some data from AirBnB (https://www.airbnb.co.uk/) concerning the locations of property that has been identified by the owners as being within Manhattan, NYC. We will first read in these data.

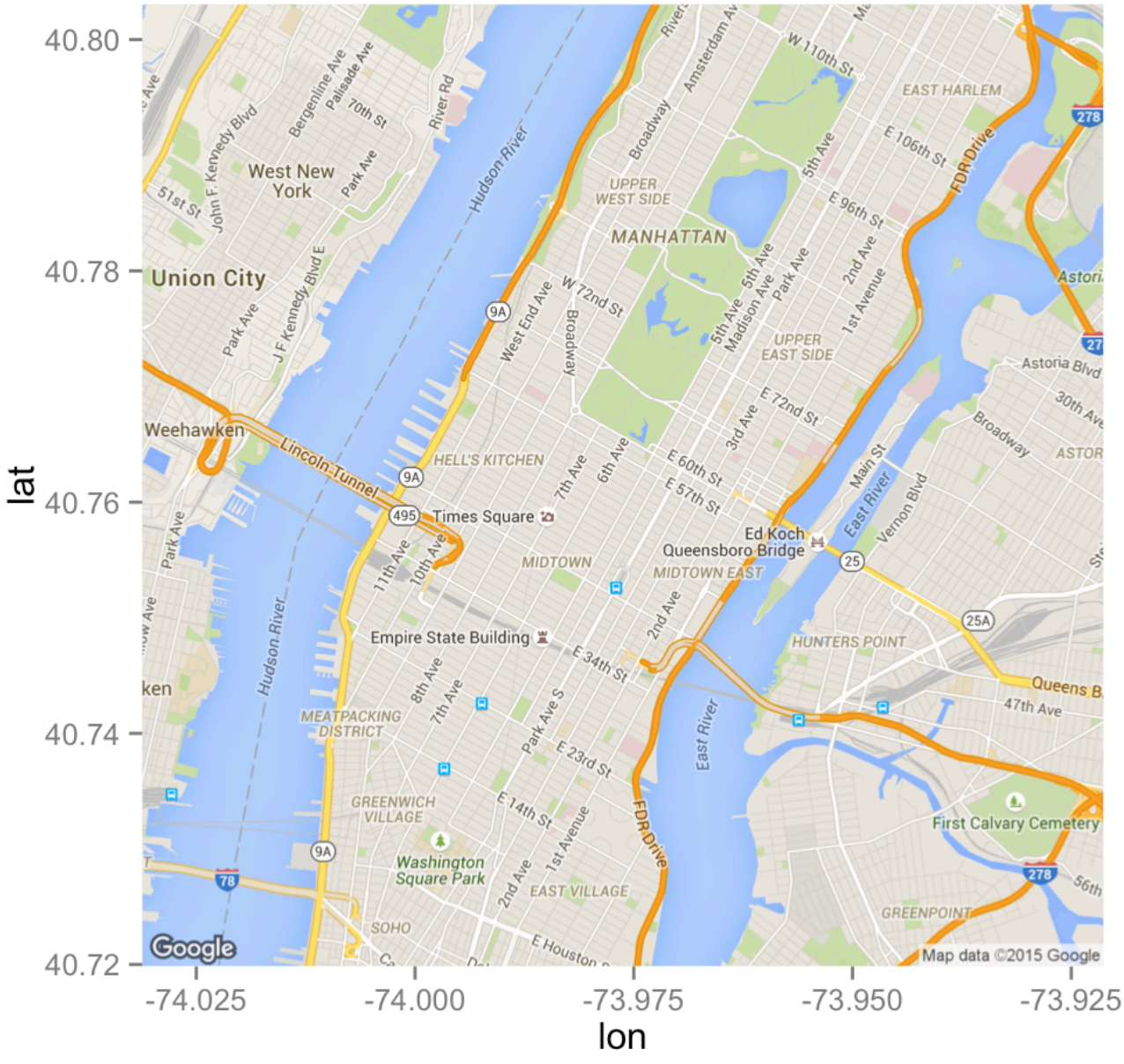
```
listings <- read.csv("listings.csv")
listings$price_beds <- listings$price / listings$beds
head(listings)
```

```
##        id latitude longitude property_type       room_type accommodates
## 1 2082223 40.71031 -74.01638     Apartment    Private room            1
## 2 2986941 40.71728 -74.01524     Apartment Entire home/apt            2
## 3 1712688 40.71177 -74.01730     Apartment Entire home/apt            2
## 4  845495 40.70743 -74.01732     Apartment Entire home/apt            4
## 5 2373737 40.70773 -74.01754     Apartment Entire home/apt            4
## 6 1777007 40.71078 -74.01623     Apartment     Shared room            1
##   bathrooms bedrooms beds bed_type price review_scores_rating price_beds
## 1         1        1    1 Real Bed    80                    90         80
## 2         1        1    1 Real Bed   300                   100        300
## 3         1        0    1 Real Bed   400                    NA        400
## 4         1        1    2 Real Bed   250                    97        125
## 5         1        1    1 Real Bed   255                    93        255
## 6        NA        1    1    Couch    43                    94         43
```

Plotting a base map

To plot a base map we use the `GetMap()` function which requires a number of input parameters including "centre" which is a latitude, longitude pair for the centre of the map. For this, we will take mean of the property locations to centre the map. The other parameter required is "zoom" which sets the scale of the map (low number = globe; high number = close to streets). The "maptype" controls the tileset used for the map. There are a range of options which can be viewed by clicking `?getmap()`.

```
map<- get_map(c(mean(listings$longitude),mean(listings$latitude)),zoom=13,maptype
= "roadmap")
P <- ggmap(map) # Note we have stored the basic map in the new object P
P
```

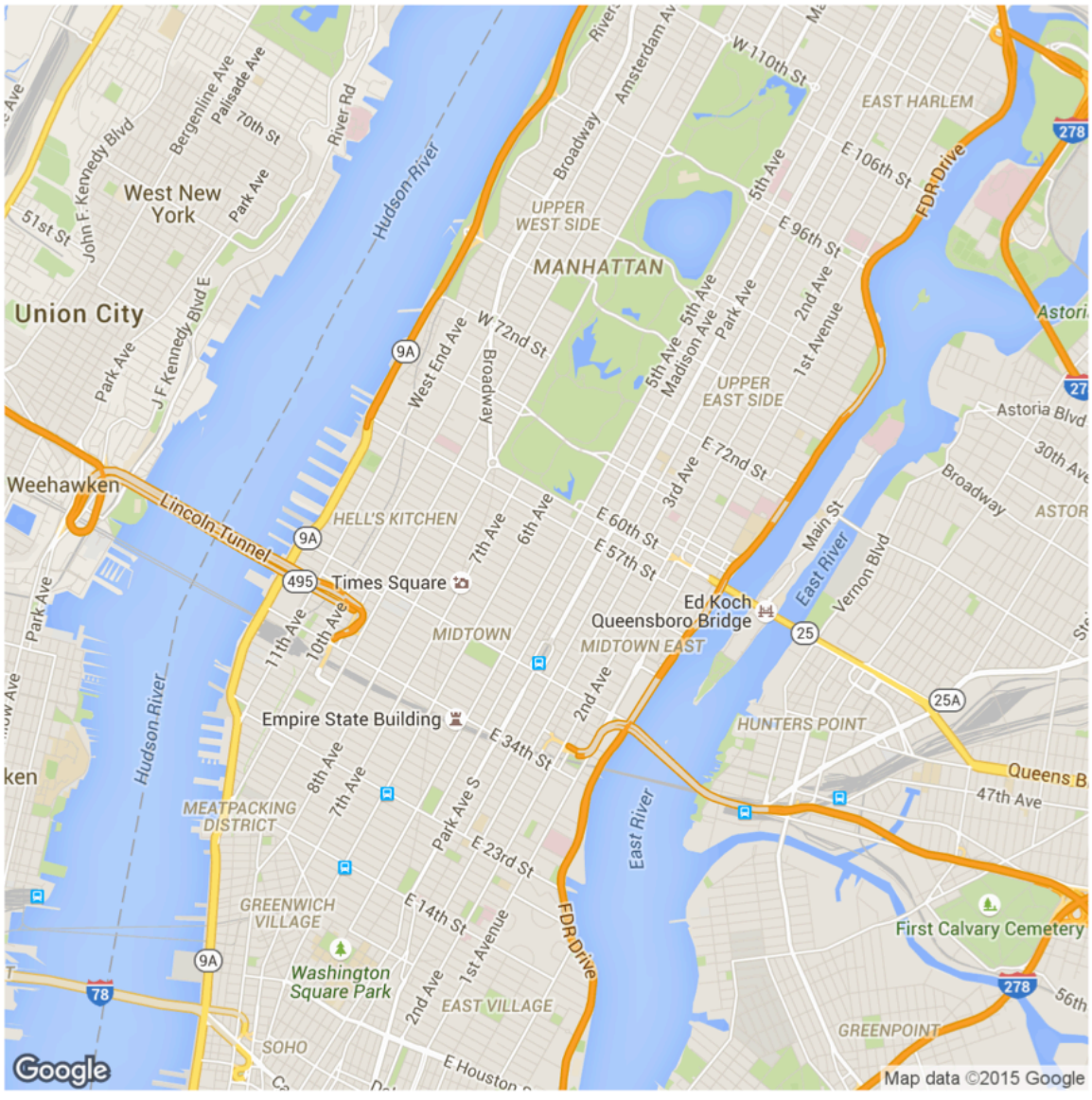


Another way in which we can setup a map is using a keyword rather than a specific lat/lon. For example, the following example will give you a map of Liverpool.

```
ggmap(get_map("Liverpool",zoom=13,maptype = "roadmap"))
```

As shown in the previous tutorial, we can control elements of the map. You will see that the x and y are labeled as latitude and longitude - unlike the example shown in the last practical, these coordinates are not of equal size, so we do not need the `coord_equal()` option. We can also control elements of the plot. For example, if we want to hide the axis.

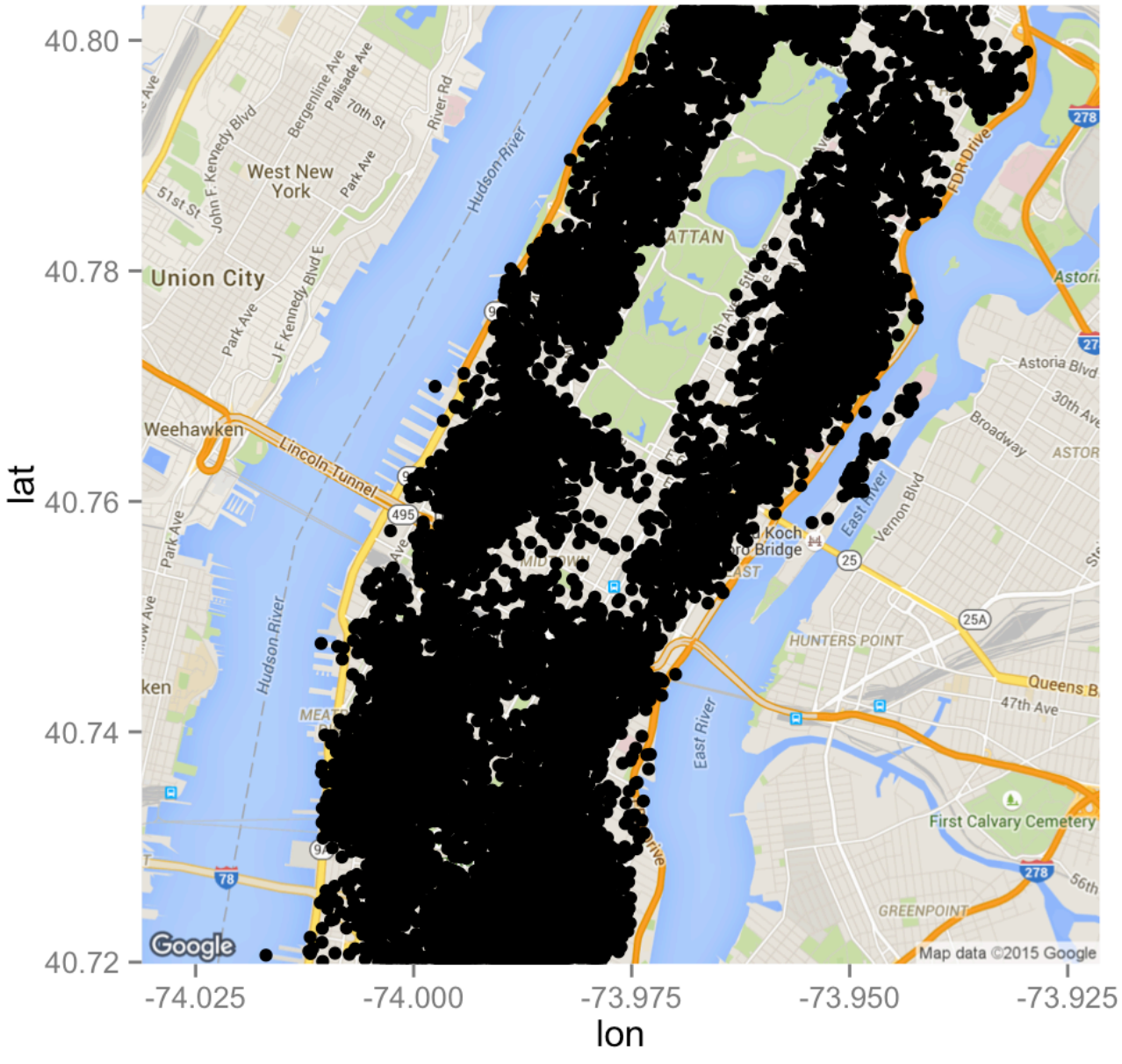
```
P +  theme(axis.line = element_blank(),
    axis.text = element_blank(),
    axis.title=element_blank(),
    axis.ticks = element_blank(),
    legend.key = element_blank(),
    panel.grid.major = element_blank(),
    panel.grid.minor = element_blank(),
    panel.border = element_blank(),
    panel.background = element_blank())
```



Plotting a point

We can now add the listings (points) to the map - each one has a latitude and longitude co-ordinate. To begin with we will just show the location of the points. We use the “size” option to adjust the point size.

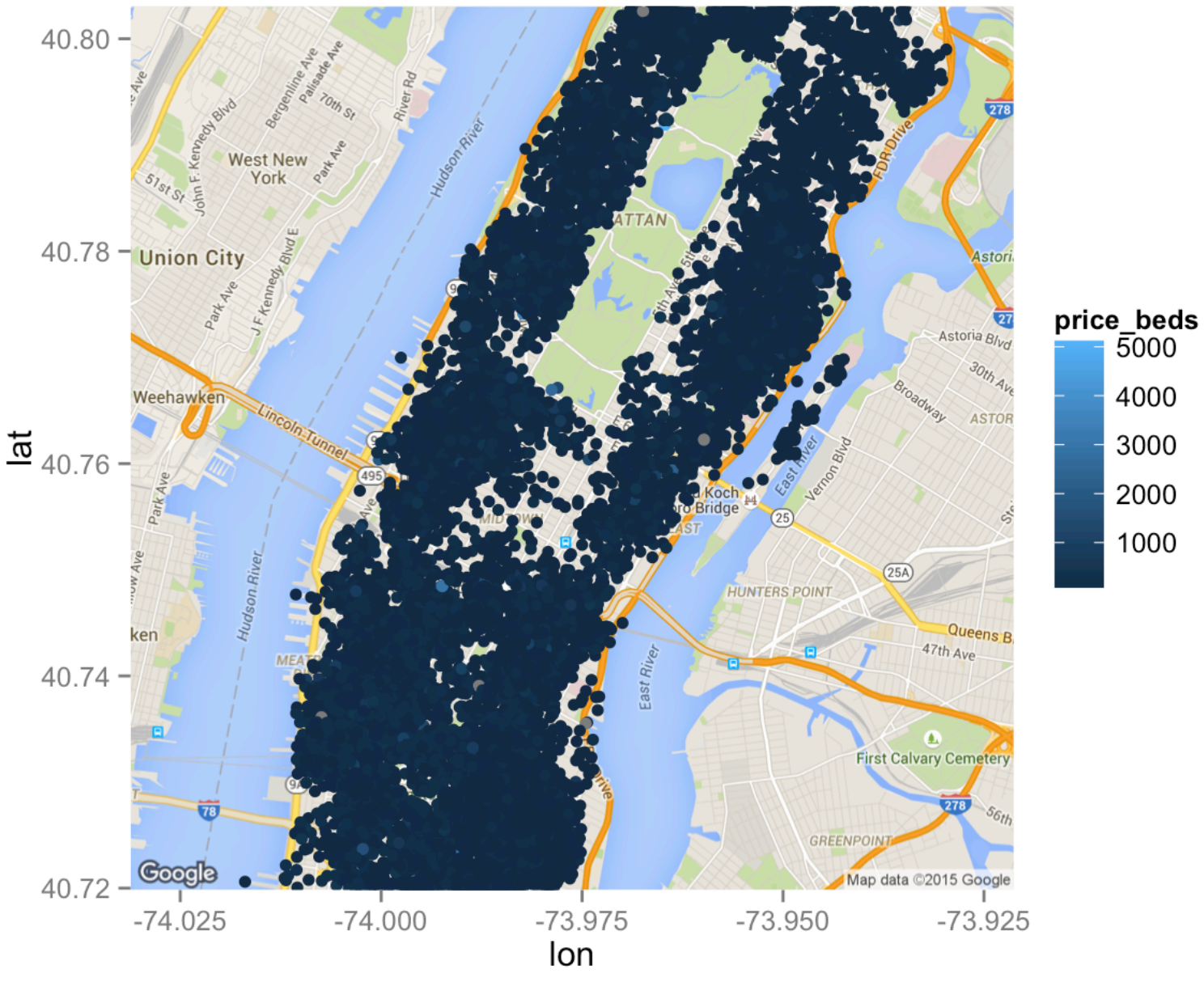
```
P + geom_point(data=listings, aes(x=longitude, y=latitude),size=2)
```



You will see this produces a map, however, also creates a warning about missing values - don't worry, this is just telling you that not all the rows of data in the data frame are visible on the map. You could make this go away if you change the zoom level - i.e. create a map with a greater geographic extent.

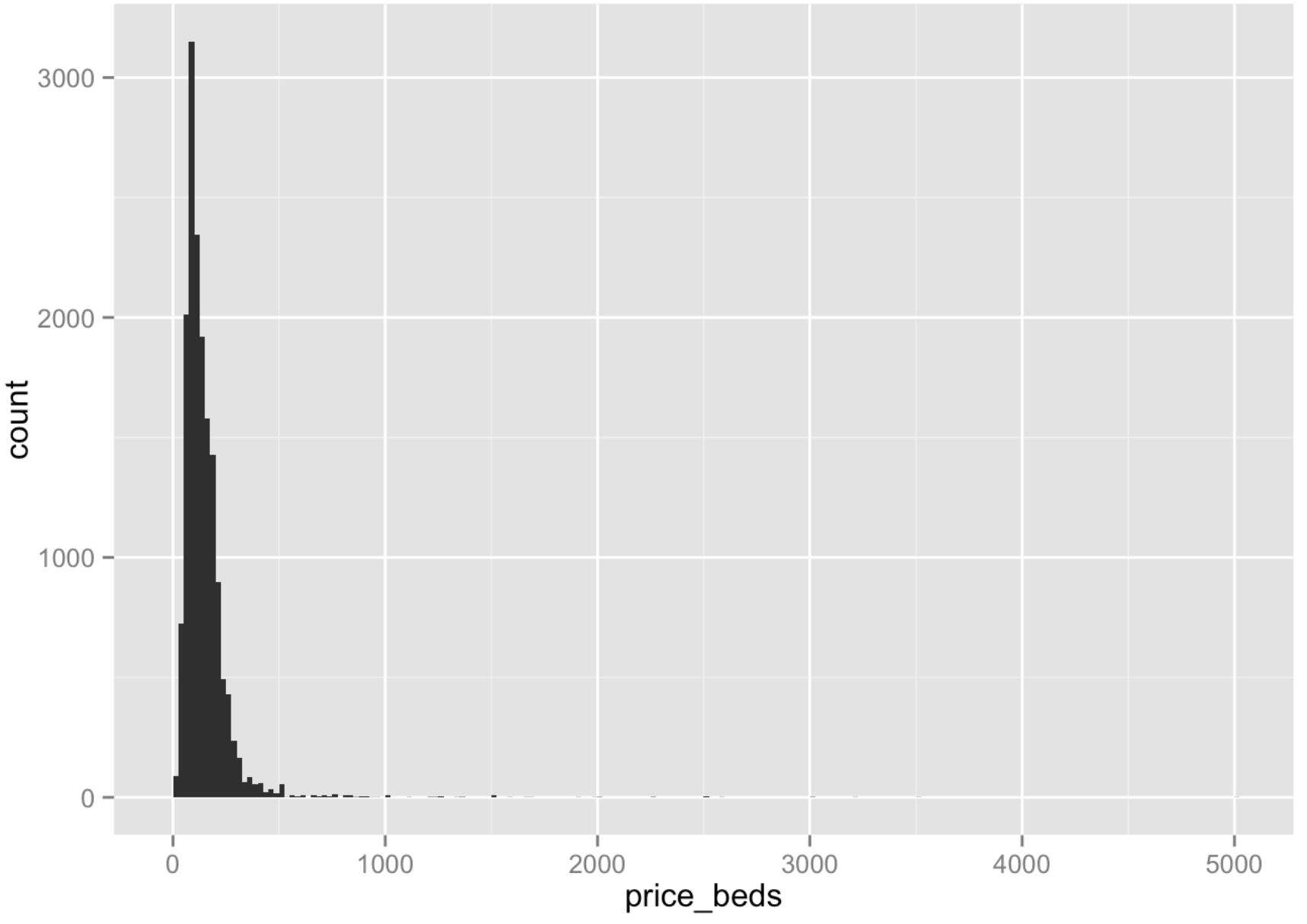
You can also adjust the colour of the points using the "colour" option.

```
P + geom_point(data=listings, aes(x=longitude, y=latitude, colour=price_beds),size
=2)
```



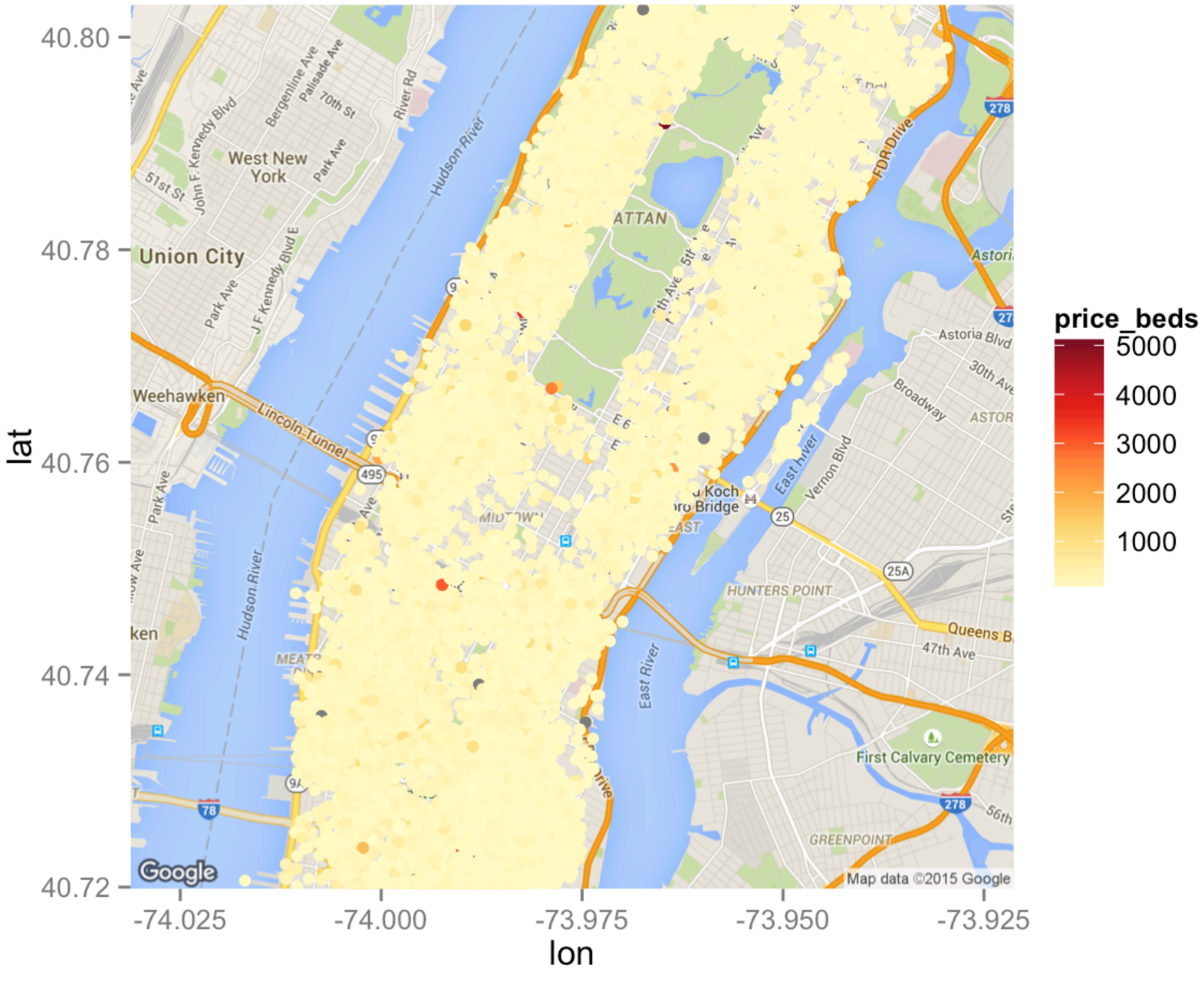
Because the price per bed is a continuous variable, the points are now scaled along a colour gradient from the highest to lowest values. However, this doesn't show you very much, as most of the values are clustered towards the bottom of the range. We can check this by plotting the values as a histogram. Each bar is a $25 bin.

```
qplot(price_beds, data=listings, geom="histogram",binwidth=25)
```



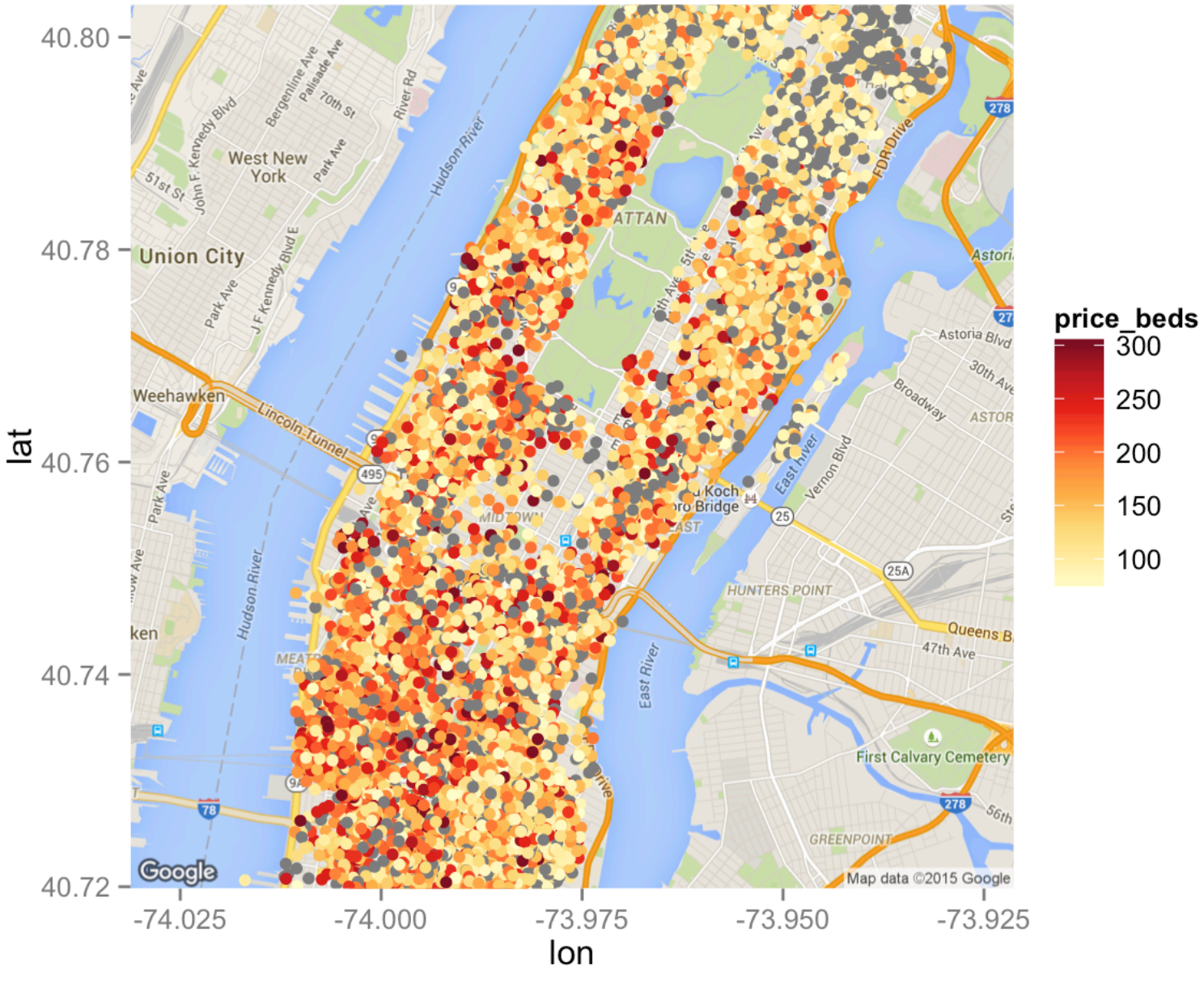
There are a number of ways in which we can adjust our map to make it more effective at communicating changes in price. First we will change the colour of the scale to one of the colour brewer pallets - for this we use the `scale_color_gradientn()` function.

```
P + geom_point(data=listings, aes(x=longitude, y=latitude, colour=price_beds),size
=2) + scale_color_gradientn(colours=brewer.pal(9,"YlOrRd"))
```



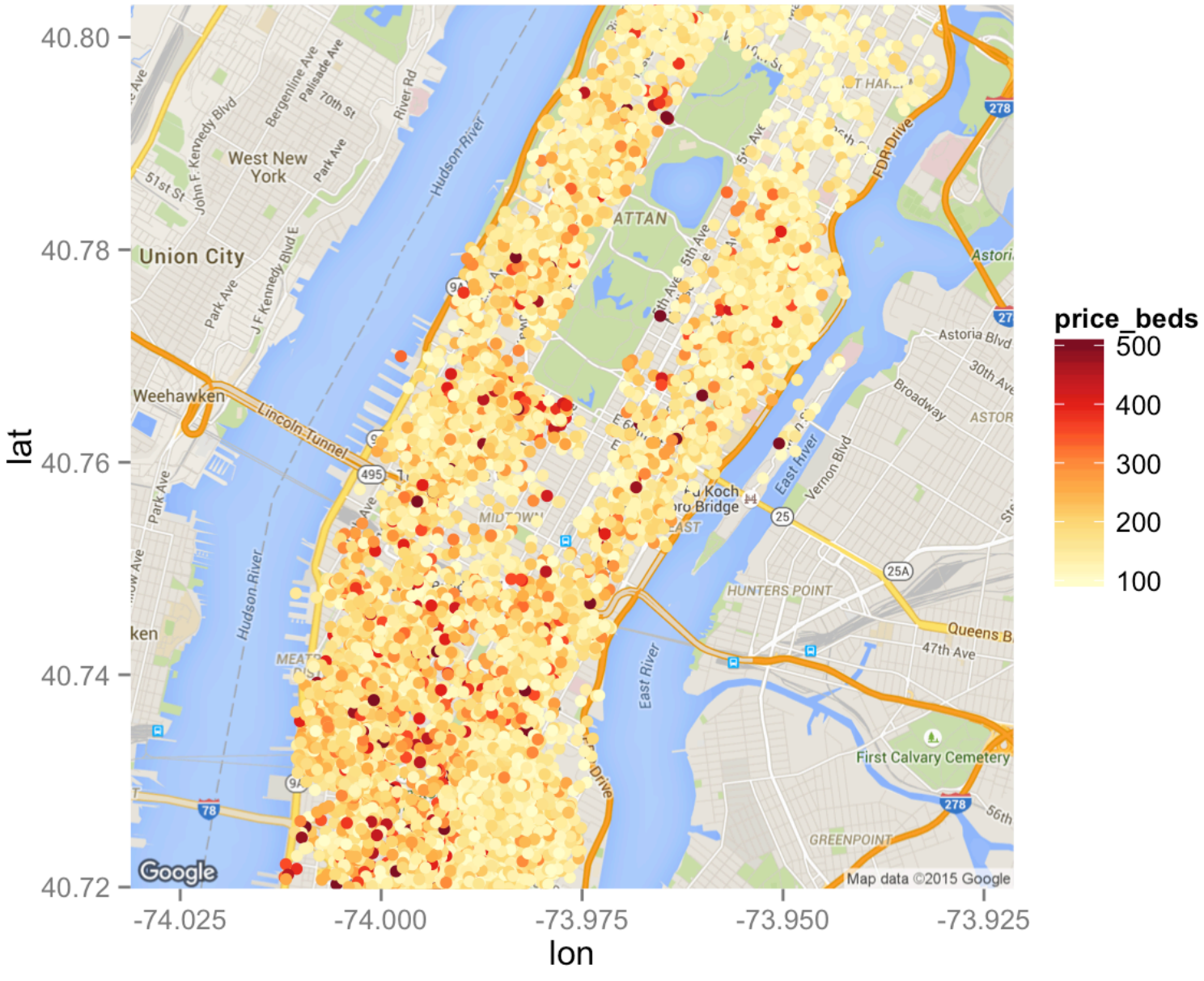
Although the colour has changed, we still have the issue with values being clustered at the end of the scale. However, there are a number of additional options that we can use to control for this. The first is “limits” which we can use to adjust the minimum and maximum value on the scale. Here we take the range 75-300.

```
P + geom_point(data=listings, aes(x=longitude, y=latitude, colour=price_beds),size
=2) + scale_color_gradientn(colours=brewer.pal(9,"YlOrRd"),limit=c(75,300))
```



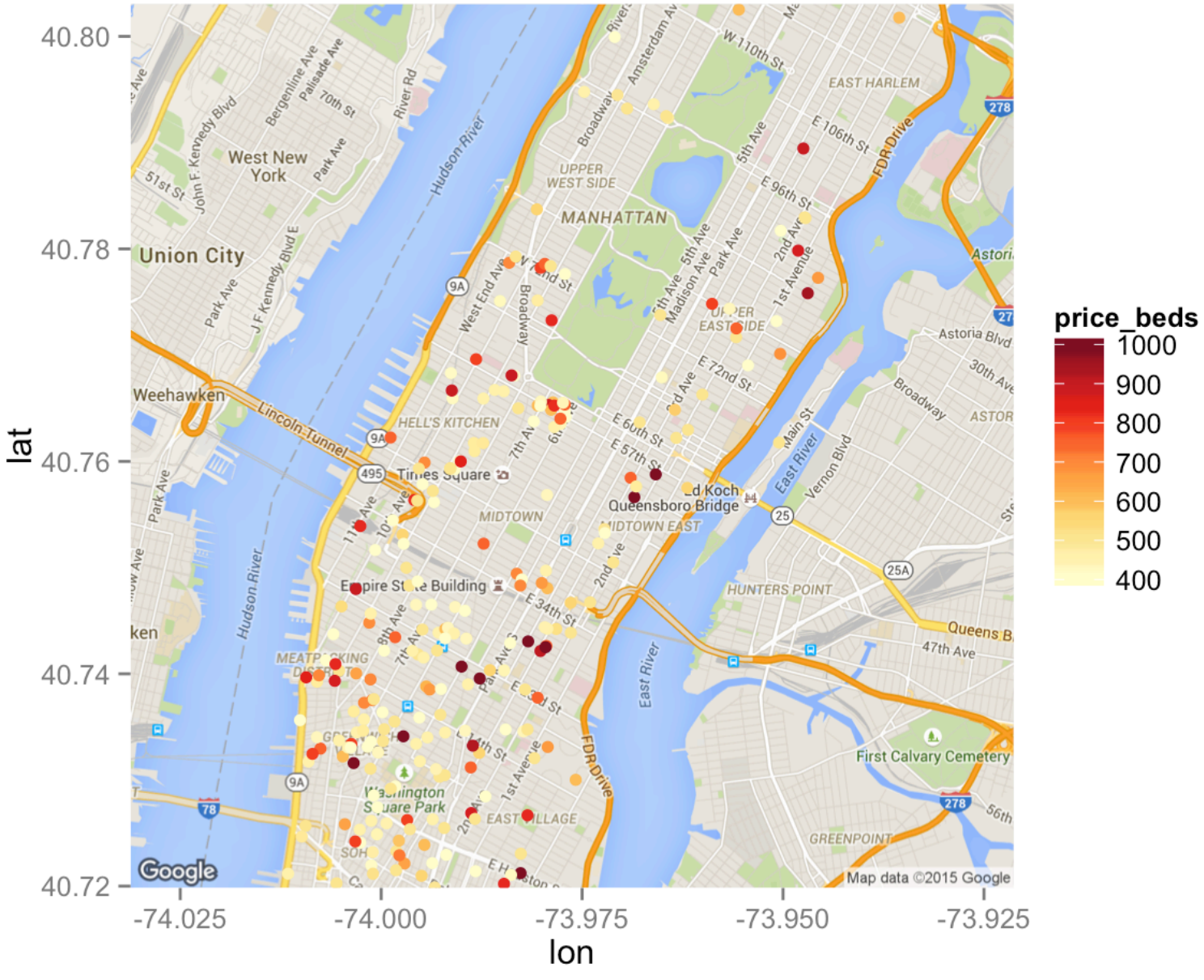
You may have noticed some grey points on the map - these are the properties with values that are outside the ranges specified. We can hide these using a further option “na.value” which you can either assign a colour, or as shown in this example, an `NA`, which makes them hidden.

```
P + geom_point(data=listings, aes(x=longitude, y=latitude, colour=price_beds),size
=2) + scale_color_gradientn(colours=brewer.pal(9,"YlOrRd"),limit=c(100,500),na.val
ue=NA)
```



We could for example use this technique to just plot the very expensive property, which we will define as between $400-1000.

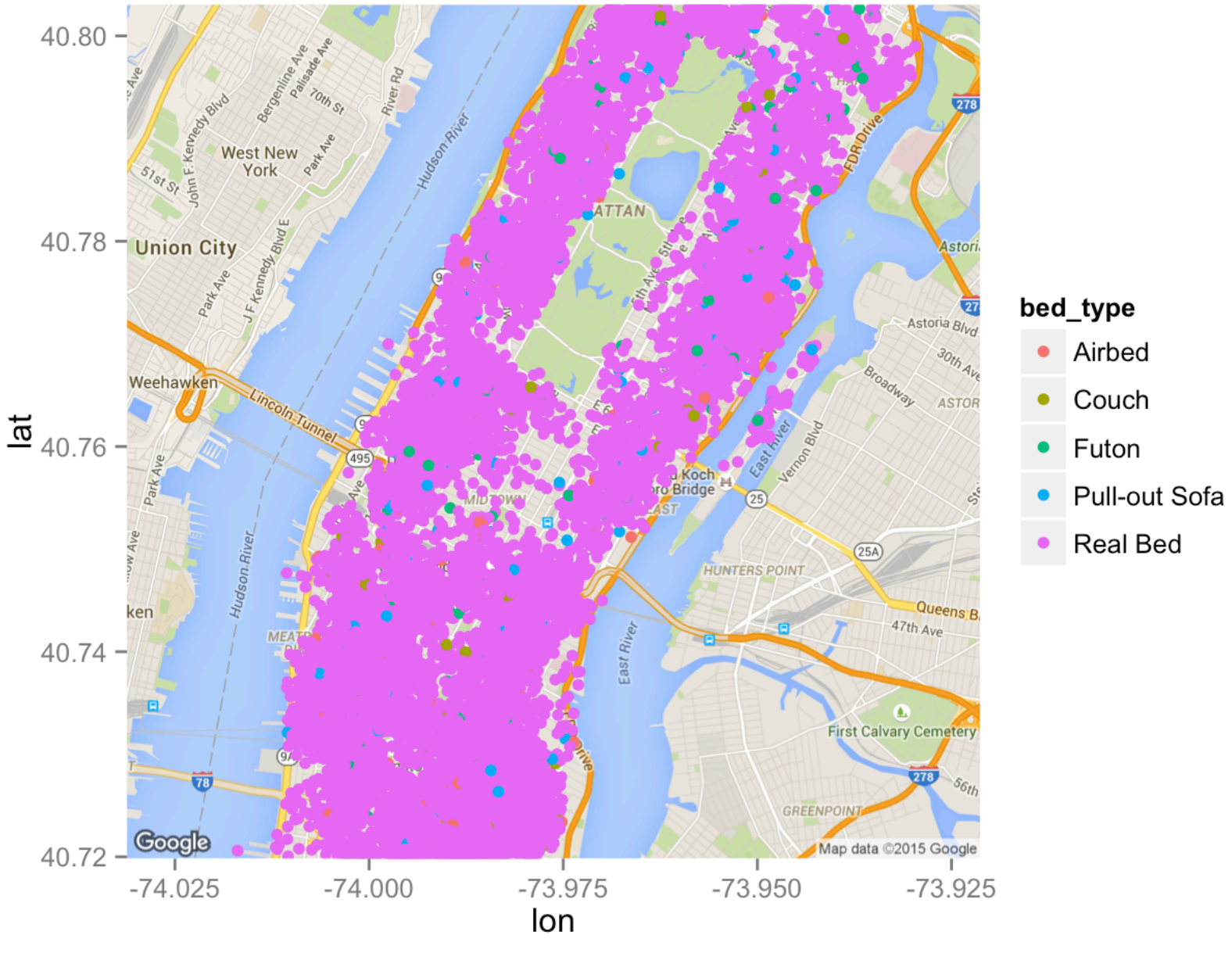
```
P + geom_point(data=listings, aes(x=longitude, y=latitude, colour=price_beds)) + s
cale_color_gradientn(colours=brewer.pal(9,"YlOrRd"),limit=c(400,1000),na.value=NA)
```



We can also use the “scale” option to change the size of the points. For example, we might want to colour the points by the bed type, but scale the points by the price.

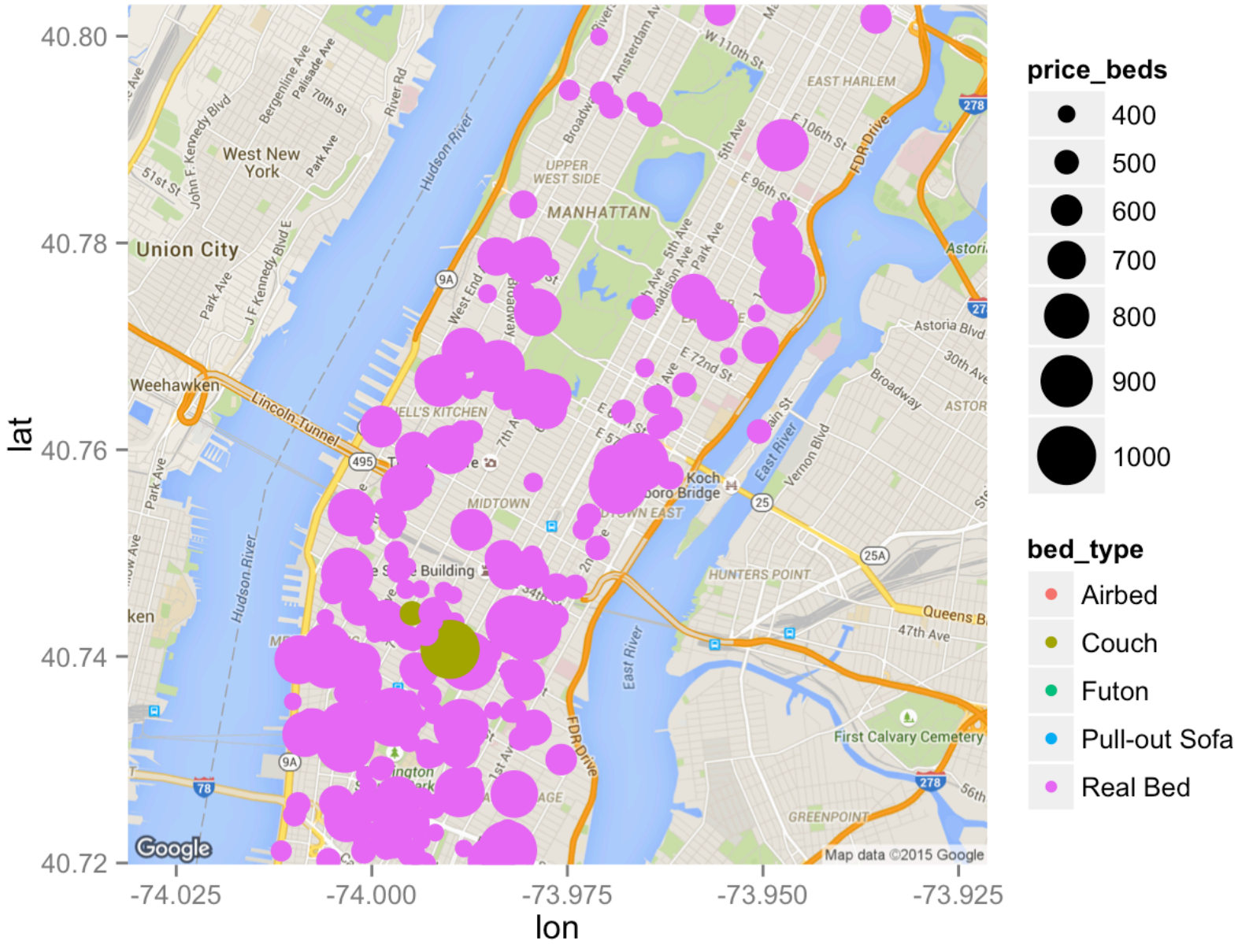
First of all we will just map the bed type - note that the variable which is attached to is a factor, so ggmap (like ggplot) displays this as a categorical value.

```
P + geom_point(data=listings, aes(x=longitude, y=latitude, colour=bed_type))
```



We can see that most of the AirBnB listings concern real beds; although there are other types across Manhattan. We can extend this plot to explore how these relate to price. Again, we will focus on more expensive property between $400 - $1000. For this we add the “size” parameter to the `aes()` and additionally, use a new function `scale_size()` which controls the range of point sizes used (in this case 3 to 10). You will see that there are two very expensive couches that can be rented!

```
P + geom_point(data=listings, aes(x=longitude, y=latitude, colour=bed_type,size=pr
ice_beds)) + scale_size(range = c(3, 10),limit=c(400,1000))
```



Plotting a Choropleth Map

For this map we will create a choropleth looking at outstanding mortgage debt within the London borough of Lewisham. The mortgage debt data are for postcode sectors and as such do not perfectly align with the borough boundaries. The analysis looks at Quarter 4 2014, and to simplify the plot, the totals are divided by one million.

First we will import the shapefile and calculate the new variable which places the “Q4_2014” in units of millions of pounds.

```
MD <- readOGR(".", "E09000023")
MD@data$Q4_2014_M <- MD@data$Q4_2014 / 1000000
```

One issue is that the shapefile is in OSGB projection which needs converting to WGS84 prior to plotting. This is easily achieved using a combination of the `CRS()` which sets the projection using the correct EPSG code (http://spatialreference.org/ref/epsg/), and `spTransform()` which does the transformation. After altering the projection, `fortify()` is used to make compatible with `ggmap()` along with a join, which appends the attributes back onto the data frame.

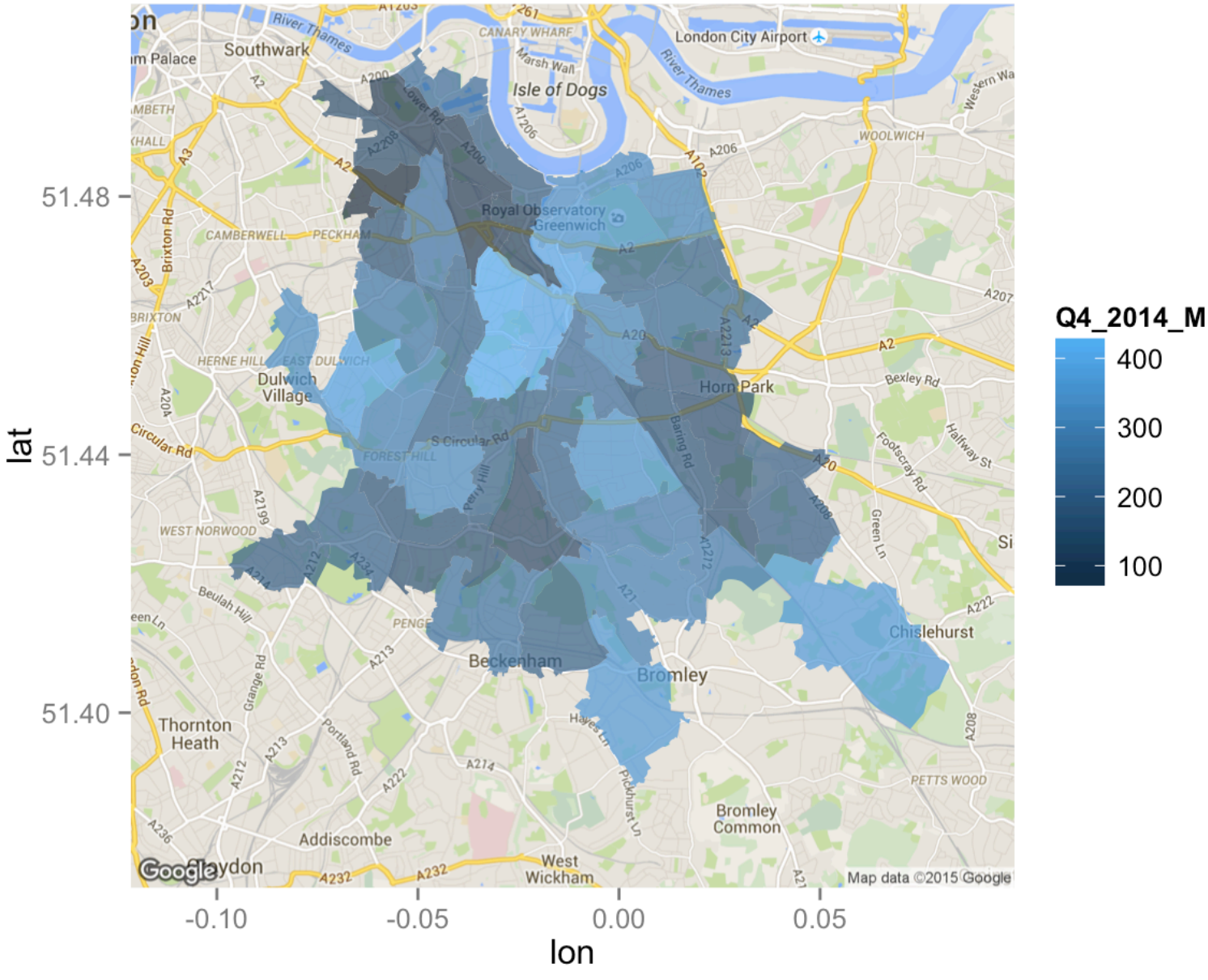
```
proj <- CRS("+init=epsg:4326") # WGS84
MD_WM <- spTransform(MD, proj)

MD_FF <- LSOA_FF <- fortify(MD_WM, region="Sector")
MD_FF <- merge(MD_FF, MD@data, by.x = "id", by.y = "Sector")
```

We can now set up a new map.

```
P <- ggmap(get_map("Lewisham",zoom=12,maptype = "roadmap"))
P
```

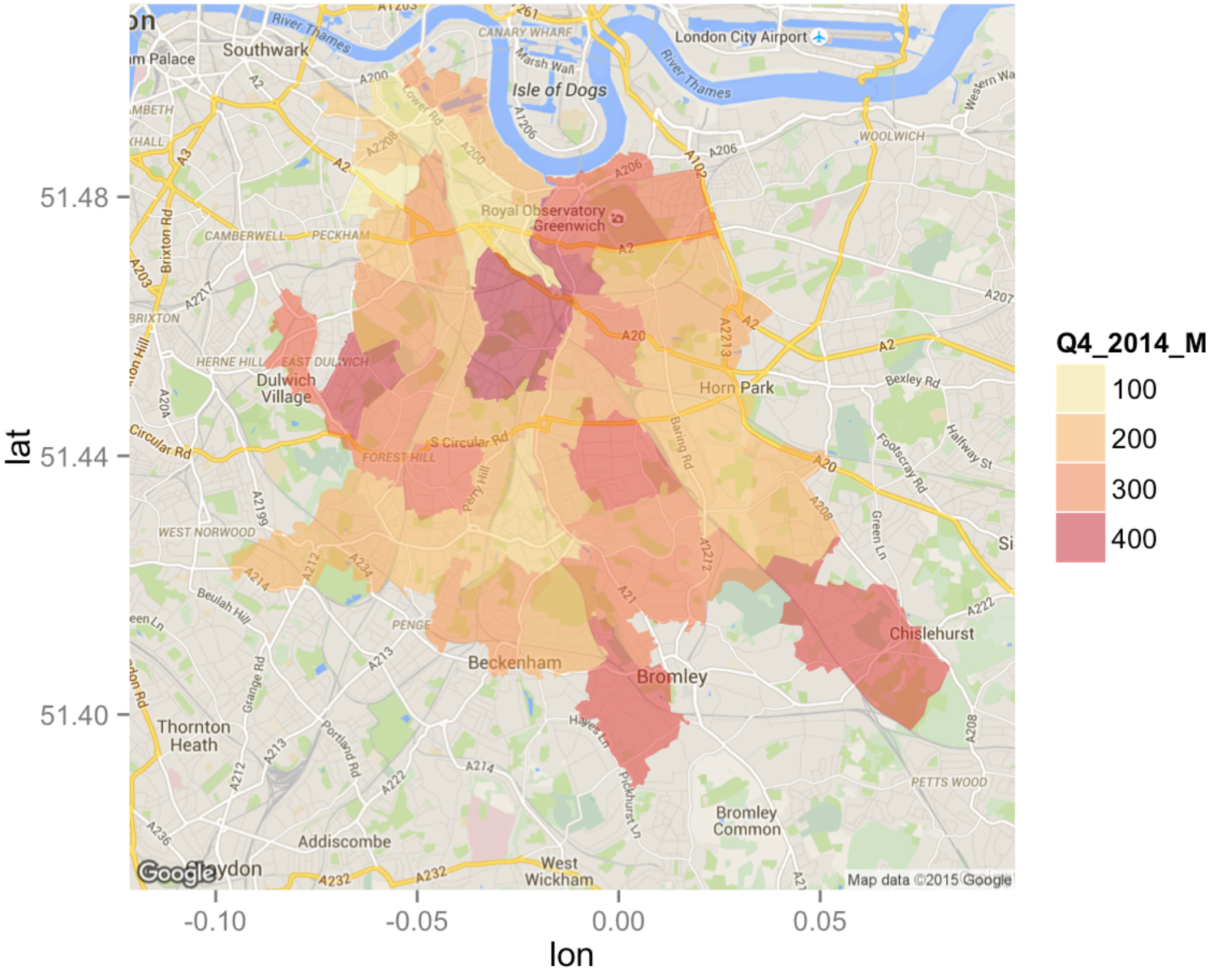
And then add the shaded polygons. The parameter “alpha” is used to adjust the transparency of the polygons.

```
P + geom_polygon(data = MD_FF,aes(x = long, y = lat, group = group, fill = Q4_2014
_M),alpha = 0.7)
```



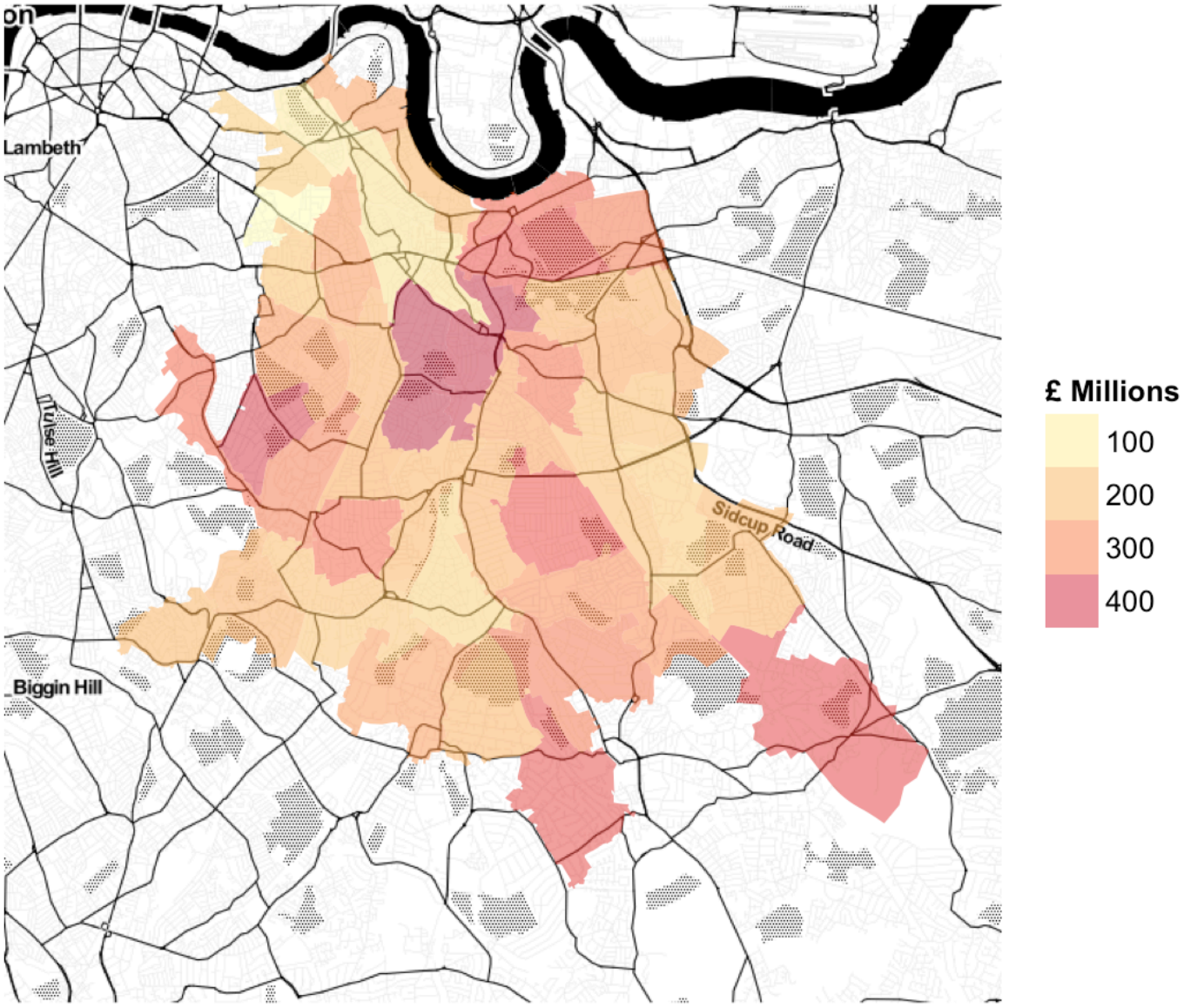
We can also switch to plotting using bins rather than on a continuous scale with the function `scale_fill_distiller()` - in this example we also swap the pallet colour.

```
P + geom_polygon(data = MD_FF,aes(x = long, y = lat, group = group, fill = Q4_2014
_M),alpha = 0.5) + scale_fill_distiller(palette = "YlOrRd")
```



It is sometimes best to use a background without much colour - which is illustrated in this final output with unwanted elements suppressed.

```
P <- ggmap(get_map("Lewisham",zoom=12,maptype = "toner"))
P + geom_polygon(data = MD_FF,aes(x = long, y = lat, group = group, fill = Q4_2014
_M),alpha = 0.5) + scale_fill_distiller(palette = "YlOrRd")  +  theme(axis.line =
element_blank(),
    axis.text = element_blank(),
    axis.title=element_blank(),
    axis.ticks = element_blank(),
    legend.key = element_blank(),
    panel.grid.major = element_blank(),
    panel.grid.minor = element_blank(),
    panel.border = element_blank(),
    panel.background = element_blank()) + labs(fill = "£ Millions")
```



Web Services

In addition to those web services which provide map content, we can also use a number of other services that conduct an analysis task. Two common tasks will be explored, including geocoding which provides a spatial coordinate reference for an object (in this case a postcode), and a routing example, which provides a time optimized route between two locations.

Postcode Geocoding

In this first example we will used the (http://postcodes.io/)http://postcodes.io/ (http://postcodes.io/) web service to geocode some postcodes. We will use a package called “jsonlite” which is used for sending and receiving “JSON” data from the web via HTTP requests. This is one example of many different geocoding services.

We can use the function `fromJSON()` to pull the results of a request form the postcodes.io service as follows.

```
postcode <- fromJSON("https://api.postcodes.io/postcodes/L693BX")
```

This has created an object called “postcode” which is a list of two items named “status” and “result”. It is possible to extract results from this list object, and an example for two items combined is as follows.

```
c(postcode$result$longitude,postcode$result$latitude)
```

```
## [1] -2.97339 53.38012
```

It is also possible to generate a data frame from all the results using the `unlist()` and `as.data.frame()` functions. If you print the data frame you will see all the results returned from the API. The row names identify the data item, with each appearing on a separate row.

```
postcode_DF <- as.data.frame(unlist(postcode))
postcode_DF
```

```
##                                                 unlist(postcode)
## status                                                       200
## result.postcode                                          L69 3BX
## result.quality                                                 1
## result.eastings                                           335347
## result.northings                                          387471
## result.country                                           England
## result.nhs_ha                                         North West
## result.longitude                                 -2.97339007774334
## result.latitude                                   53.3801193491673
## result.parliamentary_constituency       Liverpool, Riverside
## result.european_electoral_region                      North West
## result.primary_care_trust                              Liverpool
## result.region                                         North West
## result.lsoa                                        Liverpool 050I
## result.msoa                                         Liverpool 050
## result.incode                                                3BX
## result.outcode                                               L69
## result.admin_district                                  Liverpool
## result.parish                          Liverpool, unparished area
## result.admin_ward                                      Riverside
## result.ccg                                          NHS Liverpool
## result.nuts                                            Liverpool
## result.codes.admin_district                            E08000012
## result.codes.admin_county                              E99999999
## result.codes.admin_ward                                E05000907
## result.codes.parish                                    E43000166
## result.codes.ccg                                       E38000101
## result.codes.nuts                                          UKD72
```

Loops

Thus far we considered a single result, however, we can use a loop to repeat the geocoding call. First we need a list of postcodes.

```
pcd_list <- c("L693BX","L18JQ","L7 8XP","L181DG","L10AH")
```

We can use a `for()` statement to iterate through the list - essentially the values of 1 through to the length of the list (i.e 5) are assigned to the variable “i”; and for each assignment, the code within the curly brackets is run. In this example it prints the value of “pcd_list[i]”.

```
for( i in 1:length(pcd_list)){
  print(pcd_list[i])
}
```

```
## [1] "L693BX"
## [1] "L18JQ"
## [1] "L7 8XP"
## [1] "L181DG"
## [1] "L10AH"
```

We can now expand this statement, so rather than printing the value, a call is sent to the postcodes.io API. In order to do this we will also use another function called `paste()` which joins two strings together. For example…

```
string1 <- "text one"
string2 <- "text two"
paste(string1,string2)
```

```
## [1] "text one text two"
```

You will see that a space has been placed between the two string items. An alternative is `paste0()` which just concatenates both strings together without the space.

```
paste0(string1,string2)
```

```
## [1] "text onetext two"
```

We can now use this within the loop to paste a selected postcode on the end of the URL string which sends the call to the postcodes.io API.

```
for( i in 1:length(pcd_list)){
  pcd <- pcd_list[i]
  postcodes_io <- "https://api.postcodes.io/postcodes/"
  postcodes_io_results <- fromJSON(paste0(postcodes_io,pcd))
  print(c(pcd,postcodes_io_results$result$longitude,postcodes_io_results$result$la
titude))
}
```

```
## [1] "L693BX"           "-2.97339007774334" "53.3801193491673"
## [1] "L18JQ"           "-2.9873179423706" "53.404454914063"
## [1] "L7 8XP"           "-2.96480927067585" "53.4095109071453"
## [1] "L181DG"           "-2.91981378795851" "53.3865128476184"
## [1] "L10AH"           "-2.9805490691596" "53.395953227081"
```

This has simply printed the results to the terminal, however, it is also possible to store the results in a new list. First we create an empty list “output_PCD” to collect the results. Within the loop we assign each of the results generated by the `fromJSON()` function to a new variable called “tmp”. We then use another function called `rbind()` to append “tmp” to the list - which is replaced each time the loop is run.

```
output_PCD <- list()

for( i in 1:length(pcd_list)){
  pcd <- pcd_list[i]
  postcodes_io <- "https://api.postcodes.io/postcodes/"
  postcodes_io_results <- fromJSON(paste0(postcodes_io,pcd))
  tmp <- c(pcd,postcodes_io_results$result$longitude,postcodes_io_results$result$l
atitude)
  output_PCD <- rbind(output_PCD,tmp)
}
```

We can then convert this to a data frame and rename the columns using `colnames()`.

```
output_PCD <- data.frame(output_PCD)
colnames(output_PCD) <- c("Postcode","longitude","latitude")
```

Batch Geocoding

Postcodes.io also enables a single search to be made with multiple postcodes. This requires us to use the `POST()` function which sends an HTTP request using the post method. The API accepts an input in JSON format which can be achieved using the "body" and "encode" options of the `POST()` function.

```
req <- POST("https://api.postcodes.io/postcodes/", body = list(postcodes=pcd_list)
,encode="json")
```

Once the request is completed, the output can be assigned to a new object using `content()`.

```
all <- content(req)
```

This creates a list, and the components of the list can be accessed as follows - using the double square bracket to refer to the list item. In this instance, the value of the eastings result is printed for the first postcode in the list.

```
all$result[[1]]$result$eastings
```

```
## [1] 335347
```

Reverse Geocoding

Another feature of many geocoding services, including postcodes.io, is to find "places" (in this case postcodes) near a given latitude and longitude. This requires a proximity search which is completed on the server, returning the results to the client - which in this case is R. This is achieved in a fairly simple way.

```
rev_geocode <- fromJSON("https://api.postcodes.io/postcodes?lon=-2.97339&lat=53.38
012")
rev_geocode_DF <- data.frame(rev_geocode$result)
head(rev_geocode_DF)
```

```
##   postcode quality eastings northings country     nhs_ha longitude
## 1  L67 1AT       1   335347    387471 England North West  -2.97339
## 2  L67 1AX       1   335347    387471 England North West  -2.97339
## 3  L67 1BD       1   335347    387471 England North West  -2.97339
## 4  L69 1SN       1   335347    387471 England North West  -2.97339
## 5  L69 1WL       1   335347    387471 England North West  -2.97339
## 6  L69 1XG       1   335347    387471 England North West  -2.97339
##   latitude parliamentary_constituency european_electoral_region
## 1 53.38012       Liverpool, Riverside                North West
## 2 53.38012       Liverpool, Riverside                North West
## 3 53.38012       Liverpool, Riverside                North West
## 4 53.38012       Liverpool, Riverside                North West
## 5 53.38012       Liverpool, Riverside                North West
## 6 53.38012       Liverpool, Riverside                North West
##   primary_care_trust     region           lsoa          msoa incode
## 1          Liverpool North West Liverpool 050I Liverpool 050    1AT
## 2          Liverpool North West Liverpool 050I Liverpool 050    1AX
## 3          Liverpool North West Liverpool 050I Liverpool 050    1BD
## 4          Liverpool North West Liverpool 050I Liverpool 050    1SN
## 5          Liverpool North West Liverpool 050I Liverpool 050    1WL
## 6          Liverpool North West Liverpool 050I Liverpool 050    1XG
##   outcode   distance admin_district                    parish
## 1     L67 0.07261792      Liverpool Liverpool, unparished area
## 2     L67 0.07261792      Liverpool Liverpool, unparished area
## 3     L67 0.07261792      Liverpool Liverpool, unparished area
## 4     L69 0.07261792      Liverpool Liverpool, unparished area
## 5     L69 0.07261792      Liverpool Liverpool, unparished area
## 6     L69 0.07261792      Liverpool Liverpool, unparished area
##   admin_county admin_ward           ccg      nuts codes.admin_district
## 1           NA  Riverside NHS Liverpool Liverpool            E08000012
## 2           NA  Riverside NHS Liverpool Liverpool            E08000012
## 3           NA  Riverside NHS Liverpool Liverpool            E08000012
## 4           NA  Riverside NHS Liverpool Liverpool            E08000012
## 5           NA  Riverside NHS Liverpool Liverpool            E08000012
## 6           NA  Riverside NHS Liverpool Liverpool            E08000012
##   codes.admin_county codes.admin_ward codes.parish codes.ccg codes.nuts
## 1          E99999999        E05000907    E43000166 E38000101      UKD72
## 2          E99999999        E05000907    E43000166 E38000101      UKD72
## 3          E99999999        E05000907    E43000166 E38000101      UKD72
## 4          E99999999        E05000907    E43000166 E38000101      UKD72
## 5          E99999999        E05000907    E43000166 E38000101      UKD72
## 6          E99999999        E05000907    E43000166 E38000101      UKD72
```

Routing

A feature of many web mapping sites is the ability to calculate an optimal route between two locations controlling for factors such as road access, one way streets and travel mode. Google make this very simple, combining route calculation with geocoding. For example, the following returns a route between two postcodes using the `GET()` function.

```
route <- GET("https://maps.googleapis.com/maps/api/directions/json?origin='L693BX'
&destination='L18JQ'")
route_content <- content(route)
```

Although a little complicated, it is possible to parse these results into a single data frame.

```
#Find out how may steps the route comprises
r_length <- length(route_content$routes[[1]]$legs[[1]]$steps)

#Build a loop to extract all steps
steps <- data.frame()
for (i in 1:r_length){
tmp <- cbind(data.frame(route_content$routes[[1]]$legs[[1]]$steps[[i]]$start_locat
ion),data.frame(route_content$routes[[1]]$legs[[1]]$steps[[i]]$end_location))
steps <- rbind(steps,tmp)
}
colnames(steps) <- c("start_lat","start_lon","end_lat","end_lon")
```

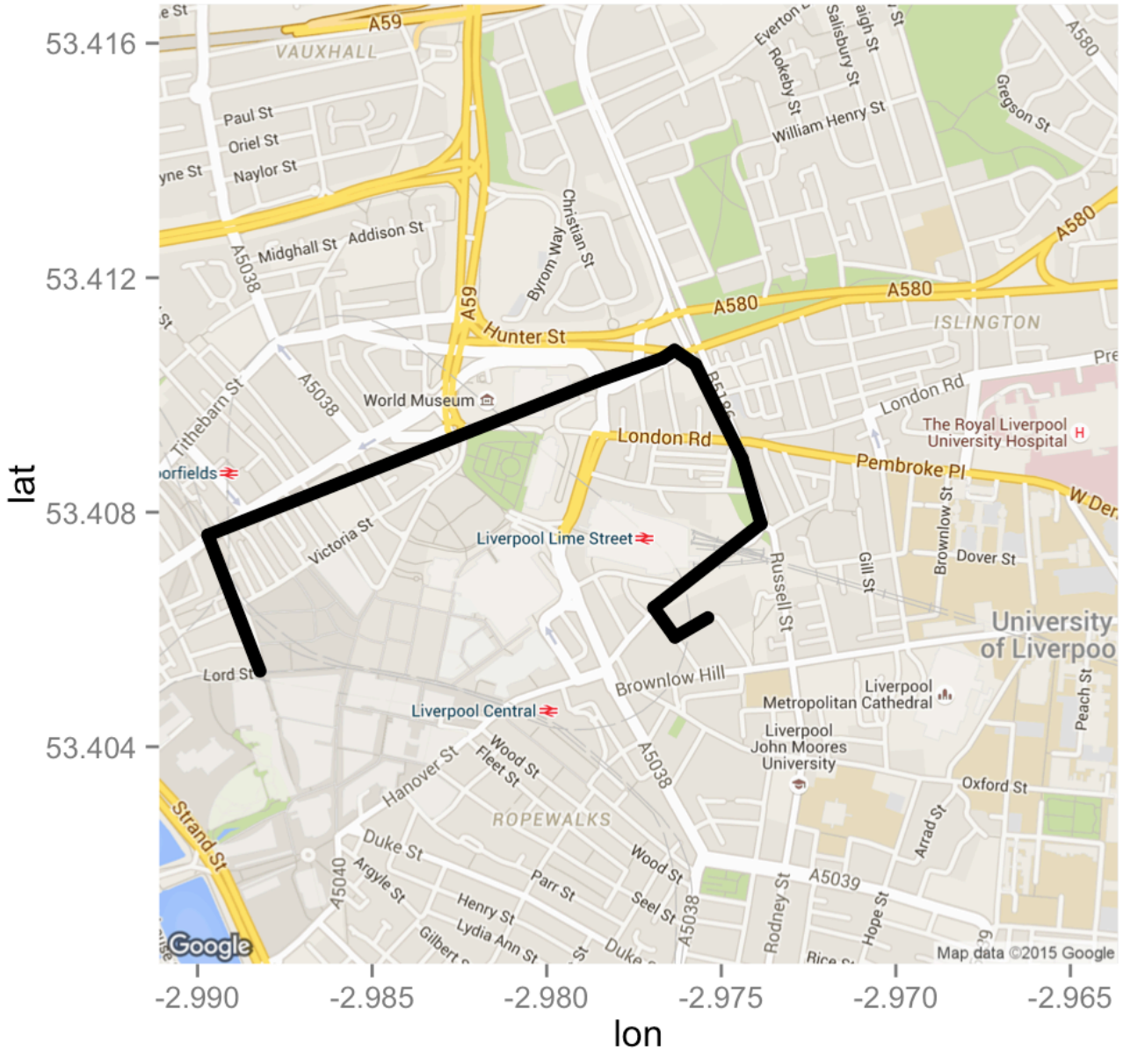
If we have a look at the data we have created you will see that the end latitude and longitude are the the same as the start in the next line.

```
steps
```

```
##    start_lat start_lon  end_lat   end_lon
## 1   53.40620 -2.975385 53.40587 -2.976326
## 2   53.40587 -2.976326 53.40637 -2.976932
## 3   53.40637 -2.976932 53.40780 -2.973862
## 4   53.40780 -2.973862 53.40891 -2.974402
## 5   53.40891 -2.974402 53.41052 -2.975742
## 6   53.41052 -2.975742 53.41076 -2.976341
## 7   53.41076 -2.976341 53.41063 -2.976676
## 8   53.41063 -2.976676 53.41023 -2.978557
## 9   53.41023 -2.978557 53.40761 -2.989706
## 10  53.40761 -2.989706 53.40529 -2.988212
```

Now we have the steps, we can use `geom_leg()` to visualize these results.

```
map<- get_map(c(mean(steps$start_lon),mean(steps$start_lat)),zoom=15,maptype = "ro
admap")
P<- ggmap(map)
P + geom_leg(aes(x = start_lon, y = start_lat, xend = end_lon, yend = end_lat), si
ze = 2,data=steps)
```



There is also another way we can perform the same analysis with much simpler code, as ggmap has routing integrated into a function called `route()`.

```
route_df <- route("L693BX", "L18JQ", structure = 'route') #Creates a route data fr
ame
route_df
```

```
##       m    km     miles seconds   minutes       hours leg       lon
## 1    72 0.072 0.0447408      12 0.2000000 0.003333333   1 -2.975385
## 2    69 0.069 0.0428766      34 0.5666667 0.009444444   2 -2.976326
## 3   261 0.261 0.1621854      58 0.9666667 0.016111111   3 -2.976932
## 4   131 0.131 0.0814034      38 0.6333333 0.010555556   4 -2.973862
## 5   200 0.200 0.1242800      44 0.7333333 0.012222222   5 -2.974402
## 6    48 0.048 0.0298272      20 0.3333333 0.005555556   6 -2.975742
## 7    27 0.027 0.0167778      10 0.1666667 0.002777778   7 -2.976341
## 8   143 0.143 0.0888602      26 0.4333333 0.007222222   8 -2.976676
## 9   862 0.862 0.5356468     146 2.4333333 0.040555556   9 -2.978557
## 10 278 0.278 0.1727492     113 1.8833333 0.031388889  10 -2.989706
## 11  NA    NA        NA      NA        NA          NA  NA -2.988212
##         lat
## 1  53.40620
## 2  53.40587
## 3  53.40637
## 4  53.40780
## 5  53.40891
## 6  53.41052
## 7  53.41076
## 8  53.41063
## 9  53.41023
## 10 53.40761
## 11 53.40529
```

We can then visualize these data.

```
map<- get_map(c(mean(steps$start_lon),mean(steps$start_lat)),zoom=15,maptype = "ro
admap")
P<- ggmap(map)
P + geom_path(aes(x = lon, y = lat),  colour = 'red', size = 2,data = route_df, li
neend = 'round')
```


lat
53.416
53.412
53.408
53.404
lon
-2.990
-2.985
-2.980
-2.975
-2.970
-2.965
A59
VAUXHALL
Paul St
Oriel St
Naylor St
Midghall St
Addison St
A5038
Byrom Way
Christian St
Everton
Rokeby St
Salisbury St
William Henry St
Gregson St
A580
A580
A580
ISLINGTON
Hunter St
World Museum
Tithebarn St
London Rd
The Royal Liverpool University Hospital
Pembroke Pl
Liverpool Lime Street
Victoria St
Dover St
Brownlow St
Gill St
Russell St
University of Liverpoo
Lord St
Brownlow Hill
Liverpool Central
Liverpool Metropolitan Cathedral
Liverpool John Moores University
Peach St
Oxford St
Arrad St
Hanover St
Wood St
Fleet St
A5038
ROPEWALKS
Strand St
Duke St
Argyle St
A5040
Parr St
Henry St
Lydia Ann St
Gilbert
Wood St
Seel St
A5039
Rodney St
Hope St
Google
Map data ©2015 Google