Introducing R for Mapping and Visualization

Alex Singleton

1 November 2015

Learning Objectives

1. Loading packages in R
2. Basic graphing with ggplot2
3. Importing spatial data into R
4. Mapping with base R
5. Mapping with ggplot2

R Packages

Packages within R enable us to extend the functionality of "base R". New functionality can be added by loading "packages" which are blocks of code containing new functions. These can be written by anyone, and shared on either CRAN (The Comprehensive R Archive Network) (https://cran.r-project.org/) or increasingly, directly from the package developers using code sharing platforms such as github.

For this practical we will be using two packages: `rgdal` which contains functions that will read a shapefile into R, and `ggplot2` which is very popular package for visualization.

The first stage is to install the packages. After this has complete, you then need load them. Package loading is required for every new R session.

```
#Installing packages
install.packages("rgdal", depend = TRUE)
install.packages("ggplot2", depend = TRUE)
install.packages("classInt", depend = TRUE)
install.packages("RColorBrewer", depend = TRUE)
install.packages("maptools", depend = TRUE)
```

```
#Load packages
library("rgdal")
```

```
## Loading required package: sp
## rgdal: version: 1.0-4, (SVN revision 548)
##  Geospatial Data Abstraction Library extensions to R successfully loaded
##  Loaded GDAL runtime: GDAL 1.11.2, released 2015/02/10
##  Path to GDAL shared files: /Library/Frameworks/R.framework/Versions/3.2/Resour
ces/library/rgdal/gdal
##  Loaded PROJ.4 runtime: Rel. 4.9.1, 04 March 2015, [PJ_VERSION: 491]
##  Path to PROJ.4 shared files: /Library/Frameworks/R.framework/Versions/3.2/Reso
urces/library/rgdal/proj
##  Linking to sp version: 1.1-1
```

```
library("ggplot2")
library("RColorBrewer")
library("maptools")
```

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

```
library("classInt")
```

Basic Graphing with ggplot2

> We are going to use the data created in the last practical, so copy the "census_small.csv" into the folder you are using for this new pratical.

First read in the data we will used for the practical.

```
census <- read.csv("census_small.csv")
```

This should look as follows…

```
head(census)
```

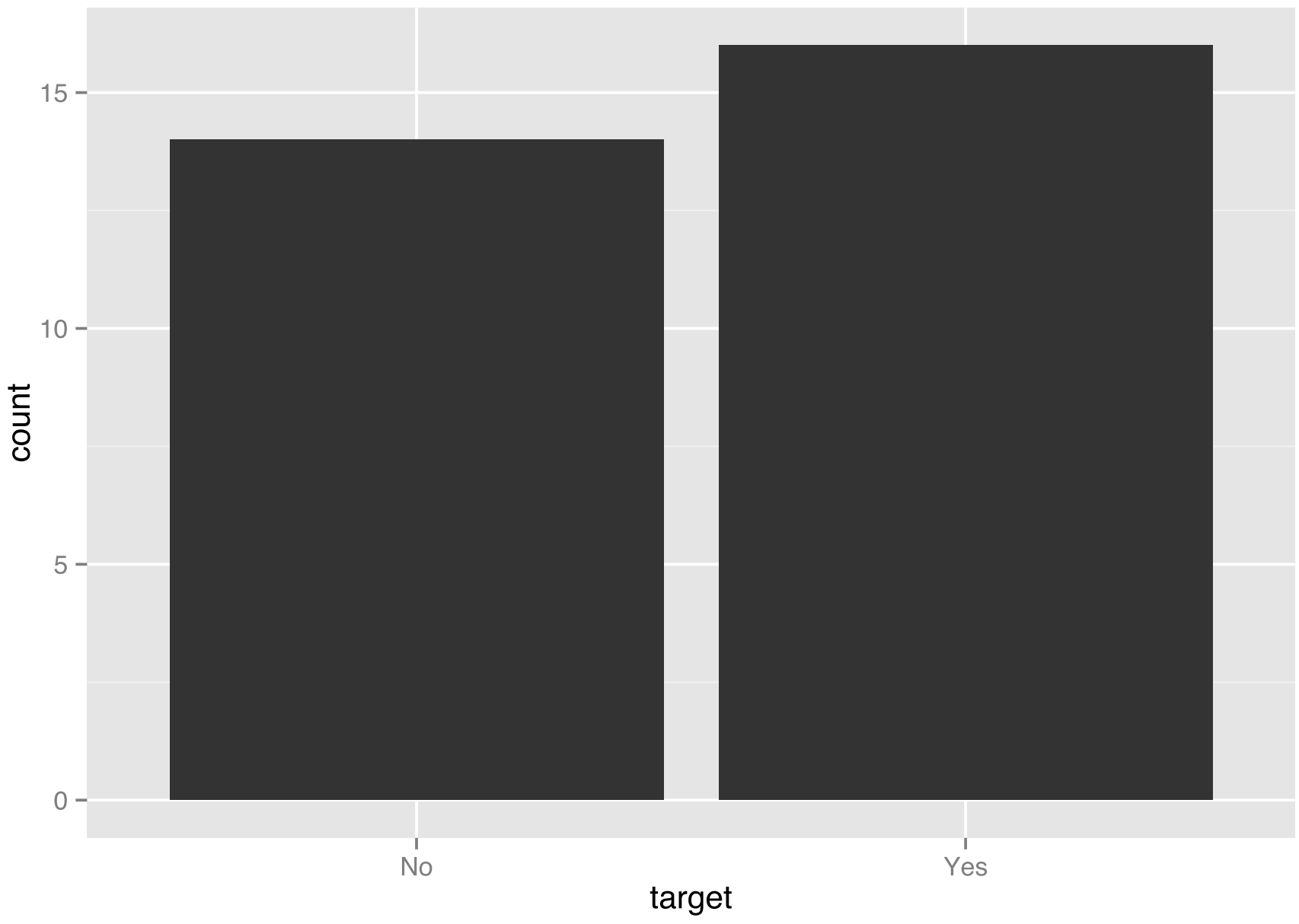
```
##   X      Code                    Ward PCT_Good_Health
## 1 1 E05000886 Allerton and Hunts Cross        48.97327
## 2 2 E05000887                Anfield        42.20538
## 3 3 E05000888             Belle Vale        40.84911
## 4 4 E05000889                Central        58.62832
## 5 5 E05000890              Childwall        51.90538
## 6 6 E05000891                 Church        53.39201
##   PCT_Higher_Managerial target PCT_Social_Rented_Households
## 1             10.091491     No                    13.005189
## 2              2.912621    Yes                    22.772576
## 3              3.931920    Yes                    42.555119
## 4              7.019923     No                    18.363917
## 5             10.787704     No                     6.937488
## 6             17.437790     No                     3.025153
```

The ggplot2 library provides a range of functions that make graphing and visualization of your data both visually appealing and simple to implement. There are two ways in which graphs can be created in ggplot2, the first is `ggplot()` which we will discuss later, and the second is `qlot()`, which has a simplified syntax.

Bar Charts

We can first create a bar chart using the factor column ("target") of the data frame object "census". The "geom" attribute is telling `qplot` what sort of plot to make. If you remember from the last practical, the target variable were wards within Liverpool where the percentage of people in good health was less than the city mean.

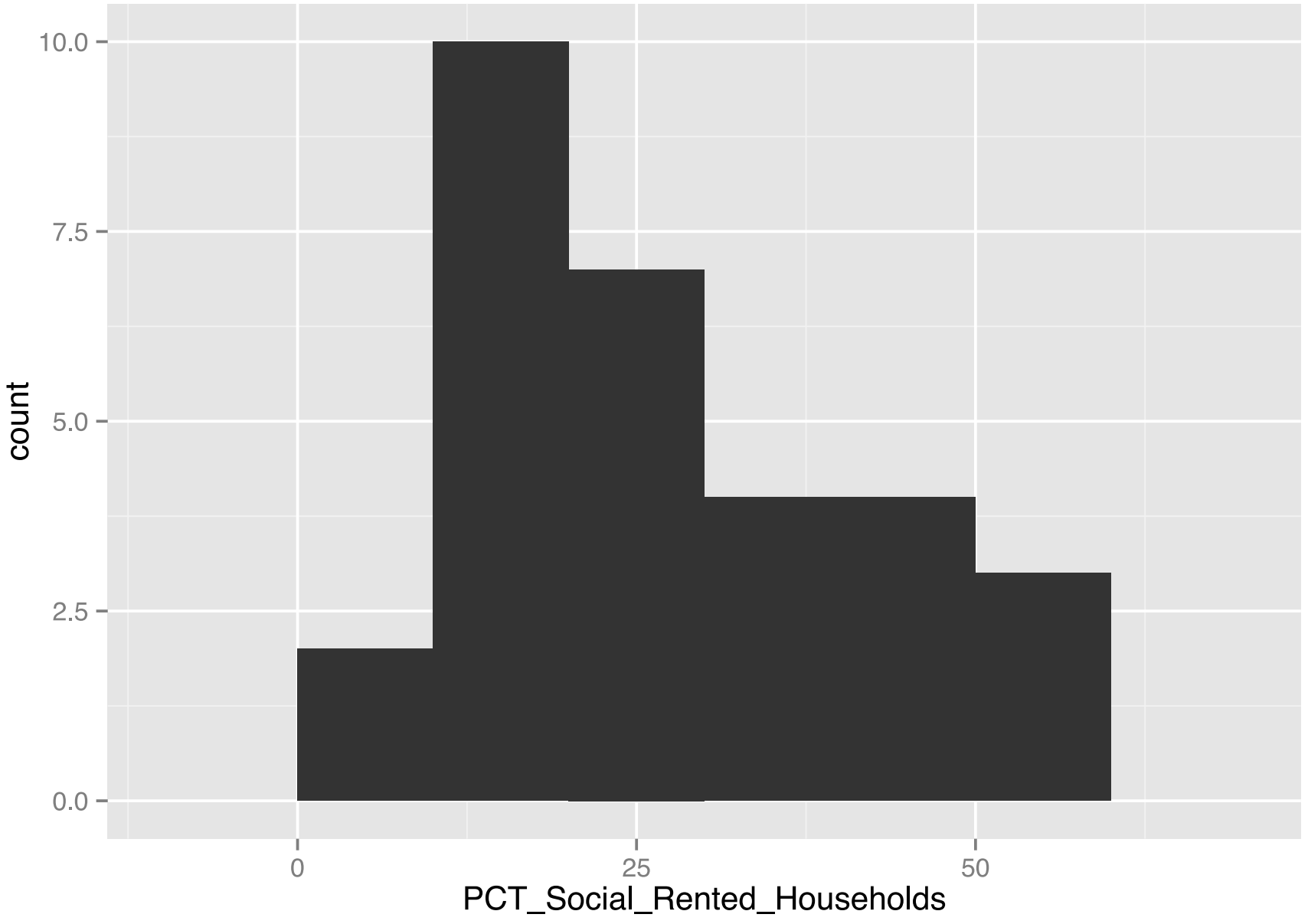
```
qplot(target,  data=census, geom="bar")
```



Histogram

We can create a histogram by changing the "geom" and variable being plotted. Try adjusting the bin width, which alters the bins into which the values of the "PCT_Social_Rented_Households" column are aggregated.

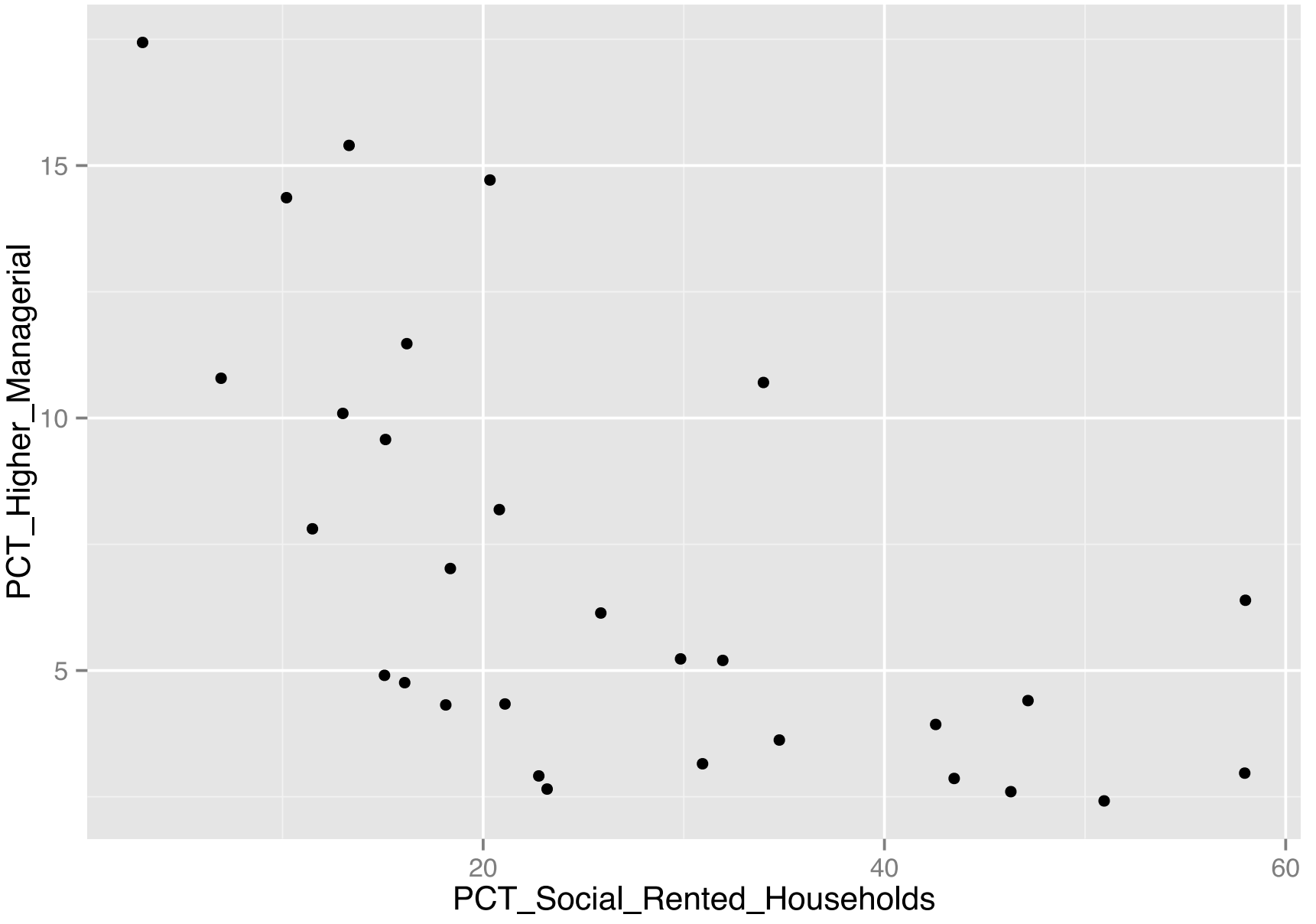
```
qplot(PCT_Social_Rented_Households, data=census, geom="histogram",binwidth=10)
```



Scatterplot

Another very common type of graph is a scatterplot which will typically plot the values of two continuous variables against one another on the x and y axis of the graph. This graph looks at the relationship between the percentage of people in socially rented housing, and those who are occupied in higher managerial roles. The default plot type is a scatterplot, so note in the next couple of examples we do not include `geom = "point"` , however, this could be added and would return the same result (try it!)

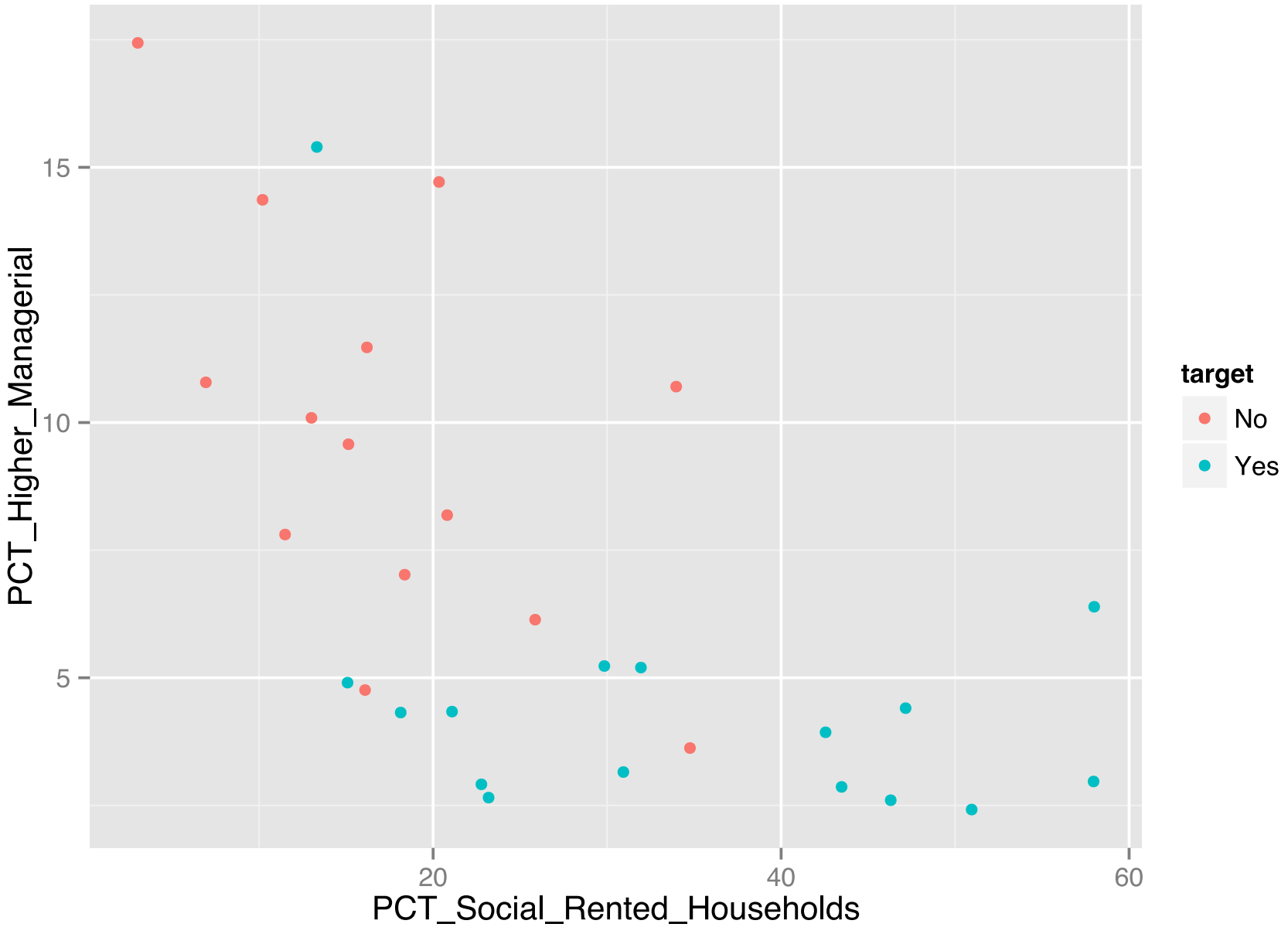
```
qplot(PCT_Social_Rented_Households, PCT_Higher_Managerial, data = census)
```



Adding colours or shapes

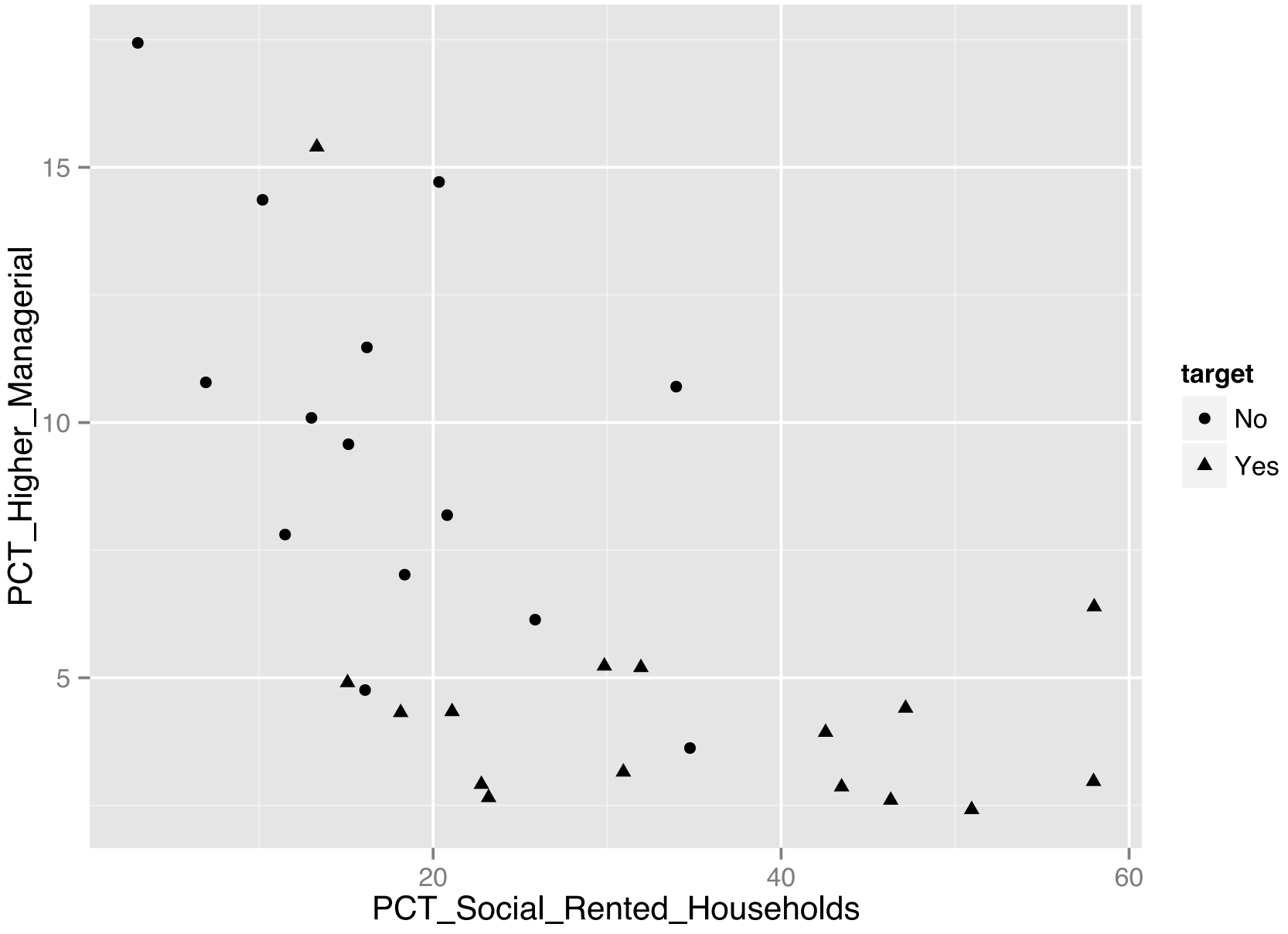
In the previous graph, all the points were black, however, if we swap these out for colour, we can highlight a factor variable, which in this case is the “target” column.

```
qplot(PCT_Social_Rented_Households, PCT_Higher_Managerial, data = census,colour=ta
rget)
```



Alternatively, you can also use “shape” to keep the points as black, but alter their shape by the factor variable.

```
qplot(PCT_Social_Rented_Households, PCT_Higher_Managerial, data = census,shape=tar
get)
```

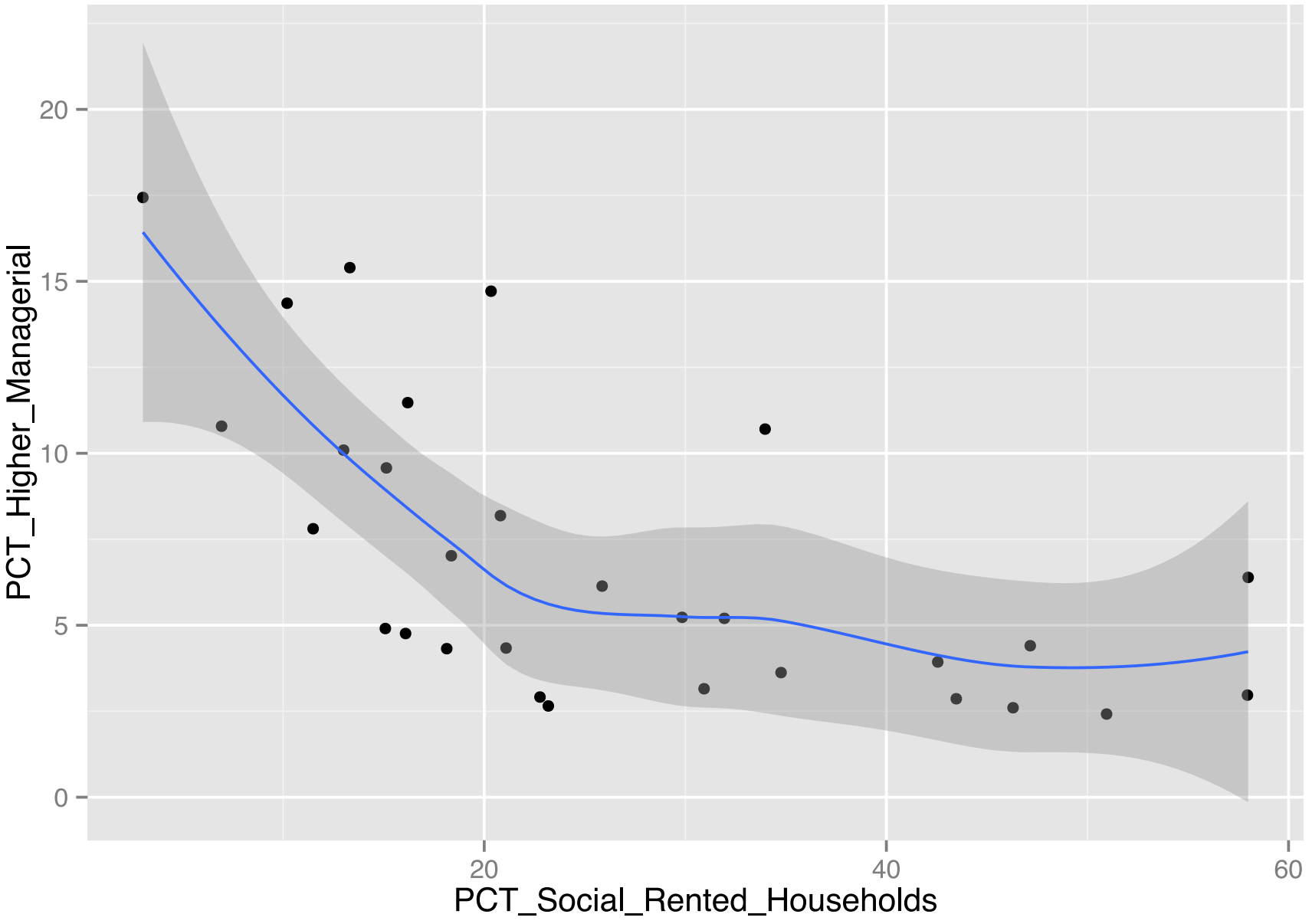


Adding a smoothed line

If we want to add a trend line to the plot this is also possible by adding an addition parameter to the “geom”.

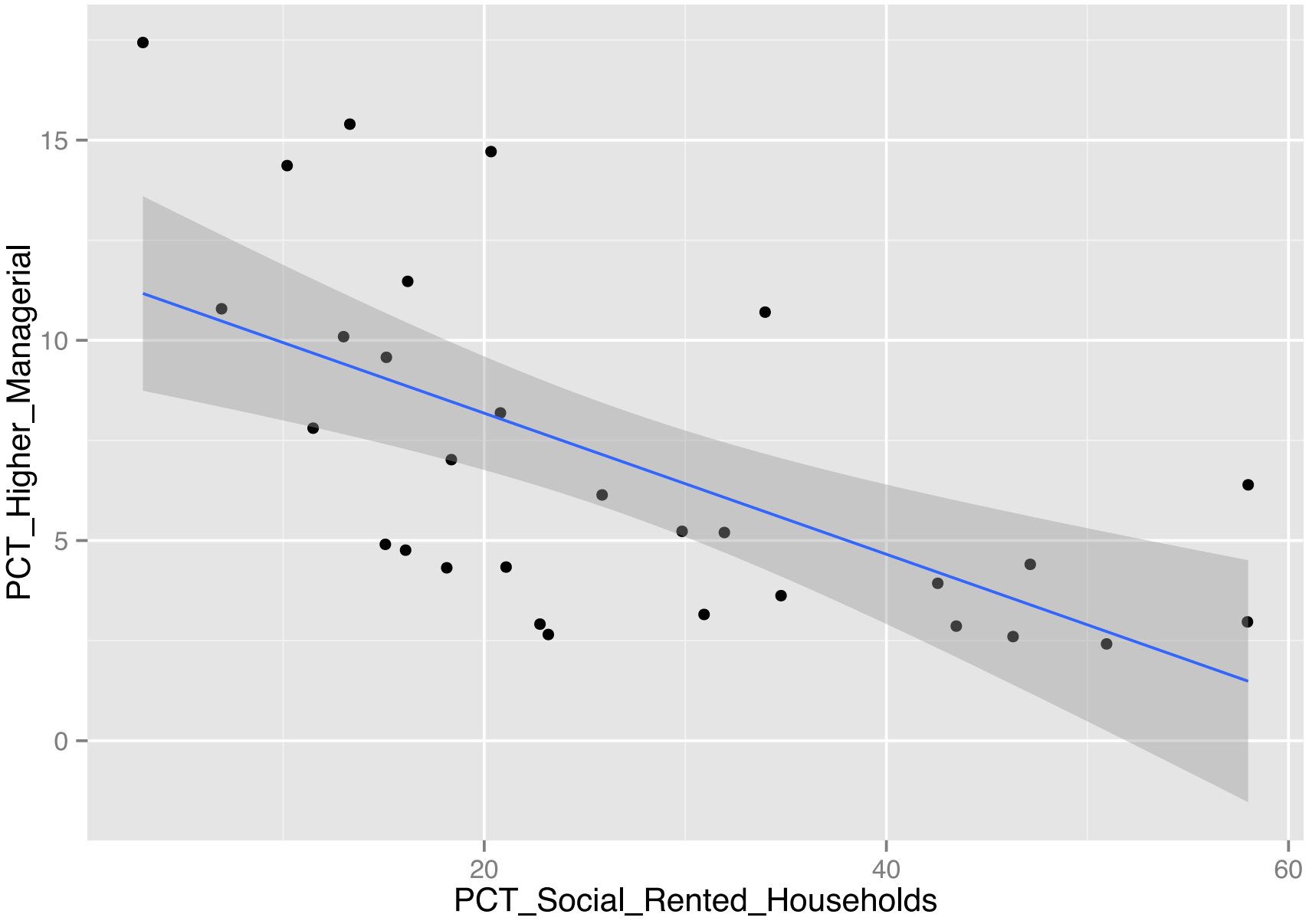
```
qplot(PCT_Social_Rented_Households, PCT_Higher_Managerial, data = census,geom = c(
"point","smooth"))
```

```
## geom_smooth: method="auto" and size of largest group is <1000, so using loess.
Use 'method = x' to change the smoothing method.
```



We might also want a simpler linear regression line; which requires two further parameters including “method”" and “formula”.

```
qplot(PCT_Social_Rented_Households, PCT_Higher_Managerial, data = census,geom = c(
"point","smooth"),method="lm", formula=y~x)
```



Line Plots

To illustrate how to create line plots we will read in some economic data downloaded from the Office for National Statistics (http://www.ons.gov.uk/ons/site-information/using-the-website/time-series/index.html#2) which concerns household expenditure since 1948.

```
household_ex <- read.csv("expenditure.csv")
```

We can then have a quick look at the data and check on the data class.

```
head(household_ex)
```

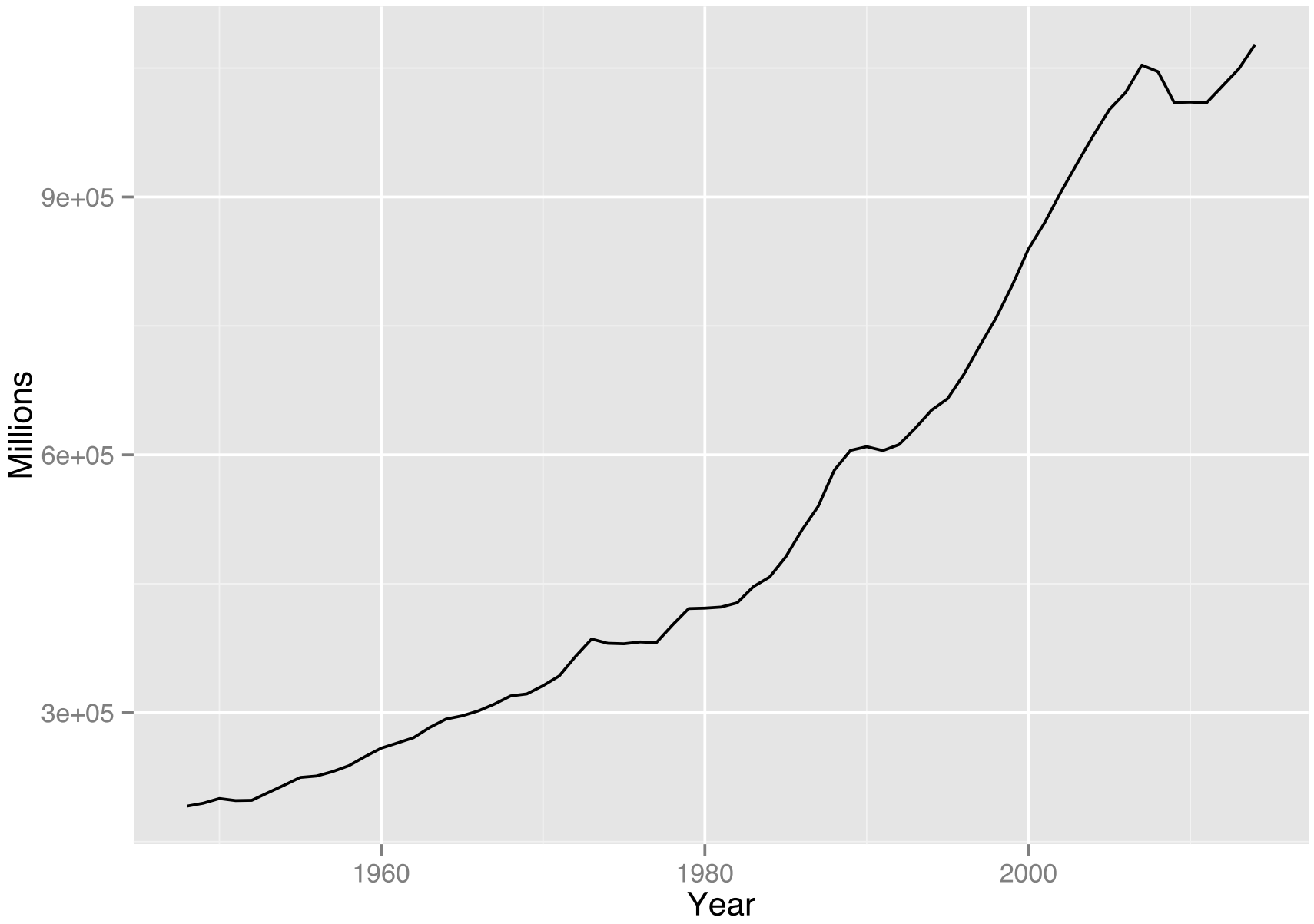
```
##   Year Millions
## 1 1948   191274
## 2 1949   194639
## 3 1950   200097
## 4 1951   197686
## 5 1952   197993
## 6 1953   206868
```

```
lapply(household_ex,class)
```

```
## $Year
## [1] "integer"
##
## $Millions
## [1] "integer"
```

We can now attempt to plot the data.

```
qplot(Year, Millions, data = household_ex, geom = "line")
```



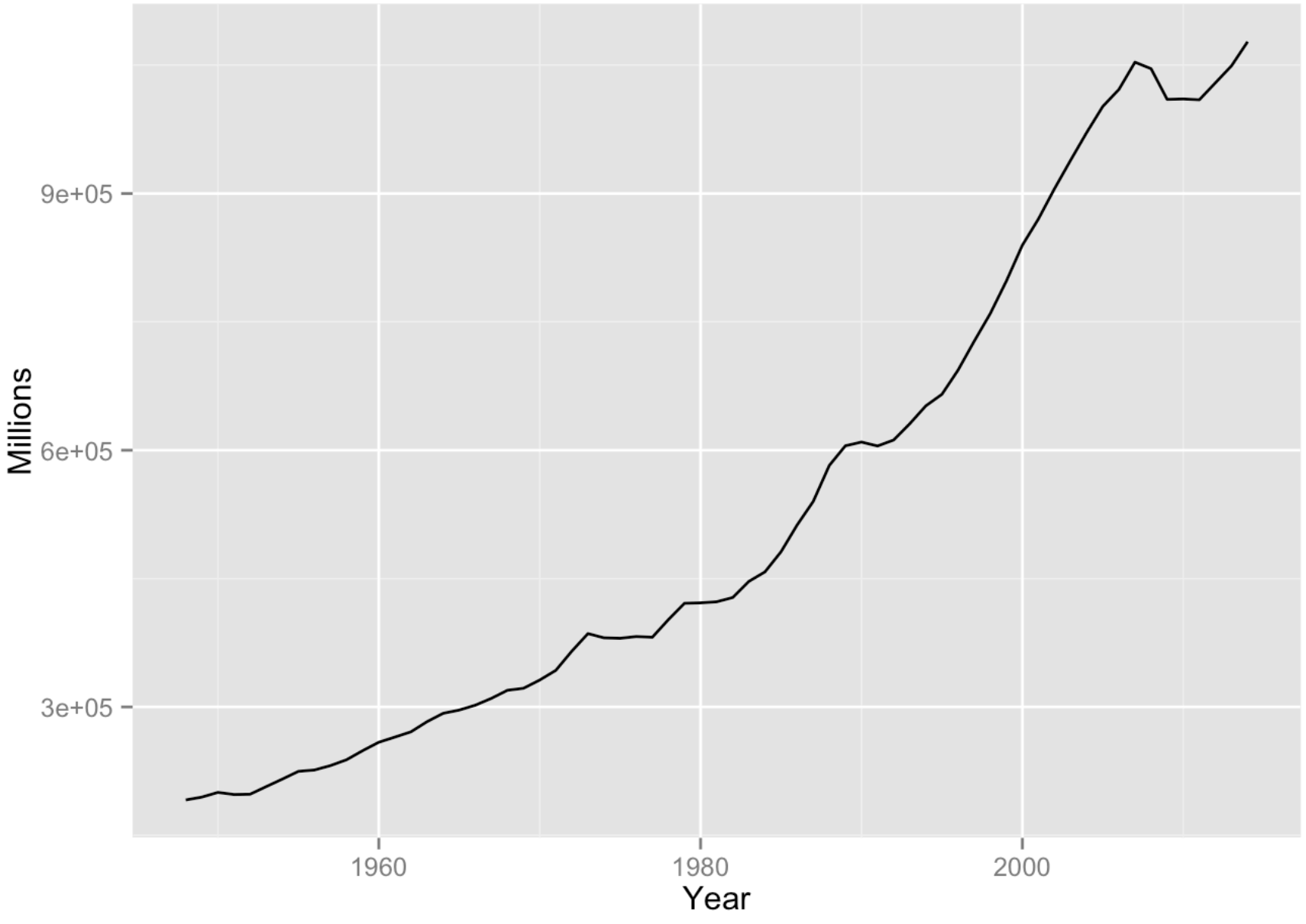
Changing axis and labels

On the y axis, ggplot2 has defaulted to using scientific notation. We can change this, however, we will swap to the main `ggplot` syntax in order to do this. The first stage is to setup the plot, telling ggplot what data to use, and which "aesthetic mappings" (variables!) will be passed to the plotting function. In fact `aes()` is a function, however never used outside of `ggplot()`. This is stored in a variable "p"

```
p <- ggplot(household_ex, aes(Year, Millions))
```

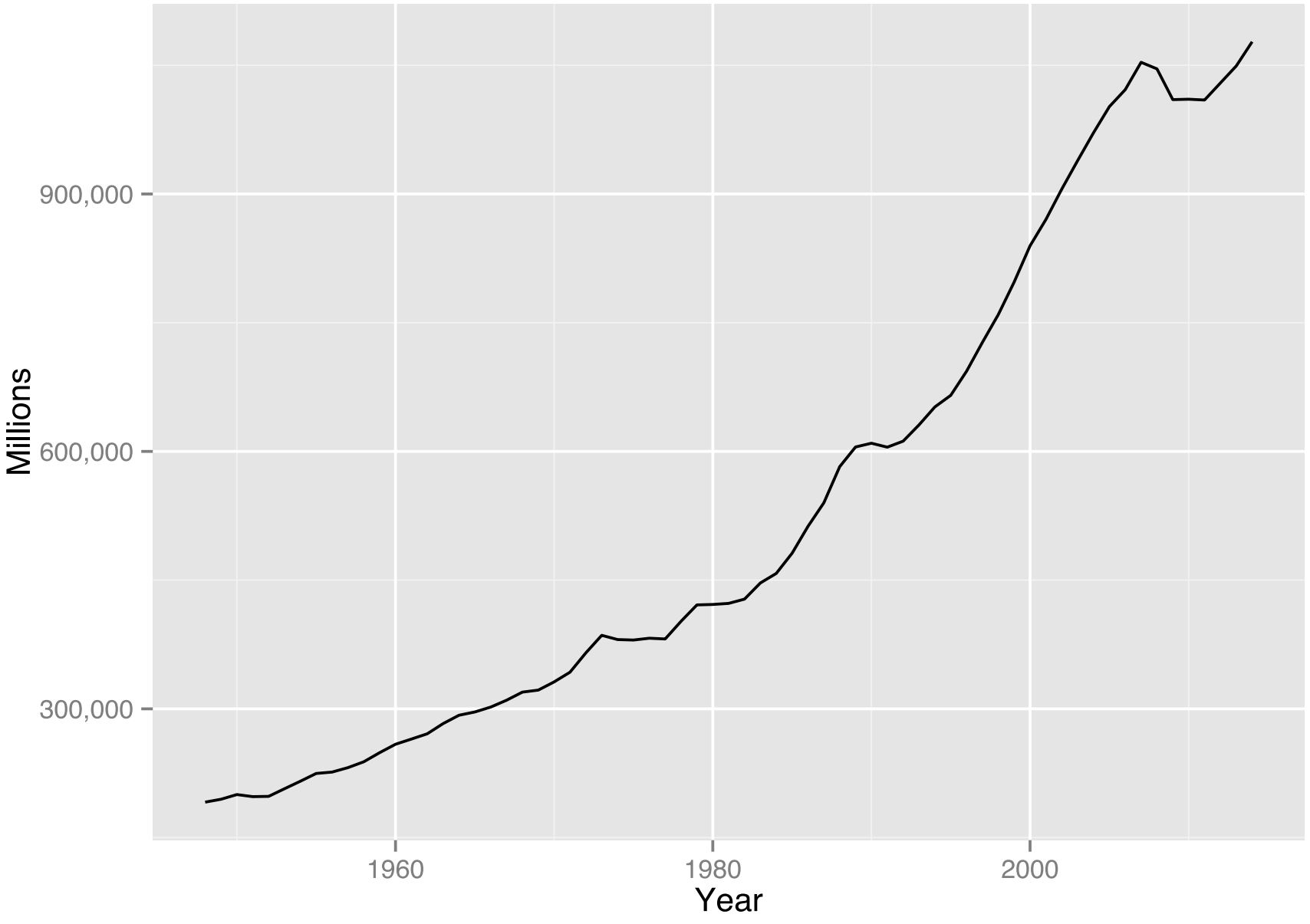
If you just typed "p" into the terminal this would return an error as you still need to tell `ggplot()` which type of graphical output is desired. We do this by adding additional parameters using the "+" symbol.

```
p + geom_line()
```



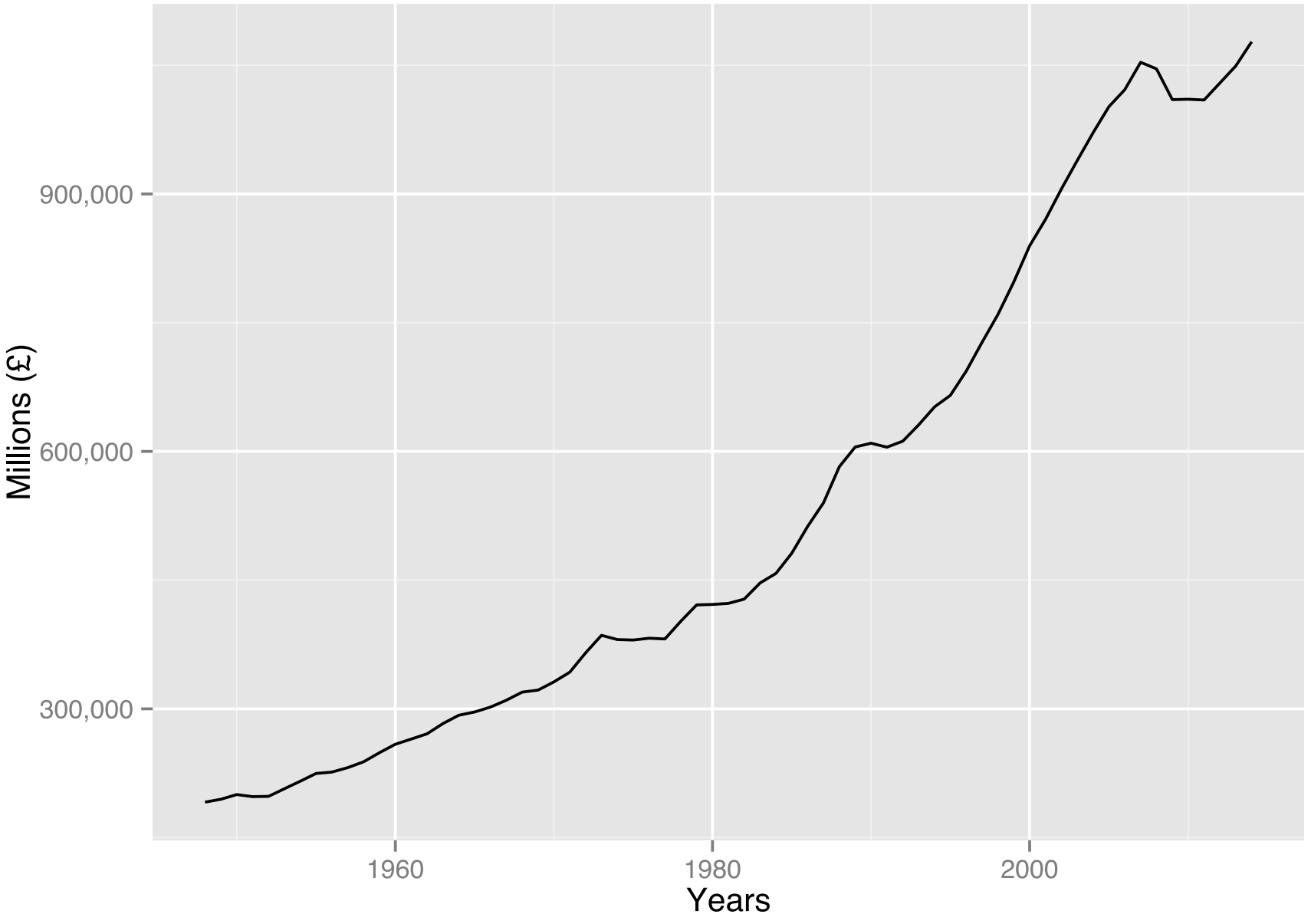
Swapping out the scientific notation requires another library called “scales”. Once loaded, we can then add an additional parameter onto the graph.

```
library(scales)
p + geom_line() + scale_y_continuous(labels = comma)
```



We can also change the x and y axis labels

```
library(scales)
p + geom_line() + scale_y_continuous(labels = comma) + labs(x="Years", y="Millions
(£)")
```



Spatial Data Frames

Before we can create a map in R, we first need to import some spatial data. We will read in two shapefiles for this example, the first containing polygons that will later be used to create a choropleth map (https://en.wikipedia.org/wiki/Choropleth_map), and the second some street segments that will be used to provide context.

We can import a shapefile into R using the `readOGR()` function. In the example, we add the name of the shapefile (without the extension) as the second parameter, and add "." to the first which signifies that R should look for the file within the current working directory.

```
#Read polygons (creates a SpatialPolygonsDataFrame object)
LSOA <- readOGR(".", "E06000042")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "E06000042"
## with 152 features
## It has 12 fields
```

```
#Read lines (creates a SpatialLinesDataFrame object)
roads <- readOGR(".", "Road")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "Road"
## with 12813 features
## It has 1 fields
```

A Spatial Data Frame object stores a range of attributes derived form the shapefile including the geometry of the features (e.g. polygon shape and location), the attributes for each feature (previously stored in the .dbf) and the projection (https://en.wikipedia.org/wiki/Map_projection).

We require a slightly different syntax to view the data contained within the spatial data frame object, and use the “@” symbol. The data is stored in a “slot” called “data”.

```
head(LSOA@data)
```

```
##    LSOA11CD imd_rank imd_score income employment education health crime
## 0 E01016710    11648     23.67   0.20       0.14     28.57   0.47  0.36
## 1 E01016711    25604      8.87   0.10       0.07     10.62  -0.60 -0.76
## 2 E01016713    22671     11.27   0.09       0.09     18.38  -0.42 -0.57
## 3 E01016714     8671     29.07   0.24       0.15     37.17   0.44  0.29
## 4 E01016715    10843     25.04   0.17       0.14     31.33   0.33  0.79
## 5 E01016716    11207     24.39   0.19       0.14     24.94   0.57  0.35
##   housing living_env idaci idaopi
## 0   17.34       2.35  0.25   0.23
## 1   19.87       2.31  0.17   0.07
## 2   18.40       7.72  0.14   0.08
## 3   22.76      19.28  0.31   0.22
## 4   25.15       3.82  0.22   0.16
## 5   18.58      14.52  0.25   0.16
```

All the available slots can be viewed as follows

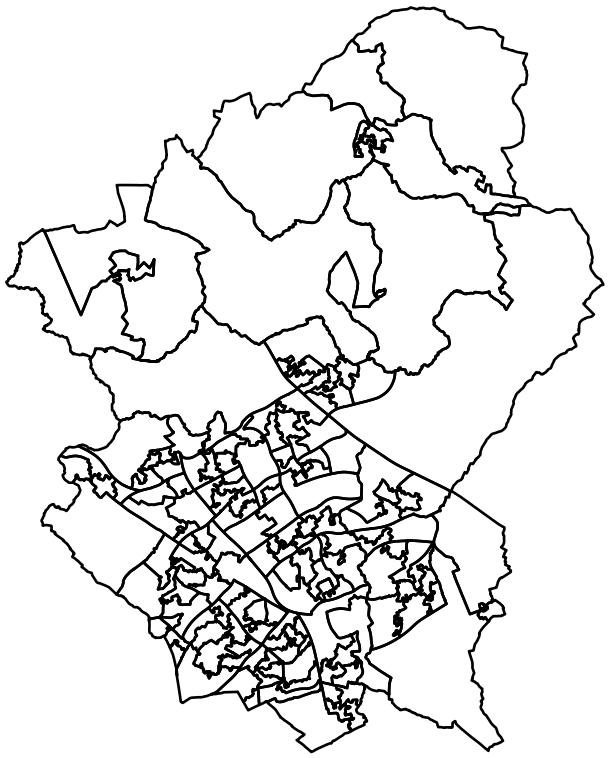
```
slotNames(LSOA)
```

```
## [1] "data"        "polygons"    "plotOrder"   "bbox"        "proj4string"
```

Plotting a spatial data frame

We can use the `plot()` function which is built into base R to show the outlines of the polygons contained within the “LSOA” object.

```
plot(LSOA)
```



This map shows the Lower Layer Super Output Area boundaries for the local authority district of Milton Keynes. The attributes of the data frame are the overall and domain scores for the 2015 Index of Multiple Deprivation (https://www.gov.uk/government/statistics/english-indices-of-deprivation-2015).

We will shade in this map using the overall IMD score which is stored in the column “imd_rank”. There are a total of 152 values.

```
LSOA@data$imd_rank
```

```
##   [1] 11648 25604 22671  8671 10843 11207 12012  3373 20067  7551  9137
##  [12] 22727 25119 12911 11061  6095 10851 20505  6437 15455 18551  7911
##  [23] 22588 27851 31225 16778  8497 22734 20954  6565   752  2839  2770
##  [34] 18299  9368 23993 29156 24245 21270 18728 16220 24263 14368 24839
##  [45] 25995 23522 25124 29853 14061 24259 22557 13307 13909 31203 14125
##  [56] 27752 21883 15621 11812 27468 25571 24412 31672 21115 22891 24975
##  [67] 32370 22815 11324 27378 15427 27970 24601 28917 27549 30837 28489
##  [78] 25950 24948 31850 25107 32657 27560 28558 25444 21660 17228  7774
##  [89] 24838  4724 16243 25828 12185  2721  2055 21149 17150 27271 25171
## [100] 30119 19675 21053  5817 20887 20572 13906 23128 20255 20793 32082
## [111] 21992 28958 10631 23268 21406 18398 27979  9328  5327 15527 12061
## [122] 13869  3727 21063 15044   825  1058   771  1544  9739  4201  4959
## [133]  8465  6155  5279 19975 24541 27256 17521 25346 29805 20147 21800
## [144] 28875 27058 27277 30067 25308 28683 15305 20663 17546
```

The first step is to find suitable breakpoints for the data contained in the imd_rank column. The contiuous data needs to be assigned into categories so different colours can be applied on a choropleth map. There are numerous ways of doing this such as jenks, standard deviations or equal intervals. In this example we use a new function `classIntervals()` from the "classInt" package to find the ranges needed to divide the the imd_rank into five categories. In this example we use the style "fisher" to specify jenks (https://en.wikipedia.org/wiki/Jenks_natural_breaks_optimization) as the break point.

```
breaks <- classIntervals(LSOA@data$imd_rank, n = 5, style = "fisher")
```

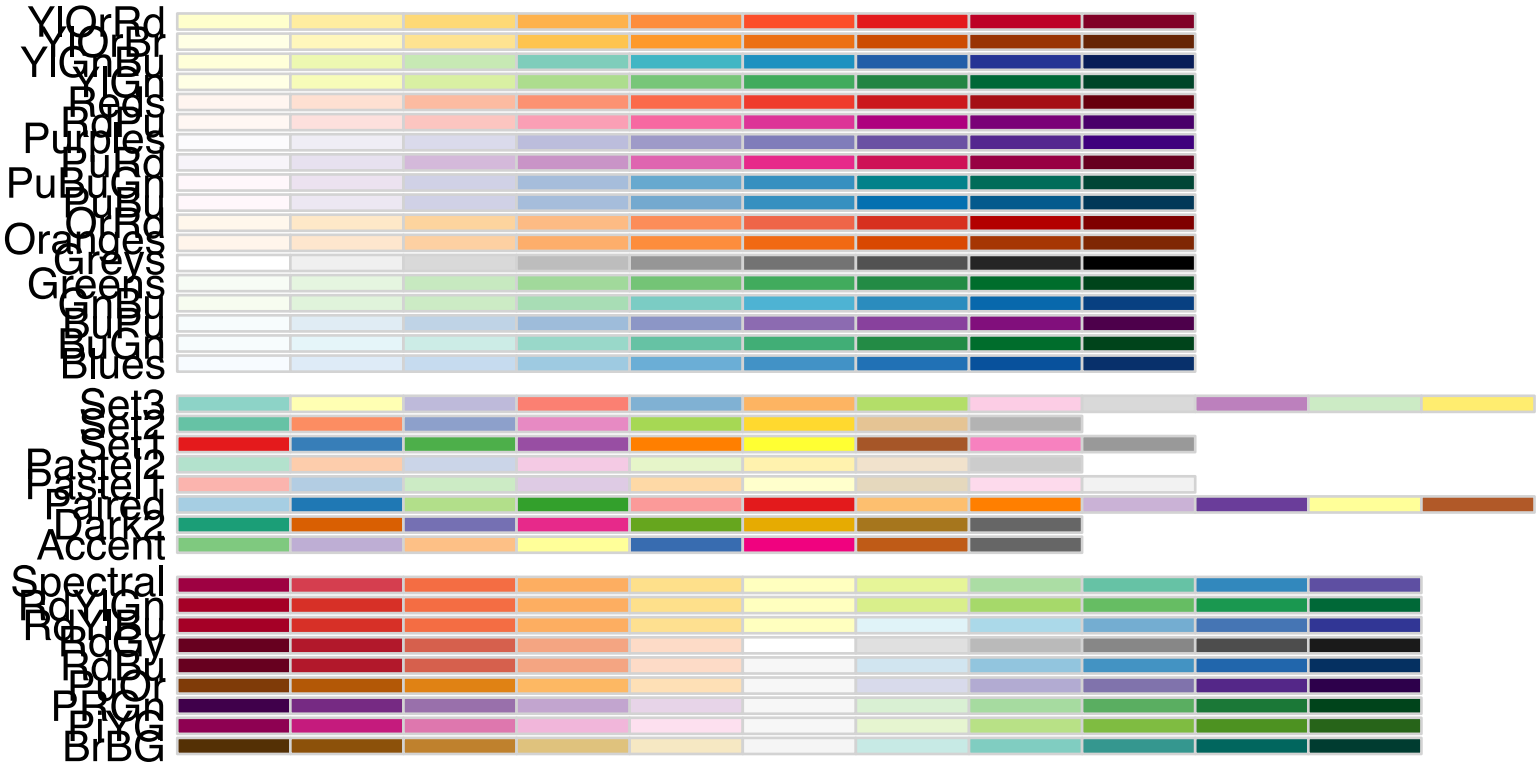
If we print the object created by the `classIntervals()` function, a summary is printed showing you what breaks have been assigned, and how many areas are within these ranges.

```
breaks
```

```
## style: fisher
##   one of 20,811,575 possible partitions of this variable into 5 classes
##        [752,7058)      [7058,13109)   [13109,19201.5) [19201.5,26526.5)
##                21                22                24                54
##   [26526.5,32657]
##                31
```

We need to choose some colours which we will assign to each of the break points in the data. We will now use another package called "RColorBrewer" which provides a series of colour pallets that are suitable for mapping. You can have a look at the colour pallets online: http://colorbrewer2.org (http://colorbrewer2.org). Each of these pallets are named; and you can see all the available pallets as follows...

```
display.brewer.all()
```



We will then choose six colours from the pallet “YlOrRd”, and print them to the terminal. You will see that the colours are stored as hex values (http://www.color-hex.com/).

```
my_colours <- brewer.pal(6, "YlOrRd")
my_colours
```

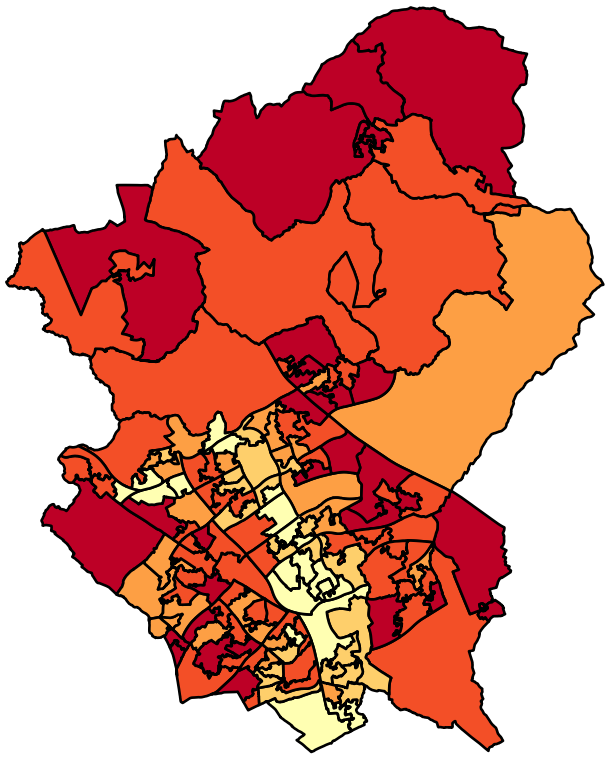
```
## [1] "#FFFFB2" "#FED976" "#FEB24C" "#FD8D3C" "#F03B20" "#BD0026"
```

We can then use the function `findColours()` to select the appropriate colour for each of the numbers we intend to map, depending on where these fit within the break points we calculated.

```
colours_to_map <- findColours(breaks, my_colours)
```

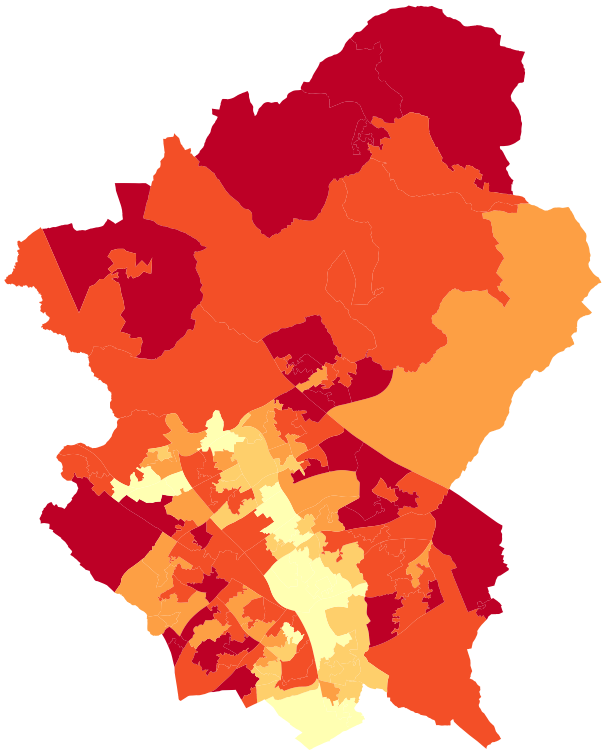
We can then create a basic map using this list of colours and the `plot()` function again.

```
plot(LSOA,col=colours_to_map)
```



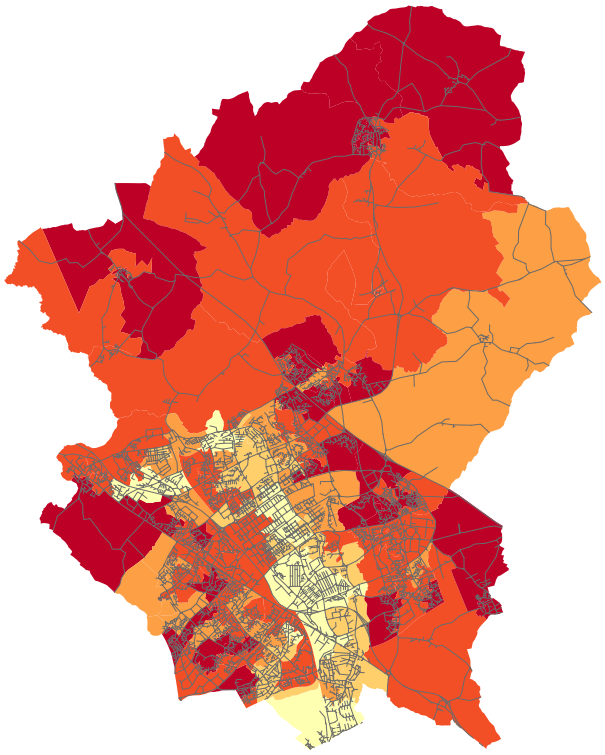
We might also want to create a map without the borders, and can be controlled with an additional parameter which is set to “NA”

```
plot(LSOA,col=colours_to_map,border = NA)
```



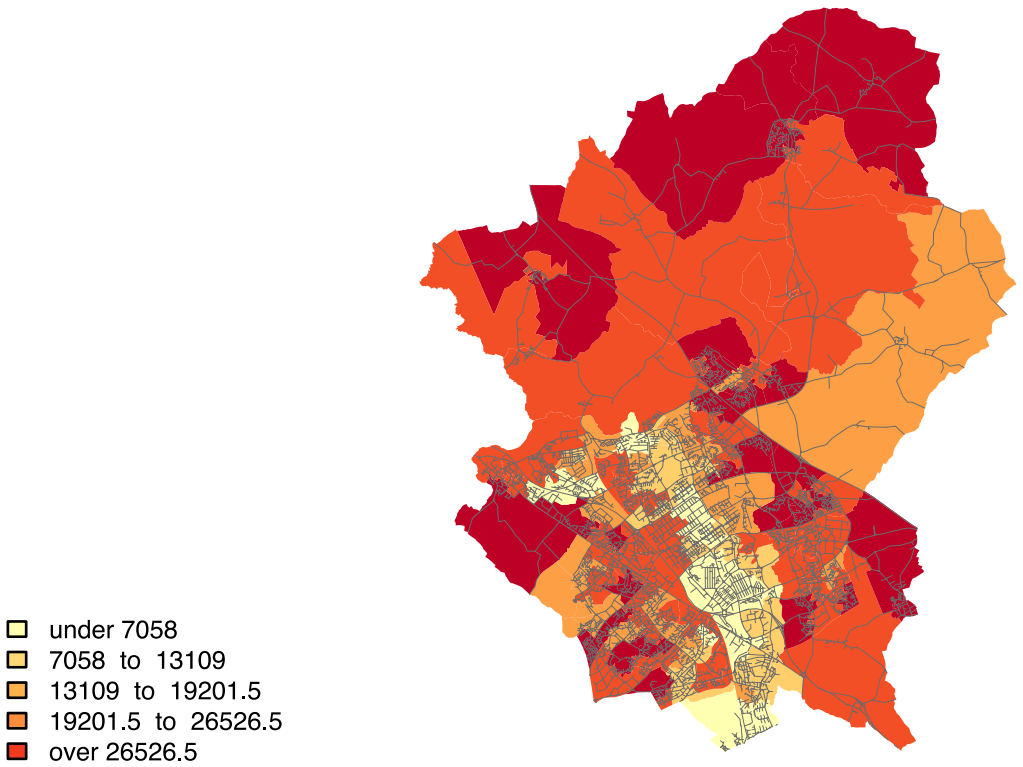
We can also add additional layers onto the map using a further parameter (“add”) which is set to “TRUE”. Without the “add=TRUE”, every time `plot()` is called, the previous plot is replaced. Two further parameters are used, “col” to specify the line colour, and “lwd” the line width.

```
plot(LSOA,col=colours_to_map,border = NA)
plot(roads,add=TRUE, col="#6B6B6B",lwd=0.3)
```



Another feature that is very common to see on a map is a legend which tells you what values the colours used on the map correspond to. This combines the `legend()` function with a further function `leglabs()` to create a legend.

```
plot(LSOA,col=colours_to_map,border = NA)
plot(roads,add=TRUE, col="#6B6B6B",lwd=0.3)
legend("bottomleft" ,legend = leglabs(breaks$brks, between = " to "), fill = my_co
lours, bty = "n",cex=0.6)
```



Building a spatial data frame

It is also possible to import a list of geographic features into R from a text file such as a CSV and then convert these to a spatial data frame. This is a little more complicated than just reading a shapefile. First we will read in a CSV and have a look at the contents.

```
schools <- read.csv("Milton_Keynes_SS_13.csv")
schools
```

```
##       URN                         SCHNAME    PCODE TOTPUPS AC5EM13 Easting
## 1  136468                  Denbigh School  MK5 6EX    1388      80  483107
## 2  136844            The Hazeley Academy  MK8 0PT    1461      71  481650
## 3  110531               Lord Grey School  MK3 6EW    1464      54  485839
## 4  135665     The Milton Keynes Academy  MK6 5LA    1267      53  486120
## 5  136454                Oakgrove School MK10 9JQ    1249      65  488622
## 6  137052                Ousedale School MK16 0BJ    2195      78  486822
## 7  110532           The Radcliffe School MK12 5BT    1145      59  480805
## 8  110517     St Paul's Catholic School  MK6 5EN    1770      56  485623
## 9  136730      Shenley Brook End School  MK5 7ZT    1521      68  483160
## 10 138439      Sir Herbert Leon Academy  MK2 3HQ     676      49  487516
## 11 110526            Stantonbury Campus MK14 6BN    1984      53  484254
## 12 136842                   Walton High  MK7 7WH    1496      64  490091
## 13 110567 The Webber Independent School MK14 6DP     140      70  484790
##    Northing
## 1    236985
## 2    236327
## 3    234073
## 4    237569
## 5    238624
## 6    243183
## 7    240797
## 8    237130
## 9    235011
## 10   232076
## 11   241007
## 12   236608
## 13   241024
```

The new object “schools” contains 13 secondary schools in Milton Keynes, including their Easting and Northing, which are the coordinates of the schools in the British National Grid (http://www.ordnancesurvey.co.uk/support/the-national-grid.html) projection. Before you can make a spatial data frame, you need to check that there are no records with blank spatial references (i.e. Easting and Northing) - in this example, we use the `subset()` function.

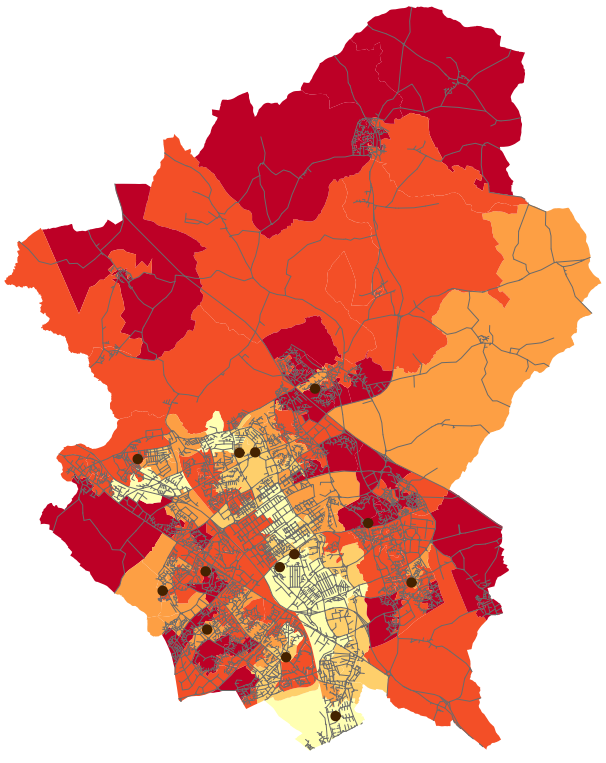
The `SpatialPointsDataFrame()` function is then specified with the coordinates of the school “coords”, which are specified as matrix of values - `cbind()` is used to “column bind” the Easting and Northing lists together, i.e. so each row is a location. The “data” parameter specifies any attribute data - in this case we just use the original data frame. Finally, the “proj4string” is specified using the `CRS()` function. These are standard lookups to known as coordinate systems (http://spatialreference.org/).

```
# Remove those schools without Easting or Northing
schools <- subset(schools, Easting != "" | Northing != "")

# Create the SpatialPointsDataFrame
schools_SDF <- SpatialPointsDataFrame(coords = cbind(schools$Easting, schools$Nort
hing), data = schools, proj4string = CRS("+init=epsg:27700"))
```

We can now plot these locations on our map.

```
plot(LSOA,col=colours_to_map,border = NA)
plot(roads,add=TRUE, col="#6B6B6B",lwd=0.3)
plot(schools_SDF, pch = 19, cex = 0.4, col = "#442200",add=TRUE)
```



Plotting a subset of the map

Much in the same way that we can subset a data frame, we can also create subsets of the plot. For example, suppose we just wanted to view a map for a a specific Ward in Milton Keynes.

First we will read in the Ward boundaries.

```
WARD <- readOGR(".", "england_cmwd_2011Polygon")
```

```
## OGR data source with driver: ESRI Shapefile
## Source: ".", layer: "england_cmwd_2011Polygon"
## with 23 features
## It has 4 fields
```

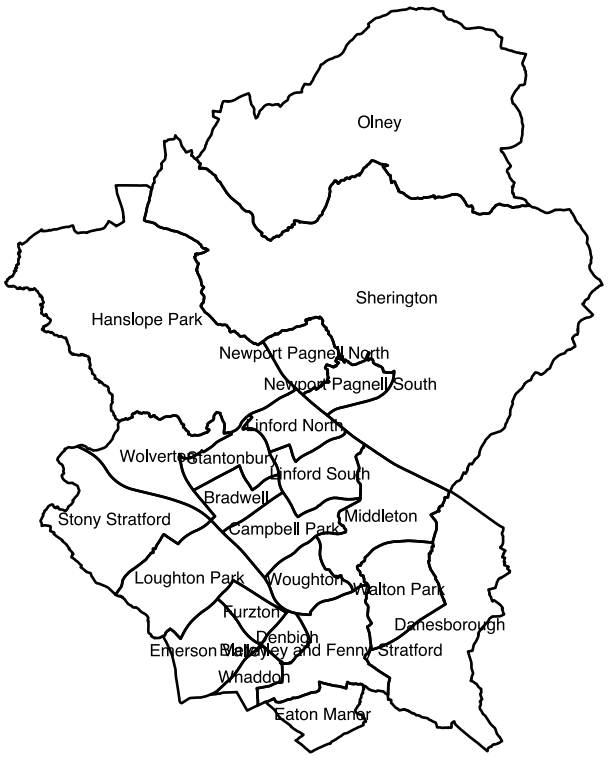
There are 22 wards, which are named as follows.

```
WARD$name
```

```
##  [1] Bletchley and Fenny Stratford Bradwell
##  [3] Campbell Park                 Danesborough
##  [5] Denbigh                       Eaton Manor
##  [7] Emerson Valley                Furzton
##  [9] Hanslope Park                 Linford North
## [11] Linford South                 Loughton Park
## [13] Middleton                     Newport Pagnell North
## [15] Newport Pagnell South         Olney
## [17] Sherington                    Stantonbury
## [19] Stony Stratford               Walton Park
## [21] Whaddon                       Wolverton
## [23] Woughton
## 23 Levels: Bletchley and Fenny Stratford Bradwell ... Woughton
```

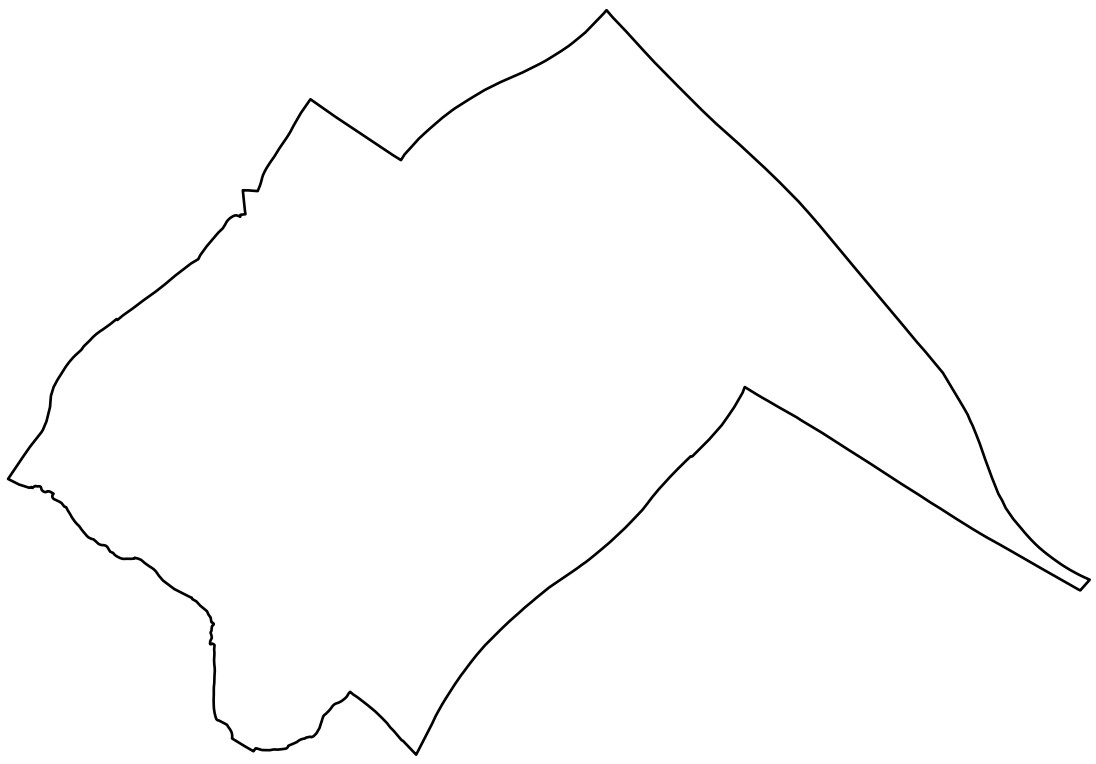
We can plot and label these as follows. The `text()` function applies text to the map, specifying three lists including the Easting, Northing and the text labels. The Easting and Northing are derived using the `coordinates()` function, which for spatial polygons takes the centre of the polygon extent.

```
plot(WARD)
text(coordinates(WARD)[, 1], coordinates(WARD)[, 2], labels = WARD@data$name, cex
= 0.4)
```



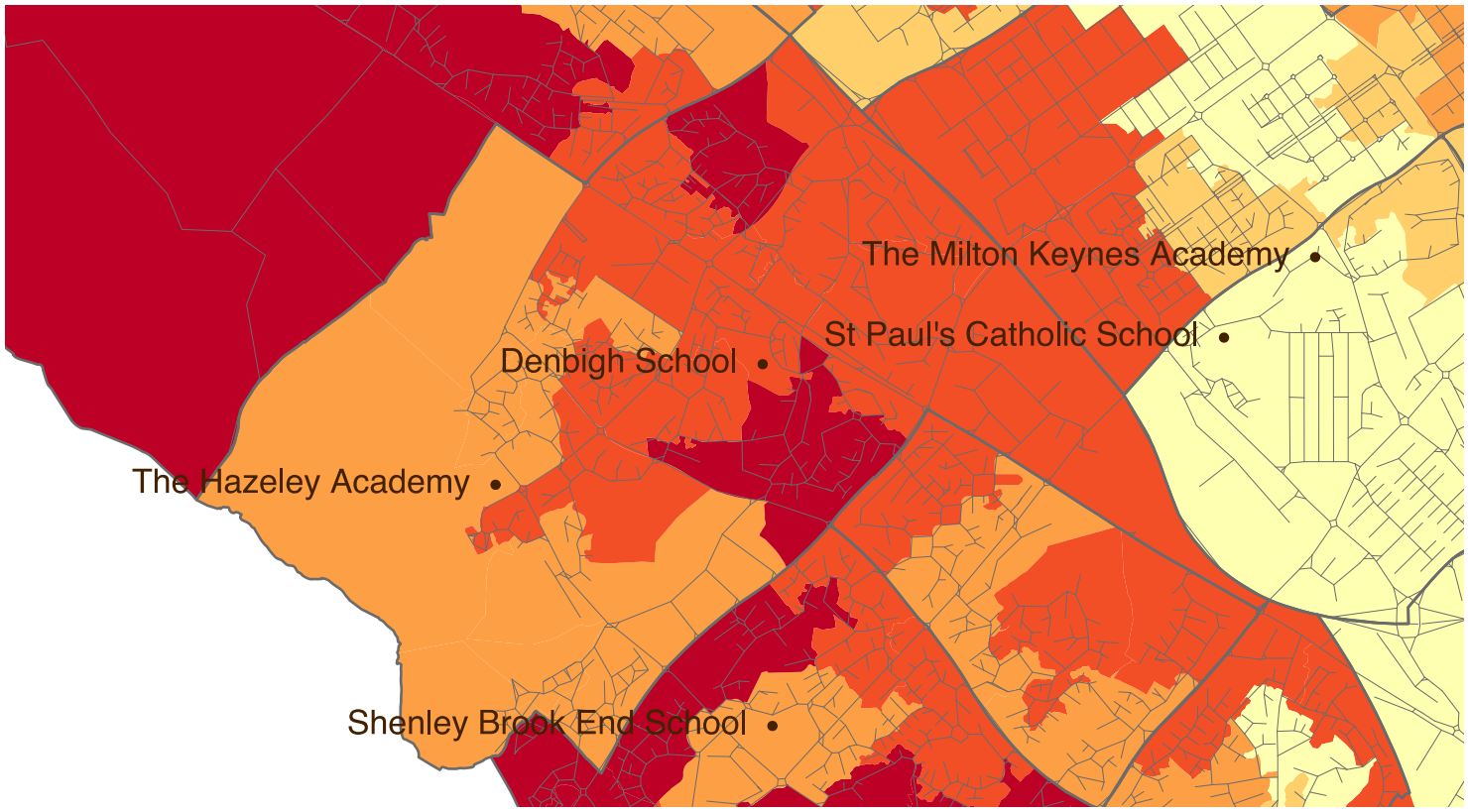
In the following example we can use the attributes of the spatial data frame to plot just a small area of the total spatial polygons data frame object. For example, to plot just Loughton Park we can use the square brackets to just select a single row that matches the name “Loughton Park”

```
plot(WARD[WARD@data$name == "Loughton Park",])
```



This is a useful technique to create a more limited mappable extent. We can use this as follows with some of the previous plots to create a “zoomed in map”. Note how we are building the map up as a series of layers.

```
plot(WARD[WARD@data$name == "Loughton Park",],border=NA) #creates the extent, note that border = NA to make this polygon invisible
plot(LSOA,col=colours_to_map,border = NA,add=TRUE) #plot the choropleth for the IMD
plot(roads,add=TRUE, col="#6B6B6B",lwd=0.3) #plot the roads
plot(schools_SDF, pch = 19, cex = 0.4, col = "#442200",add=TRUE) #plot the schools
plot(WARD,border="#6B6B6B",add=TRUE)#Add the ward boundaries, however this time they have a colour assigned
text(coordinates(schools_SDF)[, 1], coordinates(schools_SDF)[, 2], labels = schools_SDF@data$SCHNAME, cex = 0.8,col="#442200",pos=2)
```



Choropleth mapping in ggmap2

The start of this practical introduced ggplot, which can also be used to plot maps. The first stage is the extract the polygon boundaries from the spatial polygons object. We do this using the `fortify()` function; the “region” parameter is the attribute used to split the polygon - in this case the unique ID for each LSOA.

```
LSOA_FF <- fortify(LSOA, region="LSOA11CD")
```

We can now have a look at the new data frame object this created.

```
head(LSOA_FF)
```

```
##       long      lat order  hole piece       group        id
## 1 487797.2 233640.4     1 FALSE     1 E01016710.1 E01016710
## 2 487795.9 233642.7     2 FALSE     1 E01016710.1 E01016710
## 3 487794.6 233644.9     3 FALSE     1 E01016710.1 E01016710
## 4 487794.4 233645.5     4 FALSE     1 E01016710.1 E01016710
## 5 487780.5 233672.9     5 FALSE     1 E01016710.1 E01016710
## 6 487779.6 233675.2     6 FALSE     1 E01016710.1 E01016710
```

You will see that the polygons have been split up into “groups” which refer to each of the LSOA codes - i.e. what you specified in the “region” attribute. The long and lat are unfortunately named as they are in fact Easting and Northings of the co-ordinates making up the polygon. However, by using the `fortify()` function we have lost the attribute information, which we can add back onto the object using the `merge()` function.

```
LSOA_FF <- merge(LSOA_FF, LSOA@data, by.x = "id", by.y = "LSOA11CD")
```

We can now use these attributes to create a choropleth map in R. First we setup the map using `ggplot()`. We can then tell ggplot how the map should look; firstly stating that the objects are polygons (`+ geom_polygon()`), second that the coordinate system are scaled equally, thus, one coordinate unit north is the same as one unit east for example (`+ coord_equal()`), and then we add some adjustment to the x and y labels, and alter the legend title (`+labs`).

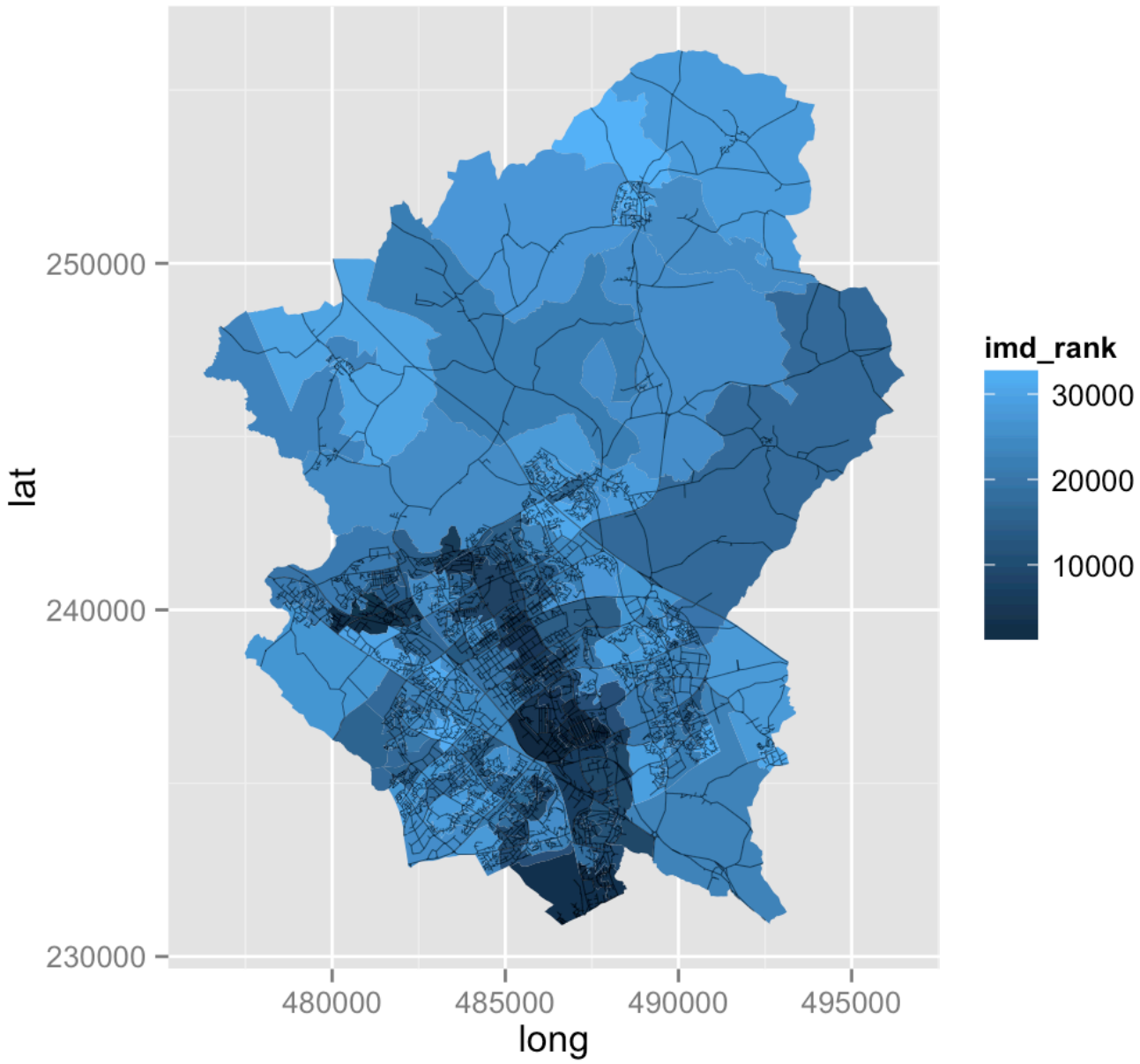
```
Map <- ggplot(LSOA_FF, aes(long, lat, group = group, fill = imd_rank)) +  geom_pol
ygon() + coord_equal() + labs(x = "Easting (m)", y = "Northing (m)", fill = "IMD")
Map
```

We can also add layers to the plot as we did in the previous example. First we will create a fortified version of the roads object.

```
roads_FF <- fortify(roads)
```

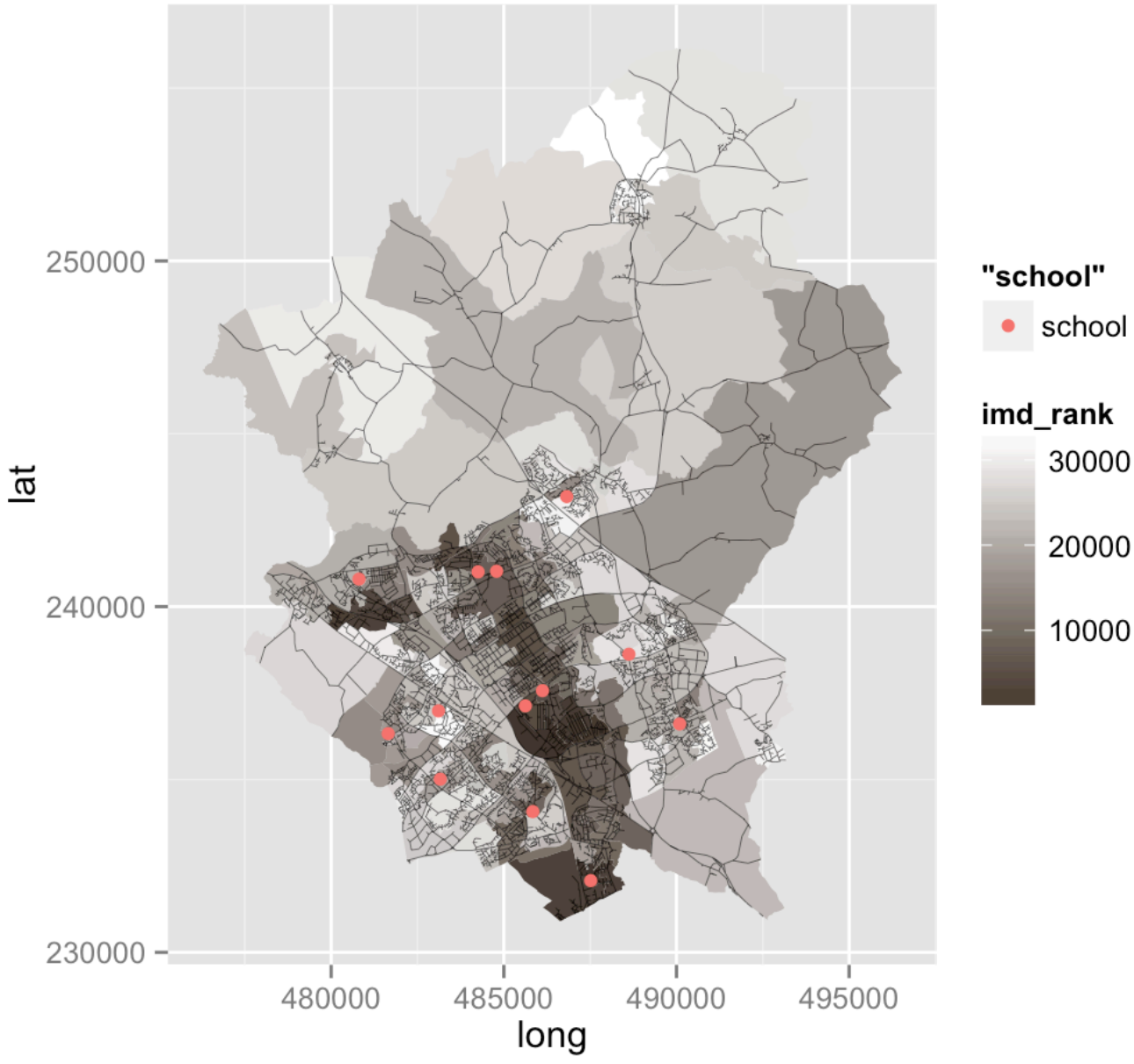
We can then add this to the “Map” object we just created, however we will build this up as layers.

```
plot1<- c(geom_polygon(data=LSOA_FF, aes(long, lat, group = group, fill = imd_rank
)))
plot2<-c(geom_path(data=roads_FF,aes(x=long, y=lat, group=group),size=0.1))
ggplot()+plot1+plot2+coord_equal()
```



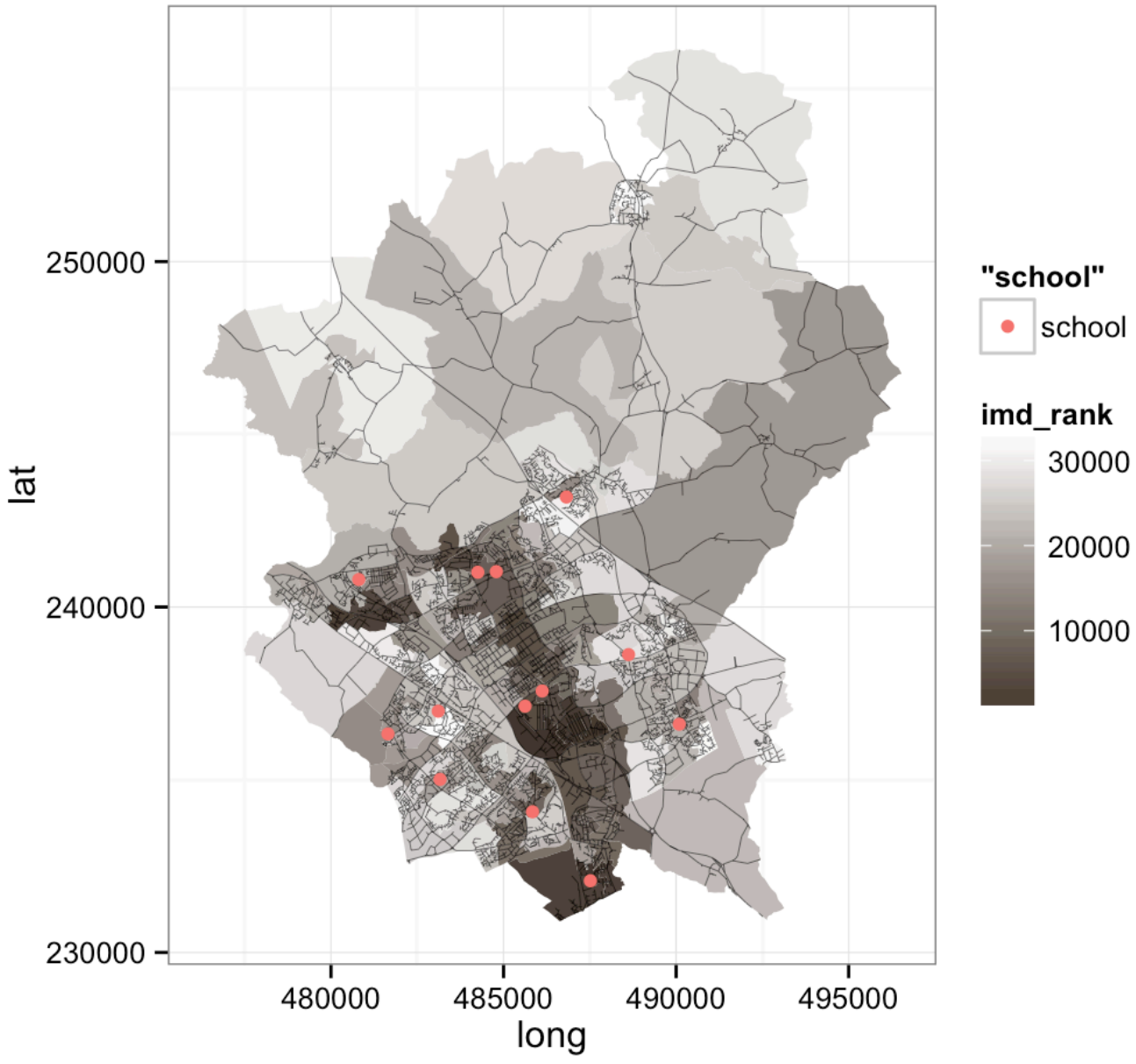
We can add a further layer for the locations of the schools, and also adjust the colour ramp.

```
plot3 <- c(geom_point(data=schools, aes(Easting, Northing,colour='school')))
ggplot()+plot1+plot2+plot3 + coord_equal() + scale_fill_gradient( low="#473B31", h
igh="#FFFFFF")
```



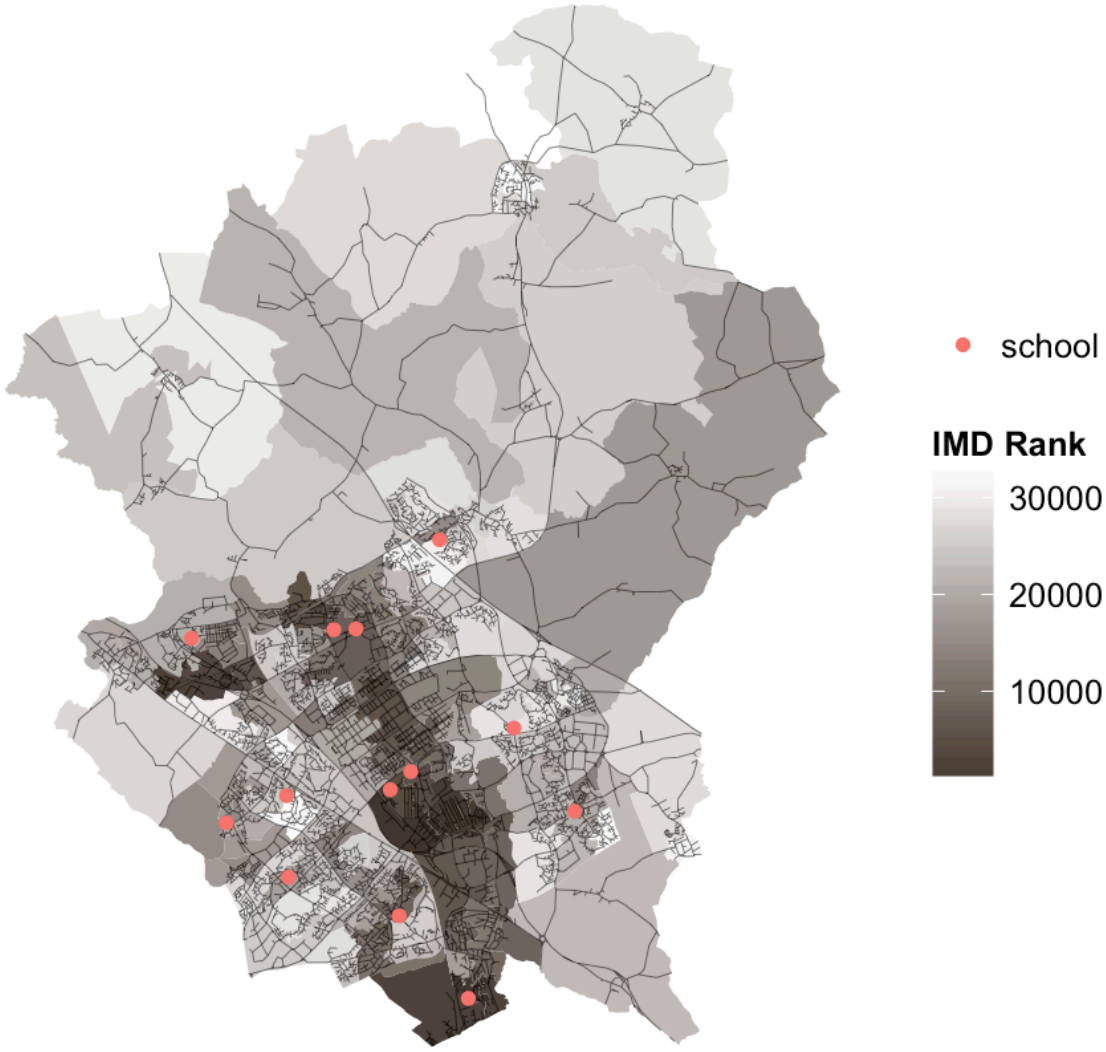
It is also possible to control the other elements of the plot using “theme_bw()” which removes many of the visible elements.

```
ggplot()+plot1+plot2+plot3 +coord_equal() + scale_fill_gradient( low="#473B31", hi
gh="#FFFFFF")  + theme_bw()
```



However, the plot is still a little cluttered and we can turn off many of the elements using “theme()”

```
ggplot()+plot1+plot2+plot3 +coord_equal() + scale_fill_gradient( low="#473B31", hi
gh="#FFFFFF")  + theme_bw() +
  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 = "IMD Rank",colour="")
```


school
IMD Rank
30000
20000
10000