A Mid Week Bit of Fun (1)

Each week I will offer a 'bit of fun' using different bits of contributed code and/or R packages. To follow them you'll need to install the packages from your CRAN mirror, including some additional packages on which they depend.

a) Geocoding locations

In a great many applications it is useful to map our data so we need to know where they are in some co-ordinate system that we understand and can use. A great deal of locational data comes not as nice tidy latitude/longitude pairs but as rather messy place name/address information. *Geocoding* (sometimes called *addmatching* (= address matching)) is the process of turning this text into precise locational coordinates and is now a major industry such that large volume applications are a significant money earner for enterprises that hold the necessary data (notably *Google*™). This is reasonable since geocoding adds a great deal of value to the thematic data.

Basically, *geocoding* take place names and looks them up in a large gazetteer of already geocoded data. The *geocode* function within the*ggmap* package makes this easy and uses the Google API, and is limited to 2,500 requests/day. There are other free geocoding options with higher daily limits if you have more data points. In this very simple example I create a vector of point address information for my home (a small village in England) and the city of Vienna (Austria) and use *geocode* to add longitude and latitudes from the Google gazetteer.

```
> library(ggmap)
Loading required package: ggplot2
Google Maps API Terms of Service: http://developers.google.com/maps/terms.
Please cite ggmap if you use it: see citation('ggmap') for details.
> locations<-c("Maidwell, England", "Vienna, Austria")
> locations
[1] "Maidwell, England" "Vienna, Austria"
> geocoded_locations<-cbind(locations,geocode(locations)
+ )
Information from URL :
http://www.datasciencetoolkit.org/maps/api/geocode/json?address=Maidwell,%20Englan
d&sensor=false
Information from URL :
http://www.datasciencetoolkit.org/maps/api/geocode/json?address=Vienna,%20Austria&
sensor=false
```

```
> geocoded_locations
        locations        lon      lat
1 Maidwell, England -0.9058576 52.38496
2   Vienna, Austria 16.3720800 48.20849
```

There are other ways of doing this, including use of dismo. This package was developed for species distribution modelling, but, like geocode it uses the *Google* API for its main engine and has a geocoding object and mapping facility that is really quite useful. You'll need to install the package dismo, then :

```
>library(XML) #needs this
> library(rgdal) #and this
>library (dismo) # and this
> place<-geocode("Maidwell, Northamptonshire, UK") #the address needs to have enough to be recognized
> place # the place object is a data frame containing the required location together with a bounding box:
```

```
originalPlace                   interpretedPlace longitude
1 Maidwell, Northamptonshire, UK Maidwell, Northamptonshire NN6, UK -0.906865
  latitude       xmin       xmax     ymin     ymax uncertainty
1 52.38395 -0.9228724 -0.8908576 52.37714 52.39076        1326
```

So we have located my village at (decimal) longitude -0.906865, latitude 52.38395. Note that

- dismo returns the long/lat location in a data frame.
- the result differs slightly according to the package used
- As is usual with lat/long values in general in the R ecosystem these are listed in reverse of the usual convention as long/lat. Annoying but OK once you remember it.

All well and good, but if you input a long list of text with place names how many do you think will be resolved and geocoded? You might like to think about all the possible problems that geocoding can turn up. Playing around with some possible input data will soon show you that large scale geocoding isn't easy.

b) Drawing pin maps with Google Earth™

This bit of fun is due to James Cheshire, Centre for Advanced Spatial Analysis (CASA) at my old *alma mater*, University College London. It shows how to export spatial data as KML so that it can be plotted on *Google Earth/ Google Maps*™. It is a good example of the careful use of comments. Even if you don't run thus sequence, please read through both the code and the commentary.

To follow the sequence you will need:

- *Google Earth*™ installed on your machine;
- An *Excel* ™ csv file that contains the data we require called London_cycle_hire_locs.csv. Download this from the learning management system. It will help if this is to your desktop

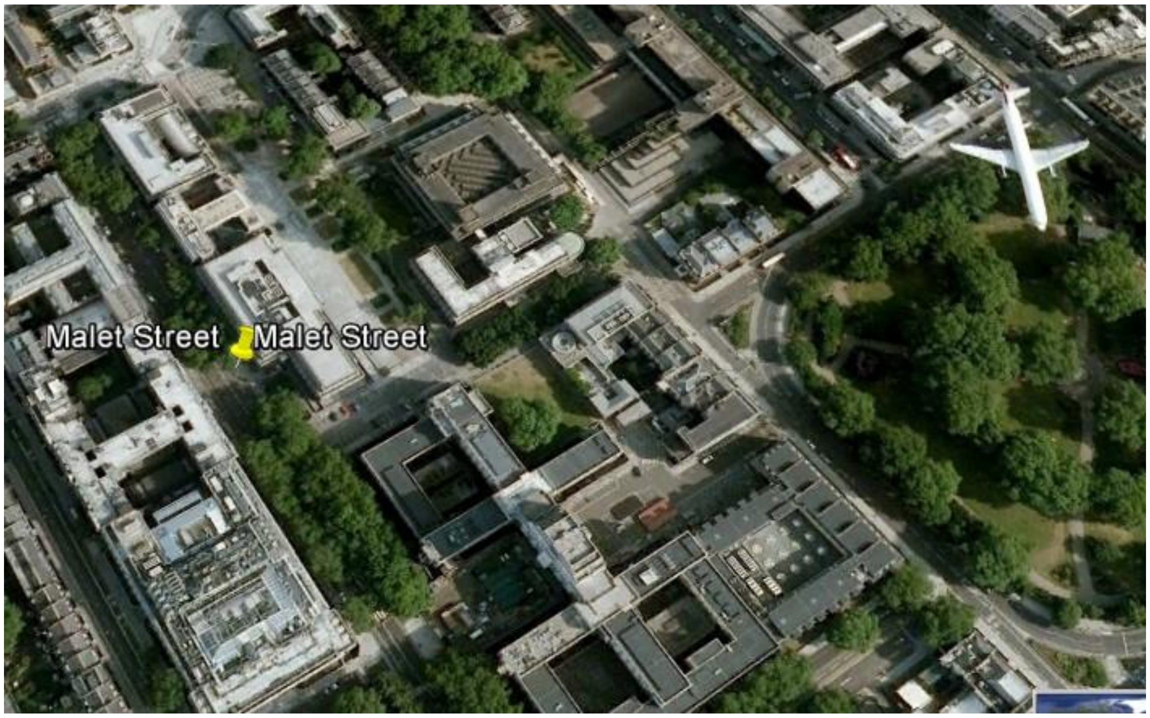
```
##Data Requirements:
##London Cycle Hire locations.
##Install and load maptools, sp, and rgdal  (if you haven't already done so):
library(maptools)
library(rgdal)
library(sp)
## Load the cycle hire locations.
## I have the file on my desk top --- you can download from the LMS
## Note the read.csv method --- very useful!
cycle<- read.csv("C:\\Users\\David\\Desktop\\London_cycle_hire_locs.csv",
header=T)
## Inspect column headings
head(cycle)
    Name      Village      X      Y Capacity
1            Kensington Olympia Station      Olympia 524384 179210       25
2           Ilchester Place   Kensington 524844 179509       24
3           Chepstow Villas Notting Hill 524897 180779       17
4          Turquoise Island Notting Hill 524940 181022       21
5        West Cromwell Road Earl's Court 525174 178737       24
6          Pembridge Villas Notting Hill 525179 180668       16

## Plot the XY coordinates (do not close the plot window)
## The X and Y are in British National Grid projection (BNG)
plot(cycle$X, cycle$Y)
## create a SpatialPointsDataframe object and add the appropriate CRS
coordinates(cycle)<- c("X", "Y")
BNG<- CRS("+init=epsg:27700")
proj4string(cycle)<-BNG
## In order for the points to be displayed in the correct place they need to be re-projected
to WGS84 geographical coordinates as used by Google products
p4s <- CRS("+proj=longlat +ellps=WGS84 +datum=WGS84")
cycle_wgs84<- spTransform(cycle, CRS= p4s)
## Using the OGR KML driver we can then export the data to KML. "dsn" should equal
the full path and file name of the exported file and the dataset_options argument allows
us to specify the labels displayed by each of the points.
```

```
writeOGR(cycle_wgs84,
dsn="C:\\Users\\David\\Desktop\\london_cycle_docks.kml", layer= "cycle_wgs84",
driver="KML", dataset_options=c("NameField=name"))
quit()
```

This sequence has created a *Keyhole Markup Language* (KML) file that *Google Earth*™ can display. With *Google Earth* running on your machine, open the KML file you just created and then use the usual Google tools to inspect the locations. The points are loaded as *labeled* pins on the map. If you click on the pin you will be able to see its full name and capacity. This works with any file of pin locations, but you need to take care to get the spatial co-ordinates into WGS84 using spTransform in the *sp* package as described below.

A year or so ago, when I was checking that the docking station in Malet Street outside where I used to work in the University of London is in the right place I noticed a strange object 'on the map' to its east:



I can assure you that at the time there was no airliner parked on the trees in Russell Square. Aerial photographs are NOT maps! Sadly, the most recent Google earth imagery for London has been updated and so no longer shows the airliner.

A Note on Map Projections

Throughout this course I have chosen not to engage with the arcanums of how geospatial data are projected, but in a lot of practical work it will be necessary to make sure that you work in a consistent co-ordinate system that puts locations where they should be. The example above is typical. The cycle locations are latitudes and longitudes in a reference system called BNG (*British Ordnance Survey National Grid*) whereas the *Google* framework data onto which these are mashed use WGS84 (World Geodetic System, 1984). It comes as a surprise for those not into geodesy that WGS84 and OSGB projections render *different* latitude/longitude co-ordinates. If you look at:

http://www.theregister.co.uk/2006/02/06/greenwich_meridian/ (checked 27-11-15)

and the debate that followed (Google "Google + Greenwich") you will see what I mean!

The difference isn't great but at this scale of mapping, and certainly if we want the hire docking stations to be on the right streets, it most certainly does matter.

Map projections are something that in practice you also need to know about when dealing with geographic data. Almost always the locational information comes as co-ordinates in some projection or other, or sometimes as what are called *geographical co-ordinates* (latitude/longitude pairs) . In R this is a co-ordinate reference system (CRS) and it provides a way of representing location on a bumpy ellipsoid (Planet Earth) on a flat sheet of paper or screen. Many scientists coming new to the field assume that this is simple, but be warned that it is not.

In our example (above) Google assumes that its data are in what's called the *World Geodetic System* of 1984 (WGS84). The package rgdal has the tools you need, but use is also made of the rather unlikely source of information about projections provided by the *European Petroleum Survey Group* (see: www.Epsg.org, checked 26-11-15). The statements are:

```
BNG<- CRS("+init=epsg:27700")
```

EPSG reference 27700 is the 1936 British Ordnance Survey National Grid in which the locations are located but to map them using Google we have to transform these data into WGS84:

```
proj4string(cycle)<-BNG
p4s <- CRS("+proj=longlat +ellps=WGS84 +datum=WGS84")
cycle_wgs84<- spTransform(cycle, CRS= p4s)
```

All these projections are referenced to some relatively arbitrary datum, which is just a point in three dimensional space, and it comes as a surprise to many that doing the transform is a *three* dimensional operation, not just a co-ordinates shift in two.

Dave Unwin

History: 30-11-13, rev 20-10-14, rev to include ggmap 25-11-15