Combining Spatial Data*

Roger Bivand

June 25, 2013

1 Introduction

2 Checking Topologies

In this vignette, we look at a practical example involving the cleaning of spatial objects originally read into R from shapefiles published by the US Census. We then aggregate them up to metropolitan areas using a text table also from the US Census.

The data in this case are for polygons representing county boundaries in 1990 of North Carolina, South Carolina, and Virginia, as shown in Fig. 1. The attribute data for each polygon are the standard polygon identifiers, state and county identifiers, and county names. All the spatial objects have the same number of columns of attribute data of the same types and with the same names. The files are provided without coordinate reference systems as shapefiles; the metadata are used for choosing the `CRS` values.

```
> owd <- getwd()
> setwd(system.file("shapes", package = "maptools"))
> library(maptools)
> nc90 <- readShapeSpatial("co37_d90")
> proj4string(nc90) <- CRS("+proj=longlat +datum=NAD27")
> sc90 <- readShapeSpatial("co45_d90")
> proj4string(sc90) <- CRS("+proj=longlat +datum=NAD27")
> va90 <- readShapeSpatial("co51_d90")
> proj4string(va90) <- CRS("+proj=longlat +datum=NAD27")
> setwd(owd)
```

As read in, shapefiles usually have the polygon IDs set to the external file feature sequence number from zero to one less than the number of features. In our case, wanting to combine three states, we need to change the ID values so that they are unique across the study area. We can use the FIPS code (Federal Information Processing Standards Publication 6-4), which is simply the two-digit state FIPS code placed in front of the three-digit within-state FIPS county code, ending up with a five-digit string uniquely identifying each county. We can also drop the first four attribute data columns, two of which (area and perimeter) are misleading for objects in geographical coordinates, and the other two are internal ID values from the software used to generate the shapefiles, replicating the original feature IDs. We can start with the data set of South Carolina (`sc90`):

*This vignette formed pp. 120–126 of the first edition of Bivand, R. S., Pebesma, E. and Gómez-Rubio V. (2008) Applied Spatial Data Analysis with R, Springer-Verlag, New York. It was retired from the second edition (2013) to accommodate material on other topics, and is made available in this form with the understanding of the publishers.

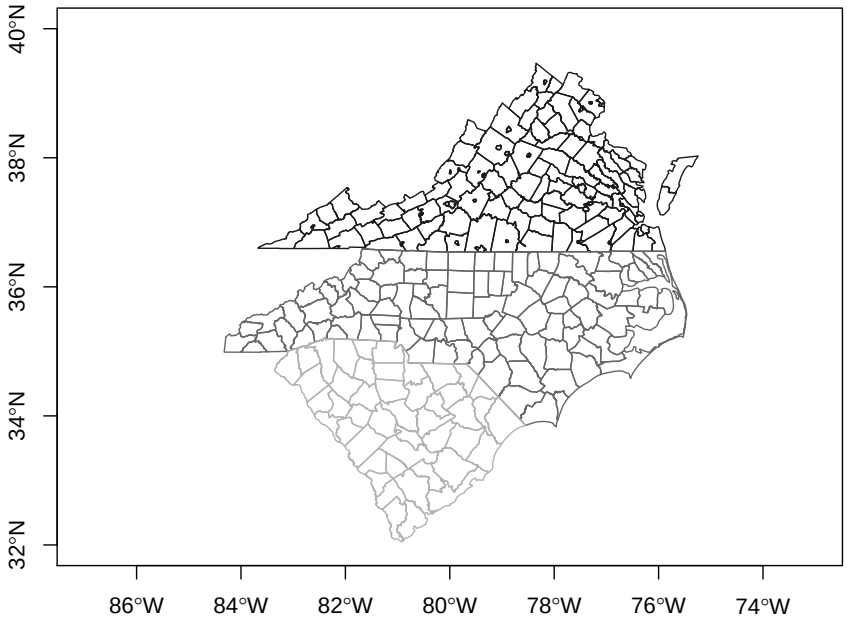

Figure 1: The three states plotted from input spatial objects using different grey colours for county boundaries

```
> library(maptools)
> names(sc90)
[1] "AREA"       "PERIMETER"  "CO45_D90_"  "CO45_D90_I" "ST"         "CO"
[7] "NAME"
> sc90a <- spChFIDs(sc90, paste(sc90$ST, sc90$CO, sep = ""))
> sc90a <- sc90a[, -(1:4)]
> names(sc90a)
[1] "ST"   "CO"   "NAME"
> proj4string(sc90a) <- CRS(proj4string(sc90a))
```

2.1 Dissolving Polygons

When we try the same sequence of commands for North Carolina, we run into difficulties:

```
> names(nc90)
[1] "AREA"       "PERIMETER"  "CO37_D90_"  "CO37_D90_I" "ST"         "CO"
[7] "NAME"
> nc90a <- spChFIDs(nc90, paste(nc90$ST, nc90$CO, sep = ""))
Error in `row.names<-.data.frame`(`*tmp*`, value = c("37009", "37005",  :
  duplicate 'row.names' are not allowed
```

Tabulating the frequencies of polygons per unique county ID, we can see that 98 of North Carolina's counties are represented by single polygons, while one has two polygons, and one (on the coast) has four.

```
> table(table(paste(nc90$ST, nc90$CO, sep = "")))

 1  2  4
98  1  1
```

One reason for spatial data being structured in this way is that it is following the OpenGIS®[1] Simple Features Specification, which allows polygons to have one and only one external boundary ring, and an unlimited number of internal boundaries – holes. This means that multiple external boundaries – such as a county made up of several islands – are represented as multiple polygons. In the specification, they are linked to attribute data through a look-up table pointing to the appropriate attribute data row.

We need to restructure the `SpatialPolygons` object such that the `Polygon` objects belonging to each county belong to the same `Polygons` object. To do this, we use a function[2] in the **maptools** package also used for dissolving or merging polygons, but which can be used here to re-package the original features, so that each `Polygons` object corresponds to one and only one county:

```
> nc90a <- unionSpatialPolygons(nc90, IDs = paste(nc90$ST, nc90$CO, sep = ""))
```

The function uses the IDs argument to set the ID slots of the output `SpatialPolygons` object. Having sorted out the polygons, we need to remove the duplicate rows from the data frame and put the pieces back together again:

```
> nc90_df <- as(nc90, "data.frame")[!duplicated(nc90$CO), -(1:4)]
> row.names(nc90_df) <- paste(nc90_df$ST, nc90_df$CO, sep = "")
> nc90b <- SpatialPolygonsDataFrame(nc90a, nc90_df)
```

2.2 Checking Hole Status

Looking again at Fig. 1, we can see that while neither North Carolina nor South Carolina has included boroughs within counties, these are frequently found in Virginia. While data read from external sources are expected to be structured correctly, with the including polygon having an outer edge and an inner hole, into which the outer edge of the included borough fits, we can also check and correct the settings of the hole slot in `Polygon` objects. The `checkPolygonsHoles` function takes a `Polygons` object as its argument, and, if multiple `Polygon` objects belong to it, checks them for hole status using functions from the **rgeos** package:

```
> va90a <- spChFIDs(va90, paste(va90$ST, va90$CO, sep = ""))
> va90a <- va90a[, -(1:4)]
> va90_pl <- slot(va90a, "polygons")
> va90_pla <- lapply(va90_pl, checkPolygonsHoles)
> p4sva <- CRS(proj4string(va90a))
> vaSP <- SpatialPolygons(va90_pla, proj4string = p4sva)
> va90b <- SpatialPolygonsDataFrame(vaSP, data = as(va90a, "data.frame"))
```

Here we have changed the `Polygons` ID values as before, and then processed each `Polygons` object in turn for internal consistency, finally re-assembling the cleaned object. So we now have three spatial objects with mutually unique IDs, and with data slots containing data frames with the same numbers and kinds of columns with the same names.

[1] See `http://www.opengeospatial.org/`.

[2] This function requires that the **rgeos** package is also installed.

3 Combining Spatial Data

It is quite often desirable to combine spatial data of the same kind, in addition to combining positional data of different kinds as discussed earlier in this chapter. There are functions `rbind` and `cbind` in R for combining objects by rows or columns, and `rbind` methods for `SpatialPixels` and `SpatialPixelsDataFrame` objects, as well as a `cbind` method for `SpatialGridDataFrame` objects are included in **sp**. In addition, methods with slightly different names to carry out similar operations are included in the **maptools** package.

3.1 Combining Positional Data

The `spRbind` method combines positional data, such as two `SpatialPoints` objects or two `SpatialPointsDataFrame` objects with matching column names and types in their data slots. The method is also implemented for `SpatialLines` and `SpatialPolygons` objects and their `*DataFrame` extensions. The methods do not check for duplication or overlapping of the spatial objects being combined, but do reject attempts to combine objects that would have resulted in non-unique IDs.

Because the methods only take two arguments, combining more than two involves repeating calls to the method:

```
> nc_sc_va90 <- spRbind(spRbind(nc90b, sc90a), va90b)
> FIPS <- row.names(nc_sc_va90)
> str(FIPS)
```

```
 chr [1:282] "37001" "37003" "37005" "37007" "37009" "37011" "37013" ...
```

```
> length(slot(nc_sc_va90, "polygons"))
```

```
[1] 282
```

3.2 Combining Attribute Data

Here, as very often found in practice, we need to combine data for the same spatial objects from different sources, where one data source includes the geometries and an identifying index variable, and other data sources include the same index variable with additional variables. They often include more observations than our geometries, sometimes have no data for some of our geometries, and not are infrequently sorted in a different order. The data cleaning involved in getting ready for analysis is a little more tedious with spatial data, as we see, but does not differ in principle from steps taken with non-spatial data.

The text file provided by the US Census tabulating which counties belonged to each metropolitan area in 1990 has a header, which has already been omitted, a footer with formatting information, and many blank columns. We remove the footer and the blank columns first, and go on to remove rows with no data – the metropolitan areas are separated in the file by empty lines. The required rows and column numbers were found by inspecting the file before reading it into R:

```
> t1 <- read.fwf(system.file("share/90mfips.txt", package = "maptools"),
+     skip = 21, widths = c(4, 4, 4, 4, 2, 6, 2, 3, 3, 1, 7, 5, 3, 51),
+     colClasses = "character")
```

```
> t2 <- t1[1:2004, c(1, 7, 8, 14)]
> t3 <- t2[complete.cases(t2), ]
> cnty1 <- t3[t3$V7 != "  ", ]
> ma1 <- t3[t3$V7 == "  ", c(1, 4)]
```

```
> cnty2 <- cnty1[which(!is.na(match(cnty1$V7, c("37", "45", "51")))),
+     ]
> cnty2$FIPS <- paste(cnty2$V7, cnty2$V8, sep = "")
```

We next break out an object with metro IDs, state and county IDs, and county names (`cnty1`), and an object with metro IDs and metro names (`ma1`). From there, we subset the counties to the three states, and add the FIPS string for each county, to make it possible to combine the new data concerning metro area membership to our combined county map. We create an object (`MA_FIPS`) of county metro IDs by matching the `cnty2` FIPS IDs with those of the counties on the map, and then retrieving the metro area names from `ma1`. These two variables are then made into a data frame, the appropriate row names inserted and combined with the county map, with method `spCbind`. At last we are ready to dissolve the counties belonging to metro areas and to discard those not belonging to metro areas, using `unionSpatialPolygons`:

```
> MA_FIPS <- cnty2$V1[match(FIPS, cnty2$FIPS)]
> MA <- ma1$V14[match(MA_FIPS, ma1$V1)]
> MA_df <- data.frame(MA_FIPS = MA_FIPS, MA = MA, row.names = FIPS)
> nc_sc_va90a <- spCbind(nc_sc_va90, MA_df)
> ncscva_MA <- unionSpatialPolygons(nc_sc_va90a, nc_sc_va90a$MA_FIPS)
```

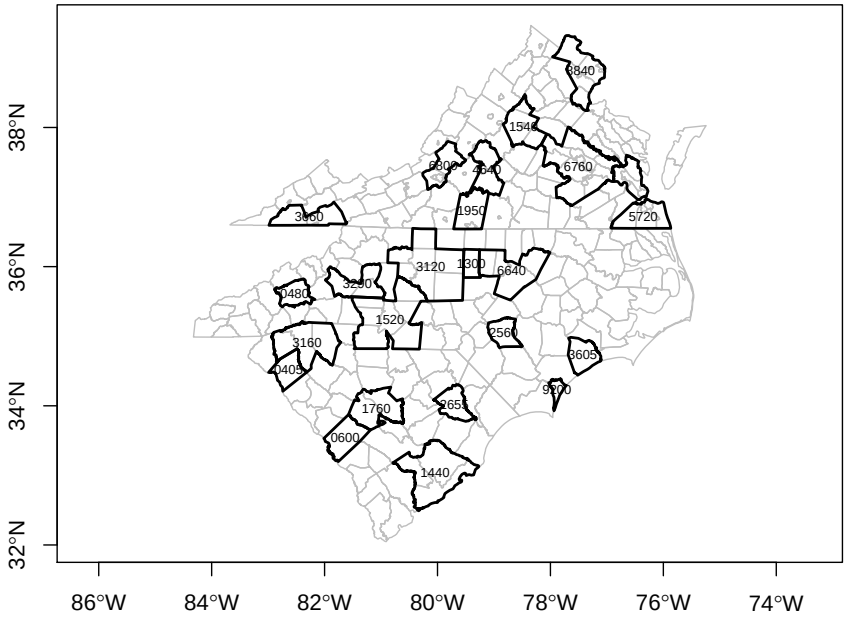

Figure 2: The three states with county boundaries plotted in grey, and Metropolitan area boundaries plotted in black; Metro area standard IDs are shown

Figure 2 shows the output object plotted on top of the cleaned input county boundaries. There does appear to be a problem, however, because one of the output boundaries has no name – it is located between 6760 and 5720 in eastern Virginia. If we do some more matching, to extract the names of the metropolitan areas, we can display the name of the area with multiple polygons:

```
> np <- sapply(slot(ncscva_MA, "polygons"), function(x) length(slot(x,
+     "Polygons")))
> table(np)

np
 1  2
22  1

> MA_fips <- row.names(ncscva_MA)
> MA_name <- ma1$V14[match(MA_fips, ma1$V1)]
> data.frame(MA_fips, MA_name)[np > 1, ]

   MA_fips                                  MA_name
18    5720  Norfolk-Virginia Beach-Newport News, VA MSA
```

The Norfolk-Virginia Beach-Newport News, VA MSA is located on both sides of Hampton Roads, and the label has been positioned at the centre point of the largest member polygon.