Applied spatial data analysis with R: status and prospects

Roger Bivand

19 February 2019

Introduction

Outline

- Spatial and spatio-temporal data are characterised by structures that distinguish them from typical tabular data
- The geometric structures also have spatial reference system information, and can adhere to standards, which may ease geometrical operations
- Satellite data and numerical model output data typically have regular grid structures, but these are often domain-specific
- Computationally intensive tasks include interpolation, upsampling, focal operations, change of support and handling vector data with very detailed boundaries, as well as modelling using Bayesian inference
- A further challenge to modelling using training sets with spatial data is how to split the observations in the presence of spatial dependence

Spatial and spatio-temporal data

Spatial data

Spatial data typically combine position data in 2D (or 3D), attribute data and metadata related to the position data. Much spatial data could be called map data or GIS data. We collect and handle much more position data since global navigation satellite system (GNSS) like GPS came on stream 20 years ago, earth observation satellites have been providing data for longer. (Geocomputation with R may be useful, as may SDSR).

```
> library(osmdata)
## Data (c) OpenStreetMap contributors, ODbL 1.0. http://www.openstreetmap.c
> library(sf)
## Linking to GEOS 3.7.1, GDAL 2.4.0, PROJ 5.2.0
> bbox <- opq(bbox = 'bergen norway')
> byb0 <- osmdata_sf(add_osm_feature(bbox, key = 'railway',
+   value = 'light_rail'))$osm_lines
> tram <- osmdata_sf(add_osm_feature(bbox, key = 'railway',
+   value = 'tram'))$osm_lines
> byb1 <- tram[!is.na(tram$name),]
> o <- intersect(names(byb0), names(byb1))
> byb <- rbind(byb0[,o], byb1[,o])
> library(mapview)
> mapview(byb)
```

Vector data

Spatial vector data is based on points, from which other geometries are constructed. Vector data is often also termed object-based spatial data. The light rail tracks are 2D vector data. The points themselves are stored as double precision floating point numbers, typically without recorded measures of accuracy (GNSS provides a measure of accuracy). Here, lines are constructed from points.

```
> all(st_is(byb, "XY"))
## [1] TRUE
> str(st_coordinates(st_geometry(byb)[[1]]))
##  num [1:14, 1:3] 5.33 5.33 5.33 5.33 5.33 ...
##  - attr(*, "dimnames")=List of 2
##  ..$ : chr [1:14] "4870663682" "331531217" "331531216" "331531215" ...
##   ..$ : chr [1:3] "X" "Y" "L1"
```

Raster data

Spatial raster data is observed using rectangular (often square) cells, within which attribute data are observed. Raster data are very rarely object-based, very often they are field-based and could have been observed everywhere. We probably do not know where within the raster cell the observed value is correct; all we know is that at the chosen resolution, this is the value representing the whole cell area.

```
> library(elevatr)
> elevation <- get_elev_raster(as(byb, "Spatial"), z = 14)
## Merging DEMs
## Reprojecting DEM to original projection
## Note: Elevation units are in meters.
## Note: The coordinate reference system is:
##  +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0,0,0
> mapview(elevation, col=terrain.colors)
```

Spatio-temporal data

Spatio-temporal data should be the rule but is not. Stacking things up over time to get a balanced stack is hard, and GIS representations do not handle time well. The LandGIS ESA CCI land cover series may also be downloaded (API coming), and displayed.

```
> library(rgdal)
> fl <- "../ESACCI-LC-L4-LCCS-Map-300m-P1Y-1992_2015-v2.0.7.tif"
> LUC <- readGDAL(fl, offset=c(10400,66450), region.dim=c(400, 700),
+   output.dim=c(400, 700))
##                   ../ESACCI-LC-L4-LCCS-Map-300m-P1Y-1992_2015-
v2.0.7.tif has GDAL driver GTiff
## and has 64800 rows and 129600 columns
> mapview(LUC[, , c(1,13,24)])
```

Representing spatial vector data in R (sp)

The **sp** package was a child of its time, using S4 formal classes, and the best compromise we then had of positional representation (not arc-node, but hard to handle holes in polygons). If we coerse **byb** to the **sp** representation, we see the formal class structure. Input/output used OGR/GDAL vector drivers in the **rgdal** package, and topological operations used GEOS in the **rgeos** package.

```
> library(sp)
> str(slot(as(st_geometry(byb), "Spatial"), "lines")[[1]])
## Formal class 'Lines' [package "sp"] with 2 slots
##   ..@ Lines:List of 1
##   .. ..$ :Formal class 'Line' [package "sp"] with 1 slot
##  .. .. .. ..@ coords: num [1:14, 1:2] 5.33 5.33 5.33 5.33 5.33 ...
##   .. .. .. .. ..- attr(*, "dimnames")=List of 2
##  .. .. .. .. .. ..$ : chr [1:14] "4870663682" "331531217" "331531216" "331
##   .. .. .. .. .. ..$ : chr [1:2] "lon" "lat"
##   ..@ ID   : chr "ID1"
```

Representing spatial vector data in R (sf)

The recent **sf** package bundles GDAL and GEOS (**sp** just defined the classes and methods, leaving I/O and computational geometry to other packages). **sf** used `data.frame` objects with one (or more) geometry column for vector data. The representation follows ISO 19125 (*Simple Features*), and has WKT (text) and WKB (binary) representations (used by GDAL and GEOS internally). The drivers include PostGIS and other database constructions permitting selection, and WFS for server APIs (**rgdal** does too, but requires more from the user).

```
> strwrap(st_as_text(st_geometry(byb)[[1]]))
## [1] "LINESTRING (5.333375 60.30436, 5.333386"
## [2] "60.30439, 5.333512 60.30463, 5.333664"
## [3] "60.30487, 5.3342 60.30559, 5.334472"
## [4] "60.30589, 5.334727 60.30613, 5.334901"
## [5] "60.30628, 5.33523 60.30652, 5.335494"
## [6] "60.30667, 5.335813 60.30682, 5.336282"
## [7] "60.30701, 5.336615 60.3071, 5.336872"
## [8] "60.30716)"
```

Representing spatial raster data in R (sp and raster)

The **raster** package S4 representation builds on the **sp** representation, using a `GridTopology` S4 object to specify the grid, and a data frame to hold the data. **raster** defines `RasterLayer`, and combinations of layers as stacks (may be different storage classes) or bricks (same storage class - array). Adding time remains an issue; **raster** could avoid reading data into memory using **rgdal** mechanisms.

```
> elevation
## class       : RasterLayer
## dimensions  : 5663, 3595, 20358485  (nrow, ncol, ncell)
## resolution  : 4.29e-05, 2.12e-05  (x, y)
## extent     : 5.229278, 5.383503, 60.2723, 60.39236  (xmin, xmax, ymin, yma
## coord. ref. : +proj=longlat +datum=WGS84 +no_defs +ellps=WGS84 +towgs84=0
##    data    source    :    /tmp/RtmpwtIir2/raster/r_tmp_2019-02-
14_101342_24305_44739.grd
## names       : layer
## values      : -5.561122, 605.8524  (min, max)
> slot(LUC, "grid")
##                              x            y
## cellcentre.offset 4.584722e+00 6.000139e+01
## cellsize          2.777778e-03 2.777778e-03
## cells.dim         7.000000e+02 4.000000e+02
```

Representing spatial raster data in R (sf and stars)

The new **stars** - Scalable, Spatiotemporal Tidy Arrays - package started looking at array structures and has built-in proxy data. Like **sf**, the development of **stars** has been supported by the R Consortium, and **stars** uses the infrastructure of **sf** to use GDAL for input/output and manipulation. In **sf**, the interface to the C++ GDAL library is based on **Rcpp**, which was not available when **rgdal** was written (plot from colour table needs current **stars** master).

```
> library(stars)
## Loading required package: abind
> LUC_stars <- read_stars(fl, proxy=TRUE)
> LUC_stars
## stars_proxy object with 1 attribute in file:
## $`ESACCI-LC-L4-LCCS-Map-300m-P1Y-1992_2015-v2.0.7.tif`
## [1] "../ESACCI-LC-L4-LCCS-Map-300m-P1Y-1992_2015-v2.0.7.tif"
##
## dimension(s):
##      from     to offset       delta
## x       1 129600   -180  0.00277778
## y       1  64800     90 -0.00277778
## band    1     24     NA          NA
##                            refsys point values
## x    +proj=longlat +datum=WGS8... FALSE   NULL
## y    +proj=longlat +datum=WGS8... FALSE   NULL
## band                          NA    NA   NULL
##
## x    [x]
## y    [y]
## band
> LUC_stars1 <- st_as_stars(LUC_stars[byb,,,])
> legs <- read.csv("../ESACCI-LC-Legend.csv", header = TRUE, sep = ";")
> plot(LUC_stars1, band=24, rgb = legs, main="")
```

Spatial reference systems

Baseline WKT and PROJ4

Spatial reference systems define how the geoid is viewed (prime meridian, ellipsoid, datum), and, if projected to the plane, where we are (central longitude, latitude, offsets, etc.). They also define the units - **sf** incorporates smart units handling. Projection (no datum change) and transformation are possible using PROJ and its `proj_api.h` interface directly (`rgdal::spTransform()` and `lwgeom::st_transform_proj()`), or through GDAL (`sf::st_transform()`).

```
> (WKT <- st_crs(byb))
## Coordinate Reference System:
##   EPSG: 4326
##   proj4string: "+proj=longlat +datum=WGS84 +no_defs"
> strwrap(gsub(",", ", ", st_as_text(WKT)))
## [1] "GEOGCS[\"WGS 84\", DATUM[\"WGS_1984\","
## [2] "SPHEROID[\"WGS 84\", 6378137, 298.257223563,"
## [3] "AUTHORITY[\"EPSG\", \"7030\"]],"
## [4] "AUTHORITY[\"EPSG\", \"6326\"]],"
## [5] "PRIMEM[\"Greenwich\", 0, AUTHORITY[\"EPSG\","
## [6] "\"8901\"]], UNIT[\"degree\", 0.0174532925199433,"
## [7] "AUTHORITY[\"EPSG\", \"9122\"]],"
## [8] "AUTHORITY[\"EPSG\", \"4326\"]]"
> byb_utm <- st_transform(byb, crs=32632)
> st_crs(byb_utm)
## Coordinate Reference System:
##   EPSG: 32632
##  proj4string: "+proj=utm +zone=32 +datum=WGS84 +units=m +no_defs"
```

Escaping the WGS84 hub: PROJ 6.0 and OGC WKT2

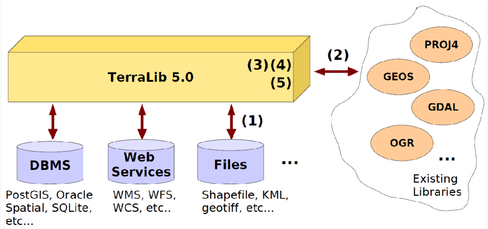
Changes in the legacy PROJ representation and WGS84 transformation hub have been coordinated through the GDAL barn raising initiative. The syntax is changing to pipelines, but crucially WGS84 will often cease to be the pivot for moving between datums. A new OGC WKT is coming, and an SQLite EPSG file database will replace CSV files. SRS will begin to support 3D by default, adding time too as SRS change.

```
> head(st_coordinates(st_geometry(byb)[[1]]), n=1)
##                   X        Y L1
## 4870663682 5.333375 60.30436  1
> system(paste("echo 5.333375 60.30436 |",
+   "proj +proj=pipeline +ellps=WGS84",
+   "+step +init=epsg:32632"), intern=TRUE)
## [1] "297436.05\t6690941.01"
> head(st_coordinates(st_geometry(byb_utm)[[1]]), n=1)
##             X       Y L1
## [1,] 297436.1 6690941  1
```

Earth observation

Volume and frequency

As earth observation (and other remote sensor) data volume (resolution) and frequency has increased, legacy technologies of downloading (usually calibrated) scenes to multiple user systems have been struggling to keep up. Terralib is an early example of trying to keep data in databases, but maybe using WMS and other API technologies.



Cloud storage

The **elevatr** example earlier shows a specialised use. They build on data being kept in the cloud, and accessed by selection through an API. There is also some client-server code in **stars**, needing to be run in two separate, concurrent R sessions:

```
> library(plumber)
> r <- plumb(system.file("plumber/server.R", package = "stars"))
> r$run(port=8000)
> source(system.file("plumber/client.R", package = "stars"),,
+   echo = TRUE)
> plot(xx)
```

Cloud storage

OpenEO proposes proof-of-concept client-server API approaches (and this issue)

```
> library(openeo) # "master" branch
> euracHost = "http://saocompute.eurac.edu/openEO_0_3_0/openeo/"
> eurac = connect(host = euracHost,disable_auth = TRUE)
## Connected to host
> pgb = eurac %>% pgb()
> bb <- list(west=10.98,east=11.25,south=46.76,north=46.58)
> tt <- c("2017-01-01T00:00:00Z","2017-01-31T00:00:00Z")
> task = pgb$collection$S2_L2A_T32TPS_20M %>%
+   pgb$filter_bbox(extent=bb) %>%
+   pgb$filter_daterange(extent=tt) %>%
+   pgb$NDVI(red="B04",nir="B8A") %>%
+   pgb$min_time()
> tf <- tempfile()
> raster = eurac %>% preview(task=task, format="GTiff",
+   output_file=tf)
> mapview(raster(raster))
```

Computationally intensive tasks

Data handling

We can download monthly CSV files of city bike use, and manipulate the input to let us use the **stplanr** package to aggregate origin-destination data. One destination is in Oslo, but otherwise things are OK. We can use CycleStreets to route the volumes onto OSM cycle paths, via an API and API key. We'd still need to aggregate the bike traffic by cycle path segment for completeness.

```
> bike_fls <- list.files("../bbs")
> trips0 <- NULL
> for (fl in bike_fls) trips0 <- rbind(trips0,
+   read.csv(file.path("../bbs", fl), header=TRUE))
> trips0 <- trips0[!(trips0[,9]==84),]
> trips <- cbind(trips0[,c(1, 4, 2, 9)], data.frame(count=1))
> from <- unique(trips0[,c(4,5,7,8)])
> names(from) <- substring(names(from), 7)
> to <- unique(trips0[,c(9,10,12,13)])
> names(to) <- substring(names(to), 5)
> stations0 <- st_as_sf(merge(from, to, all=TRUE),
+   coords=c("station_longitude", "station_latitude"))
> stations <- aggregate(stations0, list(stations0$station_id),
+   head, n=1)
> suppressWarnings(stations <- st_cast(stations, "POINT"))
> st_crs(stations) <- 4326
> od <- aggregate(trips[,-(1:4)], list(trips$start_station_id,
+   trips$end_station_id), sum)
> library(stplanr)
> od_lines <- od2line(flow=od, zones=stations, zone_code="Group.1",
+   origin_code="Group.1", dest_code="Group.2")
> mapview(od_lines, alpha=0.2, lwd=(od_lines$x/max(od_lines$x))*10)
> Sys.setenv(CYCLESTREET="xXxXXxXxXxXxXxXxX")
> od_routes <- line2route(od_lines, "route_cyclestreet",
+   plan = "fastest")
> ## od_routes <- readRDS("../od_routes.rds")
> mapview(od_routes, alpha=0.2, lwd=(od_lines$x/max(od_lines$x))*10)
```

Interpolation

Interpolation from data locations to other locations, also for re-sampling data resolution for raster data, is often computationally intensive. Predicive Soil Mapping with R uses random forest approaches to prediction, but this isn't new, see this example from 2003.

```
> library(geoR)
> data(SIC)
> library(sp)
> sic.100SP <- SpatialPointsDataFrame(SpatialPoints(sic.100$coords),
+  data=data.frame(precip=sic.100$data, altitude=sic.100$altitude))
> sic.allSP <- SpatialPointsDataFrame(SpatialPoints(sic.all$coords),
+  data=data.frame(precip=sic.all$data, altitude=sic.all$altitude))
> names(sic.allSP) <- c("precip", "altitude")
> sic.367SP <- sic.allSP[which(!rownames(sic.all$coords) %in%
+   rownames(sic.100$coords)),]
> library(automap)
> aK <- autoKrige(formula=precip ~ altitude, input_data=sic.100SP,
+   new_data=sic.367SP)
## [using universal kriging]
> res <- aK$krige_output
```

Interpolation

```
> library(rgeos)
> dist0 <- as.data.frame(gDistance(sic.100SP["precip"],
+   sic.100SP["precip"], byid=TRUE))
> names(dist0) <- paste("layer", names(dist0), sep=".")
> dist1 <- as.data.frame(gDistance(sic.100SP["precip"],
+   sic.367SP["precip"], byid=TRUE))
> names(dist1) <- paste("layer", names(dist1), sep=".")
> rm.precip <- cbind(as.data.frame(sic.100SP)[c("precip", "altitude")],
+   dist0)
> rm.precip1 <- cbind(as.data.frame(sic.367SP)[c("altitude")], dist1)
> dn0 <- paste(names(dist0), collapse="+")
> fm0 <- as.formula(paste("precip ~ altitude +", dn0))
> library(ranger)
> m.precip <- ranger(fm0, rm.precip, quantreg=TRUE, num.trees=150,
+   seed=1, keep.inbag=TRUE)
> res$rf_pred <- predict(m.precip, rm.precip1,
+   type="response")$predictions
> res$rf_sd1 <- predict(m.precip, rm.precip1, type="se",
+   se.method="infjack")$se
> res$rf_sd2 <- predict(m.precip, rm.precip1, type="se",
+   se.method="jack")$se
> spplot(res, zcol=c(1,4))
> spplot(res, zcol=c(3,5,6))
```

Inference

Spatial CV

Inference

Conclusions

Conclusions

-
-
-
-