Spatial Interpolation Notes

January 5, 2021

1 Introduction

We want to use interpolation because it is reasonable to assume that spatially distributed variables are also spatially correlated. It is not always true, but often worth exploring as part of an analysis. There are multiple different methods to interpolate data that depend on different underlying assumptions. These methods are described below.

Information from ARCGIS and Applied Spatial Data Analysis With R (2013) unless listed otherwise

There are two main methods used to interpolate data and estimate a surface for geospatial data:

- Inverse distance weighting (IDW) and spline methods
- Kriging

IDW and splines are deterministic interpolation methods as they are directly based on the surrounding values or smoothed formulas. Kriging is different as it uses autocorrelation and takes position into account in the statistical models. Kriging uses a certain number of neighbouring points, or all points within a specified radius (cf kNN).

The general formula for IDW and kriging is:

$$\hat{Z}(s_0) = \sum_{i=1}^{N} w(s_i) Z(s_i)$$

where: $Z(s_i)$ = the measured value at the ith location
$w(s_i)$ = an unknown weight for the measured value at the ith location
s_0 = the prediction location
N = the number of measured values

The difference between IDW and kriging is that in IDW, $w(s_i)$ only depends on distance to prediction location. In kriging $w(s_i)$ also depends on autocorrelation i.e. spatial relationship between prediction locations.

2 Inverse distance weighting (IDW)

IDW determines cell values using a linearly weighted combination of surrounding values. The weights are function of the inverse distance. The general form for the IDW function is:

$$\hat{Z}(s_0) = \frac{\sum_{i=1}^{n} w(s_i) Z(s_i)}{\sum_{i=1}^{n} w(s_i)}$$

where: $Z(s_i)$ = the measured value at the ith location
$w(s_i) = ||s_i - s_0||^{-p}$
$||\cdot||$ = Euclidean distance
p = an inverse distance weighting power, defaulting to 2

The value of p determines how much closer values are preferred. As p increases, IDW approaches a one-nearest-neighbour interpolation model. p can be selected using cross-validation.

Another way to control IDW interpolation is through selecting the number of neighbouring observations to include. This can improve speed of interpolation, and may be used when there is reason to believe that distant points have little correlation. There are two approaches for varying the number of points used for interpolation:

1. Varying search radius
    - The number of points to include is fixed, and the radius changes to include that set number
    - Depends on the density of observations fluctuating
    - The maximum radius can also be set, in which case all points will be included if that max radius is reached before n
2. Fixed search radius
    - Set a radius and minimum number of points
    - If $n <$ minimum number of points at set radius, the radius increases until the minimum is reached.

In addition to these two approaches, barriers can be created to limit the searches for neighbouring points, i.e. only search for this side of a river.

3 Kriging

One of the key benefits of kriging is that in addition to using autocorrelation, it is able to estimate uncertainty in the interpolation. It can do this because it is based on a spatial arrangement of empirical observations, rather than a presumed model of spatial distribution. Although kriging preferentially weights closer observations, its use of autocorrelation means that clusters are not over-fit i.e. lowering bias as each point in a cluster provides less information than a single point.

The kriging predictor is an "optimal linear predictor" and an exact interpolator. This means that prediction error is each interpolated value is calculated to minimize the prediction error for that point. It also means that the interpolated value for sampled points is equal to the actual value, and all interpolated values will be the Best Linear Unbiased Predictors (BLUPs).

Kriging is only helpful where there is at least moderate spatial autocorrelation. If there is not, then simpler methods like IDW, will generally perform as well as kriging.

3.1 Assumptions in kriging

Information for assumptions from Columbia

For kriging to be used, there are a number of assumptions/conditions to be met. These conditions can be checked in exploratory data analysis.

1. Assumption of intrinsic stationarity
    - Means that the joint probability distribution does not vary across the study space, so the same parameters (e.g. mean, range and sill etc) are valid across the space

- Means one variogram is valid across the space

2. Assumption of isotropy
    - Uniformity in all directions (semivariance identical in all directions)

By making these assumptions, we are assuming that the samples are randomly generated by the function $Z(s)$ with a mean (m) and residual ($e(s)$).

$$Z(s) = m + e(s)$$

where: $E\left(Z(s)\right) = m$

The assumption of *intrinsic stationarity* and *isotropy* can be relaxed to create models where the mean varies spatially. In instances like this, the measured values can be assumed to be randomly generated by a linear function of known predictors $X_j(s)$.

$$\begin{aligned} Z(s) &= \sum_{j=0}^{p} X_j(s)\beta_j + e(s) \\ &= X\beta + e(s) \end{aligned} \tag{1}$$

3.2 Creating a prediction map with kriging

There are two steps:

1. Create the variograms and covariance functions to estimate the spatial autocorrelation values that depend on the model of autocorrelation (fitting a model).
2. Predict the unknown values

3.2.1 Variography (spatial modelling/structural analysis)

There are often too many pairs of spatial points to calculate and plot the distance for each pair. Instead, spatial distances are put into lag bins i.e. all points in the range $40m < h \leq 50m$ of point A, and calculate the semivariance. The semivariance is equal to half the variance of the differences between all possible points spaced a constant distance apart. Assuming *isotropy* and *intrinsic stationarity*, we can generalise the distances between points and use the distance $||h||$ rather than the vector $\mathbf{h}$, i.e. use bins.

$$\hat{\gamma}(\tilde{h}_j) = \frac{1}{2N_h} \sum_{i=1}^{N_h} \left(Z(s_i) - Z(s_i + h)\right)^2, \ \forall h \in \tilde{h}_j$$

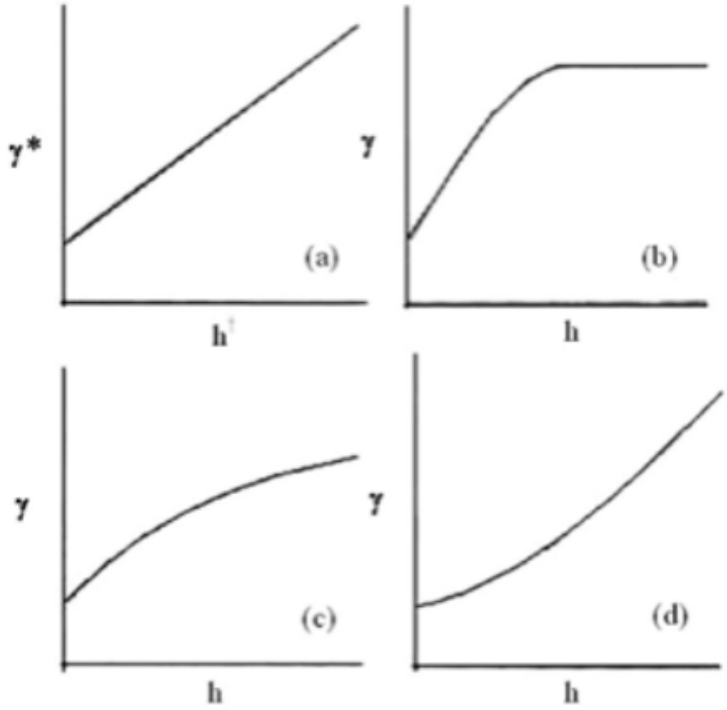
where: $Z(s_i)$ = the measured value at the ith location
$\hat{\gamma}(\tilde{h}_j)$ = sample variogram
N_h = sample data points
$\tilde{h}_j$ = distance bins (intervals)

Plotting the distance vs semivariance produces an empirical semivariogram. Closer items should be more similar, therefore lower semivariance. The opposite is true for further points.

A model is fit to the empirical semivariogram (cf regression). Different types of models can be fit the the semivariogram, and the optimal model can be selected using metrics like RMSE, MLE, and Bayesian methods:

- Spherical (most common)
- Circular
- Exponential
- Gaussian
- Linear

Figure 1: Different types of models used in spatial modelling (Poilou 2008). a) Linear semi-variogram; (b) spherical semi-variogram; (c) exponential semi-variogram; and (d) power semi-variogram



There are a number of key points on the figures:

- Range
  - The Range is the point at which the semivariance first levels off
  - Items within the range are autocorrelated (distance matters)
  - Items outside the range are not autocorrelated (distance no longer changes the semivariance)
- Sill
  - The Sill is the height at which the semivariance levels off to
- Nugget
  - The minimum value of semivariance ($\gamma(h = 0)$)
  - Theoretically there is no semivariance when $h = 0$, but in reality it is present due to measurement error or spatial sources of variation at distances smaller than the sample interval (or both)
- Partial Sill
  - Amount of semivariance between Sill and the Nugget

3.2.2 Predictions

Now a model has been fit to the semivariance and autocorrelation can be observed, predictions can be made within the domain. Kriging differs from IDW as it uses the semivariogram to calculate the weights. There are a number of methods used in kriging:

1. Ordinary kriging
    - Assumes the constant mean is unknown
2. Universal kriging
    - Assumes there's a prevailing trend, relaxing the assumption of stationarity for the mean, but maintaining a constant variance
    - Trend is modelled with polynomial function, and subtracted from observed
    - Semivariogram is modelled on the residuals to produce autocorrelations
3. Block kriging
    - Estimates averaged values over gridded "blocks" rather than single points
    - These blocks often have smaller prediction errors than are seen for individual points
4. Covariate kriging
    - Additional observed variables (which are often correlated with each other and the variable of interest) are used to enhance the precision of the interpolation of the variable of interest at each location
5. Poisson kriging
    - Used for incidence counts and disease rates

3.3 Limitations

Information for limitations from Columbia

There are a number of limitations of kriging.

1. Since the weights of the kriging interpolator depend on the modeled variogram, kriging is quite sensitive to misspecification of the variogram model
2. Similarly, the assumptions of the kriging model (e.g. that of second-order stationarity) may be difficult to meet in the context of many environmental exposures
    - Some newer methods (e.g. Bayesian approaches) have thus been developed to try and surmount these obstacles
3. In general, the accuracy of interpolation by kriging will be limited if the number of sampled observations is small, the data is limited in spatial scope, or the data are in fact not amply spatially correlated
    - In these cases, a sample variogram is hard to generate, and methods such as land-use regression may prove preferable to kriging for spatial prediction

4 Natural Neighbour

Natural neighbour is a local method that examines samples near the point of interest and evaluates the relative overlap with their areas. The relative overlaps are then used to create the weights for interpolation. Because of this, it is also known as "area-stealing" (Sibson) interpolation. Natural neighbour interpolation therefore does not infer trends that are not already present in the data, and the surface passes through the points, and is smooth in between.

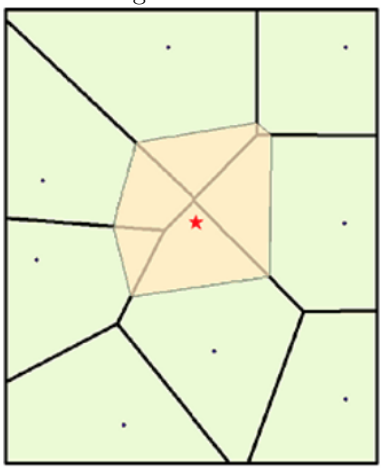
The areas are called Voronoi (Thiessen) polygons. Voronoi polygons are created by examining the space around points and drawing the boundary so that every place inside the boundary is closest to the polygon's point than any other. Formally this is written as:

$$R_k = \{x \in X \mid d(x, P_k) < d(x, P_j),\ \forall j \neq k)\}$$

where: R_k = Voronoi polygon of point k
P_k = Point k
P_j = Neighbouring point j

An example of this can be seen in the figure below.

Figure 2: Natural neighbour method of interpolation



5 Splines

Splines are a smoothing function that pass through all the input points and attempt to create a smooth surface between them. As such, it is best for gently varying surfaces e.g. pollution concentrations. The surface is fit to a specified number of neighbouring input points. The basic spline is also known as a thin plate interpolation. There are two conditions that minimum curvature splines must follow:

1. The surface must pass through all data points
2. The surface must have minimum curvature i.e. minimize the cumulative sum of squares of the second derivative terms of the surface at each point

One possible issue with thin plate interpolation is that there may be rapid change in first derivatives around each data point. Increasing the number of points used for interpolation can help to smooth the surface as the cell is influenced by a greater number of more distant points. Splines create rectangular regions of equal size, with the same number in the $x-$ and $y-$ directions. Each region must contain at least 8 points, but different densities resulting from data that is not uniformly distributed can lead to regions containing different numbers of points.

Generally, the spline formula is:

$$S(x,y) = T(x,y) + \sum_{j=1}^{N} \lambda_j R(r_j)$$

where: N = total number of points to be used in interpolation
λ_j = coefficients found by the solution of a system of linear equations
r_j = the distance from the point (x,y) to the jth point

There are two spline types, which define the terms $T(x,y)$ and $R(r_j)$ differently.

5.1 Regularized splines

A regularized spline creates a smooth and gradually changing surface, allowing values outside those observed in the data.

$$T(x,y) = a_1 + a_2 x + a_3 y$$

where: a_i = coefficients found by the solutions of a system of linear equations

and,

$$R(r) = \frac{1}{2\pi}\left\{\frac{r^2}{4}\left[\ln\left(\frac{r}{2\tau}\right) + c - 1\right] + \tau^2\left[K_0\left(\frac{r}{\tau}\right) + c + \ln\left(\frac{r}{2\pi}\right)\right]\right\}$$

where: r = the distance between the point and the sample
τ^2 = the Weight parameter
K_0 = the modified Bessel function
c = a constant equal to 0.577215

In regularized splines, the Weight parameter (τ^2) specifies the weights attached to the third derivatives terms during minimization. Larger weights result in smoother surfaces and smooth first-derivative surfaces. Typical values range between 0 and 0.5.

5.2 Tension splines

A tension spline creates a less smooth surface with values more tightly constrained by the sample data range.

$$T(x,y) = a_1$$

where: a_1 = a coefficient found by the solutions of a system of linear equations

and,

$$R(r) = -\frac{1}{2\pi\varphi^2}\left[\ln\left(\frac{r\varphi}{2}\right) + c + K_0(r\varphi)\right]$$

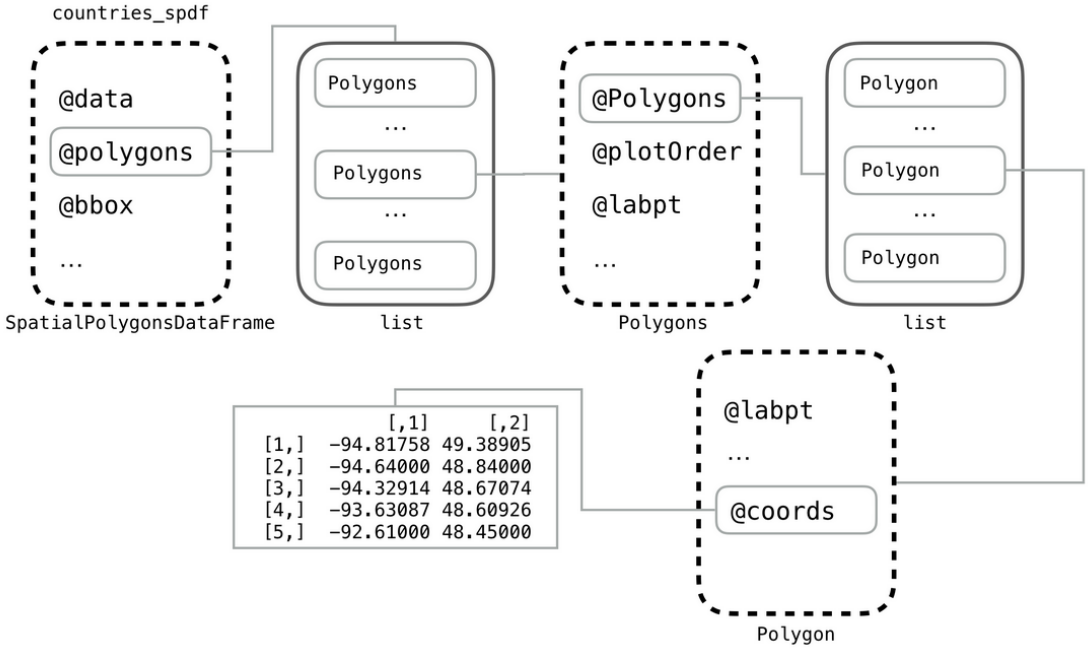
where: r = the distance between the point and the sample
φ^2 = the Weight parameter
K_0 = the modified Bessel function
c = a constant equal to 0.577215

The tension method differs from regularized splines as it attaches the Weight parameter (φ^2) to first-derivative terms, not third-derivative terms. Larger values of φ^2 lower the tension and result in a coarser surface as the first-derivative surface is not smooth, passing through all the points. $\varphi^2 = 0$ results in a basic thin plate surface. Typical values range between 0 and 10.

6 Datacamp: Visualizing geospatial data in R

- *ggmap* package very useful for quickly producing static spatial plots
  - *ggmap:get_map(long, lat)* pulls basemap based lat long
  - *ggmap(*ggmap*, base_layer = ggplot(df, aes(long, lat))) + geom_point()* allows you to plot layers over map and retain same aes() e.g. for faceting
- Different types of spatial data
  - Point data
  - Line data - assumes points connected by straight lines
  - Polygon data
    - Data associated with enclosed area of points
    - *ggplot2::geom_poly()*
  - Raster (grid) data
    - Regular grid specified by origin and steps in x and y axis, and data is associated with cells in grid
    - *ggplot2::geom_tile(aes(fill = *var*))* used to create raster
- Polygon data
  - Difficult to described
    - Order of joining up points matters
    - Polygons may be broken up e.g. by river therefore needing multiple polygons to describe it
- *sp* data structures better that *dataframes* for storing spatial data as don't have to repeat info like groups and order for polygons, and contains information about the coordinate sytem itself, which is useful when working with multiple systems/for sharing
- *spdf* is an **S4** data type
  - Useful adaptation of *sp* structure as also contains dataframe
  - Items are **slots** that are accessed with the @ symbol e.g. *spdf@polygon*
    - Each **slot** contains a list that is *another* **S4** object (e.g. Polygons) (see Fig. 3)
    - Can pull information as with normal dataframes using $ symbol e.g.
      - *is_nz = countries_spdf$name == "New Zealand"*
      - *nz = countries_spdf[is_nz,]*
- *tmap* package designed to plot spatial data, rather than requiring dataframe format, like *ggplot2*
  - *tm_shape()* adds basemap
  - *tm_raster()* creates choropleth for rasters
  - *tm_fill()* creates choropleth for polygons
  - Can save interactive *leaflet* map using *tmap_save(filename = *.html)*
- *raster* package better to work with raster data than *sp* and *ggplot2*

Figure 3: *spdf* data structure



  - Creates an **S4** object
  - More efficient as stores data in matrix like format, where each value is associated with a cell in the raster grid
    * Multiple matrices act as layers to provide more information
      · *Multi-band/multi-layer* rasters
      · e.g. single band for red/green/blue light to produce colours
    * Reduces reproducing the same grid
  - *rasterVis::levelplot()* good for quickly visualizing rasters

- *classInt::classIntervals()* useful to bin continuous variables for choropleths
- *rgdal::readOGR* used to read in shape files (polygons)
- *proj4string()* allows you to define the coordinate reference system (CRS) and projection when no present, or print it where it is present
- *rgdal::spTransform()* used to transform CRS
  - *tmap* does the transformation automatically
- *sp::merge(spdf, df, by.x, by.y)* used to add new information to sp dataframe
  - *spdf* have information on x number of **poylgons** (*not observations, like a normal df*)
  - Both *@data* and *@polygon* slots contain an *@ID* slot to match the data to the polygons
    * Need to make sure you match them when adding new data otherwise may become unordered

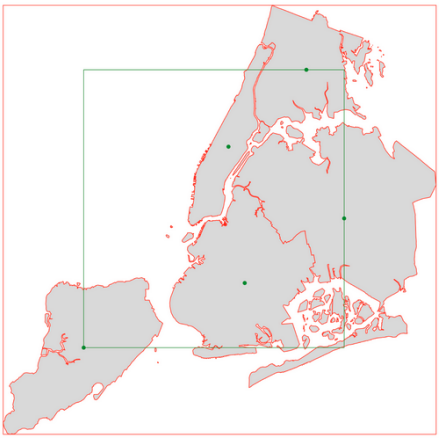
7 Datacamp: Spatial analysis with sf and raster in R

- *sf* package new way to read and use spatial information
  - *sf::st_read("file.sh")* reads shape files into dataframe with special *geometry* **list column**
    * Means they can be manipulated like standard dfs, e.g. using *dplyr* etc
  - Also save **metadata** that can be seen with *head()* call

- *raster*
  - *raster::brick("file.tif")* reads in single band raster data
  - *raster::raster("file.tif")* reads in single band raster data
- *sf::st_area()* calculates area of individual features
  - Produces a vector **with units** in metadata
    - Means you have to use *unclass()* function to remove units for calculations e.g. *df %>% filter(unclass(areas) > 3000)*
- *sf::st_area()* calculates area of individual features
- *sf::st_length()* calculates length of individual
- *raster::plotRGB()* used to plot multi-band raster images quickly (if bands correspond to RGB)
  - *plot()* creates plots for each band separately
- To **add** CRS information:
  - For *sf* data:
    - *sf::st_crs()* shows you the current CRS information
      - *$epsg:* gives numeric code
      - *$proj4string:* gives string code
    - *sf::st_crs()* can assign CRS
      - *sf::str_crs(sf) <- 4236* for EPSG
      - *sf::str_crs(sf) <- "+proj=..."* for proj4string
  - For *raster* data:
    - *crs()* shows and assigns current CRS information using proj4string
      - *crs() <- "+proj=..."*
- To **change** CRS information:
  - *sf::st_transform(sf, crs = other_crs(raster, asText=TRUE))* for vectors (*sf* data)
    - Can use either EPSG or proj4string
    - *asText=TRUE* required to force crs from raster format to text when using a mix of polygons and rasters
  - *raster::projectRaster()* for rasters
    - When specifying CRS with EPSG, must use *projectRaster(raster, crs = "+init=epsg:32618")*
    - Use proj4string as normal (*projectRaster(rater, crs = proj4string)*)
- *sf::st_cast(, "MULTIPOINT")* used to *cast* polygons in *$geometry* to bundles of points (MULTIPOINT) to then calculate number of vertices
  - *sum(sapply(pts, length))*
- *sf::st_simplify()* used to simplify spatial data by reducing number of vertices so will be processed much faster
- *methods::as(sf, Class = "Spatial")* converts *sf* object to *sp* object
- *sf::st_as_sf()* converts *sp* object to *sf* object
  - Also used to convert dataframe of coordinates to *sf* object
  - *sf::st_as_sf(pts, coords = c("lon", "lat"), crs = proj4string/EPSG)*
    - Longitude must be listed first
- *sf::st_write(sf, "sf.csv", layer_options = "GEOMETRY=AS_XY")* to write *sf* object to csv with coordinate information
- *raster::aggregate(raster, fact = factor, fun = function)* used to reduce resolution of raster by a certain factor in each direction (x and y) and function (e.g. taking the mean value)
- *raster::reclassify(raster, rcl = reclas_matrix)* reclassifies values in ranges to a new value (all specified in matrix)

- Should use projected CRS when doing analysis as will use common distances like meters vs degrees in unprojected CRS
  - Make sure all layers have the same CRS so calculations make sense and are aligned
- *sf::st_buffer(sf, dist = x)* Create a buffer of xm around a point/section of a spatial plot
  - Useful for identifying objects within that buffer
- *sf::st_centroid(sf)* useful for calculating geographic centre of polygons
- Defining regions
  - Bounding box
    - *sf::st_bbox(poly)* used to calculate points for bounding box
    - *sf::st_make_grid(poly, n = 1)* used to create a bounding box around the polygons (or centroids if specified etc)
      - $n = 1$ specifies only want one polygon, rather than multi-row, multi-column grid

Figure 4: Bounding box example



  - Dissolving features
    - *sf::st_union()*
      - Dissolves polygons into a single polygon
      - Clusters individual points into a MULTIPOINT geometry
  - Convex Hull
    - Used to create tighter bounding box around points
      - Need to cluster individual points (e.g. centroids) into MULTIPOINTS before drawing convex hull
    - Can be created around buffers that overlap

```
# Buffer the beech trees by 3000
beech_buffer <- st_buffer(beech, dist = 3000)

# Limit the object to just geometry
beech_buffers <- st_geometry(beech_buffer)

# Dissolve the buffers
beech_buf_union <- st_union(beech_buffers)

# Plot the dissolved buffers
plot(beech_buf_union)

# Create the convex hull
beech_hull <- st_convex_hull(beech_buf_union)
```

Figure 5: Convex hull example

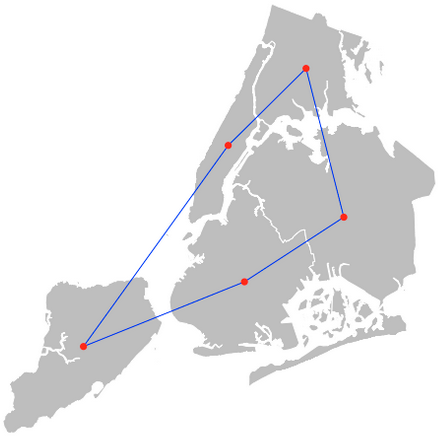
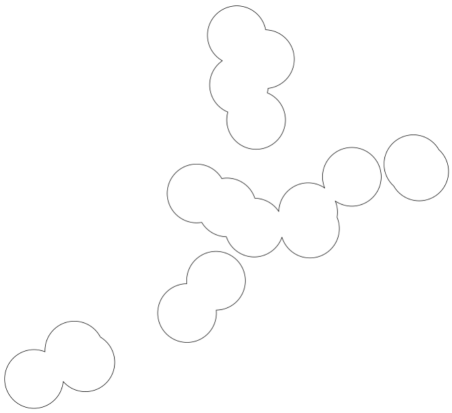


Figure 6: Overlapping buffers that can be used in a convex hull



- Multilayer geoprocessing and relationships
  - *sf::st_join(out_df, attr_df)* used to join two dataframes with spatial information when there are not columns that can be used to link them, so need to join based on geographic information
    - First object is the df you want as the output (contains most useful information)
    - Second object is the df that contains the attributes you want added
  - *sf::st_intersects(large_df, small_df)* shows which geographies intersects
    - *sf::st_intersection(large_df, small_df)* clips the resulting list so only those that intersect are shown, and not the rest of the polygons
  - *sf::st_contains(large_df, small_df)* shows which geographies are completely contained in another polygon
  - *sf::st_distance(feat1, feat2)* calculates distances between features
- Geoprocessing with rasters
  - *raster::mask(raster, mask = poly)* to mask all areas but the polygons of interest
  - *raster::crop(raster, mask = poly)* to crop the raster to just include the polygons (cf. bounding box)
  - *raster::extract(raster, poly, fun = mean)* extracts the values from polygons as a list using a function (e.g. NULL returns all values, and mean returns the means from each polygon)
  - *raster::overlay(raster1, raster2, fun = func)* allows you to perform functions to rasters e.g. first raster contains elevation, second contains the multiplication values, and the function says to multiple the two rasters together
- *ggplot2::geom_sf()* can create maps of *sf* data
  - *aes(fill = var)* used to create choropleths
- *tmap::tmap_arrange(map1, map2, nrow)* allows you to plot multiple maps in a grid

8 Datacamp: Spatial statistics in R

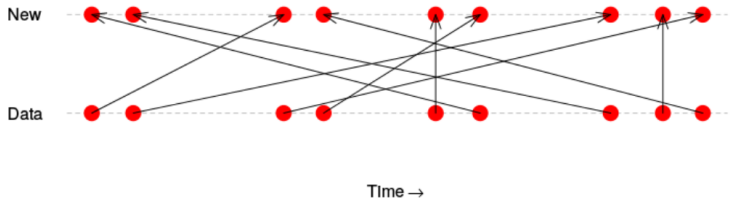
- The 'window' is the study area where events (e.g. points) occur
- Spatial point pattern is the set of observed events and the window
- Spatial point process is stochastic process that generates the points in a window
- Most of spatial point pattern analysis is trying to infer the spatial point process that would produce the dataset (pattern)
- The 'mark' are the values observed e.g. height of tree etc
- *spatstat* package very useful
  - Stores spatial point patterns in *ppp* objects (planar point pattern)
- Quadrat test used to test spatial randomness
  - Split window into even subregions
  - Count number of events in each subregions
  - Plot histogram of event counts vs number of quadrants
  - Should follow Poisson distribution if random
  - Use Chi-squared test for observed vs expected
  - *spatstat::quadrat.test()*
- Generally, points can be generated in two different manners:
  1. Regular (inhibitory) processes that are completely random and follow a uniform Poisson distribution
     - *spatstat::rpoispp()* useful for generating Poisson point process i.e. spatially random points
  2. Clustered processes where points occur more together than under a uniform Poisson process
     - Thomas process is a clustered pattern where 'parent' points are uniformly distributed and 'child' points are clustered around the parent
       - *spatstat::rThomas()* useful for generating a Thomas process
     - Strauss process is a clustered pattern with a *repulsive* point pattern i.e. no points can be closer than a threshold
       - *spatstat::rStrauss()*
       - Sometimes called regularly spaced
- Quadrat test can lose power with too few or too many subregions
- Other tests don't rely on arbitrary subregions and rely on estimated properties of the process
  - Nearest neighbour distributions
    - Plot histogram of NN distances
    - Plot cumulative distribution of NN distances (*G(r)*) (*spatstat::nndist()*)
    - Poisson distribution can be used to calculate cumulative distribution curve ($G(r) = 1 - exp(-\lambda\pi r^2)$)
    - *spatstat::Gest()* estimates function *G(r)*
    - Need to use edge-correction as events near the edge of the window have less area for a nearest neighbour
  - Ripley's reduced second moment measure (K function)
    - K is the number of expected events to occur within a given distance scaled by the intensity
    - Calculate for all points and plot events vs distance (*spatstat::Kest()*)
    - For completely random event, should be $\pi \times d^2$
    - Needs edge corrections, like *G(r)* test
    - Use Monte-Carlo test
      - Simulate spatially random point patterns in your window and calculate K function plots, before overlaying observed K function

        - Calculate *envelope* (the minimum and maximum values at each distance) *spatstat::envelope(ppp, Kest, correction)*
        - If larger than simulations at any point - indication of clustering at distance
        - If smaller than simulations - indication of inhibitory process
        - If use 99 simulations can calculate p-value easily (1/(99+1)) (assuming it is the largest value)
        - $plot(K\_cluster\_env, . - pi * r^2 \sim r)$ used to subtract expected K function of Poisson process to help visualize

- Bivariate point pattern
  - Point pattern map with colors representing different outcomes e.g. cholera vs not cholera
- Cross nearest neighbour function
  - Similar to NN but measure nearest distances between **case and control** events
- Cross K-function
  - Similar to K-function, but number of control events expected within radius of case events
- Null hypothesis for cross-functions is that the proportion of each type is uniform over space
  - Simplest test is that the cross-functions are identical to regular *G(r)* and *K(r)* functions for cases/controls i.e. no location where cases occur more often that expected
  - Can estimate and map rate of cases over study area
    1. Work out intensity of cases over area
    2. Work out intensity of cases and controls over area
    3. Divide 1) by 2) to get proportion of cases
    4. Use kernel smoothing to get continuous estimate of intensity
       (a) Replace each point in the point pattern with a 'kernel'
           * Simple localized 2D function centered on the point
       (b) Add up all the kernels to get smooth intensity
           * *spatstat::density()* will produce kernel density estimates
           * *spatstat::split()* will split categorical data into separate point patterns that can be used to calculate and visualize relative proportions (density ratios)
           * *spatialkernel::spseg(, opt = 3)* will calculate bandwith for kernel function (use CV to optimize kernel smoothing)
             - Use to compute ratio and spatial estimate of the rate of cases
             - *opt = 3* indicates want to calculate p-values
    5. Use Monte-Carlo simulation to calculate uncertainty in estimate
       (a) If there are 200 cases out of 1000 total events, produce 99 simulations where the 200 cases are randomly selected from any of the 1000 events using the kernel function to weigh likelihoods
       (b) Compare density ratio from data to simulated density ratios
       (c) Can plot density ratio estimates with significant areas highlighted based on comparison to simulated densities
           * *spatialkernel::plotmc()*
- Space-time
  - Time point process also has window, but instead has min and max time points, not a polygon
  - Usually both spatial and temporal clustering
  - Sometimes clusters in both dimensions
    * Spatial and temporal processes not independent
  - Can use space-time K function
    * Expected number of points within distance d and time t
      - Creates cylindrical volume of space-time
    * If spatial and temporal processes independent

· $K_{st}(s,t) = K_s(s) \times K_t(t)$
· Test independence by checking how two sides of equation differ
· $D(s,t) = K_{st}(s,t) - K_s(s) \times K_t(t)$
· $D(s,t) \approx 0$
* Test significance using Monte-Carlo simulation for permutation test
· Randomly permute event times
· Retains spatial distribution, just breaks links between space and time

Figure 7: Space-Time Monte-Carlo simulation method



· Compare observed $D(s,t)$ vs distribution of simulated values
* *splancs::stmctest()* used, but requires matrix rather than df

- Often don't have point data, e.g. due to confidentiality, instead have areas with subregions summarized
  - *spdep::spplot(spdf, "var")* good way to quickly create choropleth
- Simple null hypothesis is that regions are spatially random
  - Alternative is that there is local spatial structure i.e. neighbours might be more similar than expected under random hypothesis
    * **Spatial autocorrelation**
  - Define neighbours:
    * Adjacent: 2 regions that share a boundary
    * Any two regions within a distance
    * *spdep::poly2nb()* calculates the neighbours of polygons and creates an *nb* object
  - Moran I statistic tests local similarity

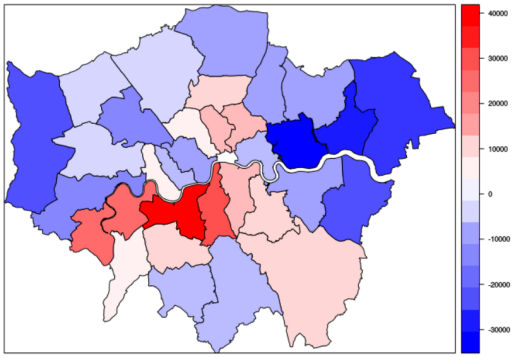
$$I = \frac{n}{\sum_i \sum_j w_{ij}} \frac{\sum_i \sum_j w_{ij}(z_i - \bar{z})(z_j - \bar{z})}{\sum_i (z_i - \bar{z})^2} \tag{2}$$

    * Sum of contributions from two neighbouring pairs
    * If two neighbours both above or below mean, then positive amount
      · Neighbours are similar
    * If two neighbours either side of mean, then contributes a negative amount
      · Neighbours are dissimilar
    * Use MC simulation by randomly arranging numbers over region and compare observed I statistic to distribution of I statistics
    * *spdep::moran.test(var, wnb)*
      · Needs a weighted *nb* object
      · *spdep::nb2listw()* creates weighted *nb* object
- Use *Standardized Mortality Ratio* to observe difference in health data between regions
  - $SMR_i = \frac{O_i}{E_i}$
  - Create *Exceedance Probability Map*
    * Triggers alert when SMR in region is above threshold with certain probability (using binomial distribution to calculate CIs and distribution around SMR)

```r
# Probability of a binomial exceeding a multiple (e) of expected deaths
binom.exceed <- function(observed, population, expected, e){
    1 - pbinom(e * expected, population, prob = observed / population)
}
```

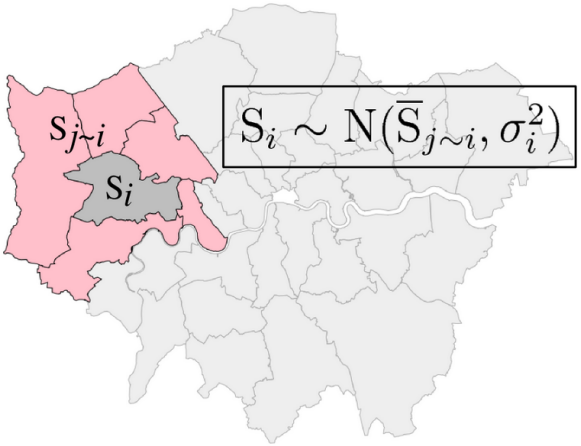
- Generalized Linear Models can be fit to spatial data
  - $Y \sim N(X\beta, \sigma^2)$ standard linear model formula
  - $Y \sim D(\mu(X\beta))$ equation for generalized linear models
    - D is the distribution chosen
    - μ is the *link function* that transforms the $X\beta$ term appropriately for the distribution
    - e.g. $Y \sim \text{Poisson}(\exp(X\beta))$ where $D = \text{Poisson}$, and $\mu = \exp()$
  - Should visualize residuals vs map to check if there is spatially smooth correlation

Figure 8: Map residuals to check for spatially smooth correlation



  - *spdep::moran.mc(res, wnb)* can use Monte-Carlo simulation to test if spatial correlation of residuals using a weight *nb* object
  - Dealing with spatially correlated residuals
    1. See if variable not included in model can be used to explain correlation e.g. age-structure of population, or pollution intensity, by refitting model with added variables
    2. Add spatial feature to model to create a **conditional auto-regression (CAR)** model
       - $Y = X\beta + S(x, y)$
       - $S(x, y)$ could be a continuous function over space, but can be easier to define for each unit
         - $S_i \sim N(\bar{S}_{j\sim i}, \sigma_i^2)$ is one example where for a region i, S comes from normal distribution with mean equal to mean of the neighbouring regions, with variance to be fitted

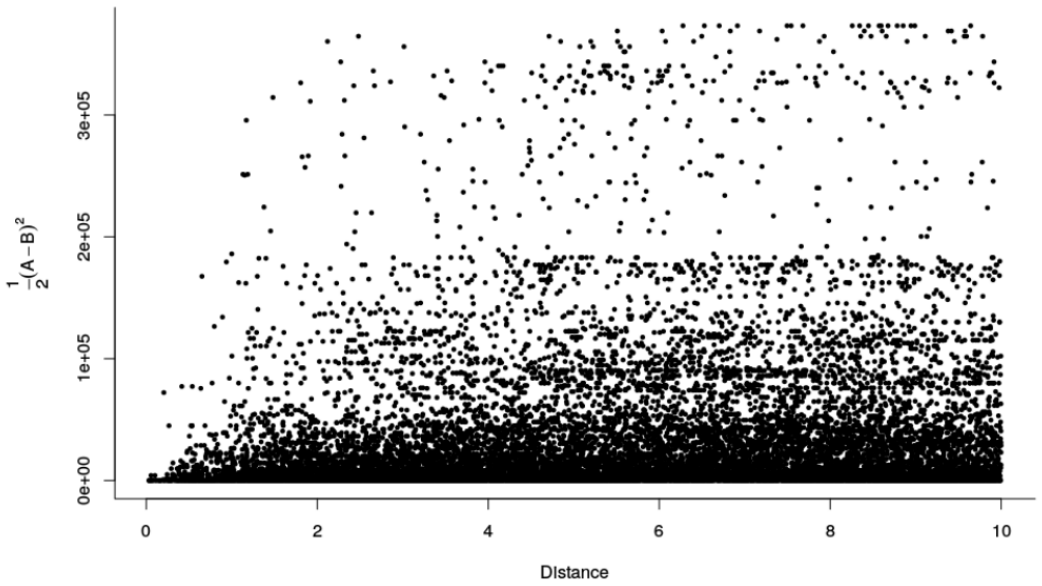
Figure 9: BYM spatial correlation linear model component



        * Hard to maximise likelihood, so instead used Bayesian inference
            · Don't get mean and variance of normal distribution of parameter
            · Instead get samples that give posterior distribution of parameter - the probability distribution of a parameter given the data
            · Use samples to compute credible intervals of posterior distribution
            · *R2BayesX::bayesx(resp vars + sx(i, bs = "spatial", map = gra))* used to add spatial component
            · *gra* object required for map (not *nb*) so use *R2BayesX::nb2gra(nb)* to convert
            · *residuals(mod_fit)[, "mu"]* will add residuals to *spdf* based on rates, not counts, so can be compared across regions with different populations

- Measurement type will vary geostatistical method used e.g. continuous vs count
- Initial geostatistical approaches
    1. Map variables to see large-scale trend
        – Fit trends explicitly as function of coordinates
    2. Show strips in one direction (anisotropy - directionality)
        – Consider anisotropic effects
    3. Discontinuities where values change rapidly
        – Model need to add barriers when calculating distance
- Need to describe correlations between points by distance
    – Variogram cloud
        * Plots half the squared difference between points vs distances
            · Known as the **semi-variance**
            · Half because difference between A-B is the same as B-A

Figure 10: Variogram cloud



        * *gstat::variogram(resp ∼ coords, cloud = TRUE)*
    – Variogram (incorrectly also called "semi-variogram")
        * Bin the variogram cloud distance
        * Average the values in each bin
            · Enough values to provide a good estimate of the mean
        * Plot mean values vs center of the distance bins

    * If semivariance flattens off, semivariance = variance of the data
    * Fit model to variogram
        · Current preferred model is the **matern** class of model due to its differentiability properties

$$C(d) = \sigma^2 \frac{2^{(1-\nu)}}{\Gamma(\nu)} \left(\sqrt{2\nu}\frac{d}{\rho}\right)^{\nu} K_{\nu}\left(\sqrt{2\nu}\frac{d}{\rho}\right) \tag{3}$$

        · *gstat::fit.variogram(vario, model = vgm(parms))*
        · Have to estimate parameters (nugget, partial sill, range)

- Kriging uses model fit from variogram and interpolates data points
  - Simplest when response is continuous variable that is assumed to be Guassian (normally distributed)
  - Prediction variance smallest when closest to data points
    * If nugget = 0, then estimate will go through the data point
  - *gstat::krige(form, val_df, miss_df, model = vario)* will predict the missing values
- Prediction maps
  - Will use kriging, but first need to create a *SpatialPixels* object

```
# ca_geo, geo_bounds have been pre-defined
ls.str()

# Plot the polygon and points
plot(geo_bounds); points(ca_geo)

# Find the corners of the boundary
bbox(geo_bounds)

# Define a 2.5km square grid over the polygon extent. The first parameter is
# the bottom left corner, the second is the width and height of each
# rectangle in the grid. The third specifies the number of rectangles in
# each direction.
grid <- GridTopology(c(537853, 5536290), c(2500, 2500), c(72, 48))

# Create points with the same coordinate system as the boundary
gridpoints <- SpatialPoints(grid, proj4string = CRS(projection(geo_bounds)))
plot(gridpoints)

# Crop out the points outside the boundary
cropped_gridpoints <- crop(gridpoints, geo_bounds)
plot(cropped_gridpoints)

# Convert to SpatialPixels (sp object equivalent)
spgrid <- SpatialPixels(cropped_gridpoints)
coordnames(spgrid) <- c("x", "y")
plot(spgrid)
```

  - Pass *SpatialPixels* grid as the new data to the *gstat::krige()* function
  - *automap::autoKrige(form, input_data, new_data, model)* will quickly fit variogram and use kriging
    * Pass to *plot()* function to plot kriging predictions, standard error, and variogram
    * Use a *SpatialPixels* grid to produce a surface rather than individual points
    * Need to carefully check results to make sure the results make sense e.g. has small nugget and rises to reach sill
    * Only possible when meets assumption of Guassian data
- Working with non-Guassian data in kriging
  - Trans-Guassian kriging transforms sample values to approximate Guassian distribution

  - Model-based geostatistics
    * Use bayesian statistics for inference e.g. Markov-Chain Monte-Carlo samplers to estimate paramter distributions
    * Computationally intensive
  - Nested variograms
    * Spatial correlation may not be well-fit to single variogram function
      · Particularly if spatial structure on more than one level
    * Fit variograms to different components and create combined function
    * May be possible to overfit and precise fit may not affect final model output much

9 Datacamp: Interactive maps with leaflet with R

- Leaflet uses 'tiles' to create layers of maps e.g. basemap, markers etc
  - Specify different tile using *leaflet::addProviderTiles()*
- Use *ggmap::geocode(location = "address")* to return lon-lat for address
- Set default view using two different methods

Figure 11: Set Leaflet default view

```
leaflet() %>%
 addTiles() %>%
 setView(lng = -73.98575,
         lat = 40.74856,
         zoom = 13)
```


```
leaflet() %>%
 addTiles() %>%
 fitBounds(
  lng1 = -73.910, lat1 = 40.773,
  lng2 = -74.060, lat2 = 40.723)
```

  - *leaflet::clearBounds()* removes previously saved default views
- Can restrict panning etc using *leaflet::leaflet(options = leaflet_options(minZoom = x, dragging = FALSE))*
- *leaflet::addMarkers(lng = df$lon, lat = df$lat, popup = df$desc)* can add markers from a df with text descriptions that show when clicked
  - *leaflet::addPopups()* alternative that don't show markers
  - *leaflet::clearMarkers()* removes previously added markers
    * Useful if saved basemap object and want to replace markers for specific map
    * *leaflet::addCircleMarkers()* can be a useful alternative when pin markers overlap - similar to scatter plot
- If you pipe the data to the map, no need to specify dataframe for every option e.g. longitudes etc.
  - Need to use the ~ symbol though e.g. *leaflet::addCircleMarkers(label = ~paste0("", name, ""))*

    - Specifying *label* instead of *popup* means the information is available on hover, rather than click

- To add color to markers, polygons etc:
    1. Specify a colour palette (depends on variable type) e.g.:
        - *pal <- colorFactor(palette = c("red", "blue", "#9b4a11"), levels = c("Public", "Private", "For-Profit"))*
        - *pal <- colorNumeric(palette = "reds", domain = c(1:50), reverse = TRUE)*
        - *pal <- colorBin(palette = "reds", bins, domain = c(1:50))*
        - *pal <- colorQuantile(palette = "reds", n, domain = c(1:50))*
    2. *leaflet::addCircleMarkers(col = ~pal(var))*
    3. Add legend
        - *leaflet::addLegend(pal = pal, values = c("Public", "Private", "For-Profit"), opacity, title, position)*

- Can make map more easily searchable with *leaflet.extras* package
    - Can geocode directly in the map without needing *ggmap::geocode()*
    - *addSearchOSM()* allows users to search for locations
    - *addReverseSearchOSM()* shows lon and lat when picked dropped
    - *addResetMapButton()* provides an icon to revert to default view
    - *addSearchFeatures(targetGroups, options = searchFeaturesOptions(zoom))* allows users to search specific groups of markers and zoom in once identified

- Can add markers in layers to allow groups to be toggled on and off
    - Have to create subsets of the dataframe before adding separately
        - Because adding in layers, order matters to stacking

```
# Load the htmltools package to clean up characters that may be
# interpreted as HTML
library(htmltools)

# Create data frame called public with only public colleges
public <- filter(ipeds, sector_label == ''Public'')

# Create a leaflet map of public colleges called m3
m3 <- leaflet() %>%
        addProviderTiles(''CartoDB'') %>%
        addCircleMarkers(
            data = public, radius = 2, label = ~htmlEscape(name),
            color = ~pal(sector_label), group = ''Public''
            ) %>%
        addLayersControl(overlayGroups = c(''Public''))
```

- Similarly, can add multiple basemap layers
    - Only one can be viewed at a time

```
leaflet() %>%
    addTiles(group = ''OSM'') %>%
    addProviderTiles(''CartoDB'', group = ''Carto'') %>%
    addProviderTiles(''Esri'') %>%
    addLayersControl(baseGroups = c(''OSM'', ''Carto'', ''Esri''))
```

    - Last basemap layer will be the one that is shown, unless chosen otherwise using toggle

- Can cluster observations together
    - *addCircleMarkers(clusterOptions = markerClusterOptions())*

- Can create choropleths
    - *leaflet::addPolygons(weight, color, label, highlight = highlightOptions)*
    - *pal <- colorBin(palette = "reds", bins, domain = sp@data$var)*

- *htmlwidgets::saveWidget()*