Lab 2.1: Data Wrangling

Spatio-temporal modeling and prediction generally involve substantial amounts of data that are available to the user in a variety of forms, but more often than not as tables in CSV files or text files. A considerable amount of time is usually spent in loading the data and pre-processing them in order to put them into a form that is suitable for analysis. Fortunately, there are several packages in R that help the user achieve these goals quickly; here we focus on the packages **dplyr** and **tidyr**, which contain functions particularly suited for the data manipulation techniques that are required. We first load the required packages, as well as **STRbook** (visit https://spacetimewithr.org for instructions on how to install **STRbook**).

```
library("dplyr")
library("tidyr")
library("STRbook")
```

As running example, we shall consider a data set from the National Oceanic and Atmospheric Administration (NOAA) that was provided to us as text in data tables available with the package **STRbook**. There are six data tables:

- Stationinfo.dat. This table contains 328 rows (one for each station) and three columns (station ID, latitude coordinate, and longitude coordinate) containing information on the stations' locations.
- Times_1990.dat. This table contains 1461 rows (one for each day between 01 January 1990 and 30 December 1993) and four columns (Julian date, year, month, day) containing the data time stamps.
- Tmax_1990.dat. This table contains 1461 rows (one for each time point) and 328 columns (one for each station location) containing all maximum temperature data with missing values coded as -9999.
- Tmin_1990.dat. Same as Tmax_1990.dat but containing minimum temperature data.
- TDP_1990.dat. Same as Tmax_1990.dat but containing temperature dew point data with missing values coded as -999.90001.
- Precip_1990.dat. Same as Tmax_1990.dat but containing precipitation data with missing values coded as -99.989998.

The first task is to reconcile all these data into one object. Before seeing how to use the spatio-temporal data classes to do this, we first consider the rather simpler task of reconciling them into a standard R data frame in long format.

Working with Spatio-Temporal Data in Long Format

The station locations, time stamps and maximum temperature data can be loaded into R from **STRbook** as follows.

```
locs <- read.table(system.file("extdata", "Stationinfo.dat",
                               package = "STRbook"),
                  col.names = c("id", "lat", "lon"))
Times <- read.table(system.file("extdata", "Times_1990.dat",
                                package = "STRbook"),
                  col.names = c("julian", "year", "month", "day"))
Tmax <- read.table(system.file("extdata", "Tmax_1990.dat",
                               package = "STRbook"))
```

In this case, `system.file` and its arguments are used to locate the data within the package **STRbook**, while `read.table` is the most important function used in R for reading data input from text files. By default, `read.table` assumes that data items are separated by a blank space, but this can be changed using the argument `sep`. Other important data input functions worth looking up include `read.csv` for comma-separated value files, and `read.delim`.

Above we have added the column names to the data `locs` and `Times` since these were not available with the original text tables. Since we did not assign column names to `Tmax`, the column names are the default ones assigned by `read.table`, that is, `V1`, `V2, ..., V328`. As these do not relate to the station ID in any way, we rename these columns as appropriate using the data in `locs`.

```
names(Tmax) <- locs$id
```

The other data can be loaded in a similar way to `Tmax`; we denote the resulting variables as `Tmin`, `TDP`, and `Precip`, respectively. One can, and should, use the functions `head` and `tail` to check that the loaded data are sensible.

Consider now the maximum-temperature data in the NOAA data set. Since each row in `Tmax` is associated with a time point, we can attach it columnwise to the data frame `Times` using `cbind`.

```
Tmax <- cbind(Times, Tmax)
head(names(Tmax), 10)

##  [1] "julian" "year"   "month"  "day"    "3804"   "3809"
##  [7] "3810"   "3811"   "3812"   "3813"
```

Now `Tmax` contains the time information in the first four columns and temperature data in the other columns. To put `Tmax` into long format we need to identify a *key–value* pair. In our case, the data are in space-wide format where the *keys* are the station IDs and the *values* are the maximum temperatures (which we store in a field named `z`). The function we use to put the data frame into long format is `gather`. This function takes the data as first argument, the key–value pair, and then the next arguments are the names of any columns to exclude as values (in this case those relating to the time stamp).

```
Tmax_long <- gather(Tmax, id, z, -julian, -year, -month, -day)
head(Tmax_long)

##   julian year month day   id  z
## 1 726834 1990     1   1 3804 35
## 2 726835 1990     1   2 3804 42
## 3 726836 1990     1   3 3804 49
## 4 726837 1990     1   4 3804 59
## 5 726838 1990     1   5 3804 41
## 6 726839 1990     1   6 3804 45
```

Note how `gather` has helped us achieve our goal: we now have a single row per measurement and multiple rows may be associated with the same time point. As is, the column `id` is of class `character` since it was extracted from the column names. Since the station ID is an integer it is more natural to ensure the field is of class `integer`.

```
Tmax_long$id <- as.integer(Tmax_long$id)
```

There is little use to keep missing data (coded as -9999 in our case) when the data are in long format. To filter out these data we can use the function `filter`. Frequently it is better to use an *inequality* criterion (e.g., less than) when filtering in this way rather than an *equivalence* criterion (is equal to) due to truncation error when storing data. This is what we do below, and filter out data with values less than -9998 rather than data with values equal to -9999. This is particularly important when processing the other variables, such as preciptation, where the missing value is -99.989998.

```
nrow(Tmax_long)

## [1] 479208

Tmax_long <- filter(Tmax_long, !(z <= -9998))
nrow(Tmax_long)

## [1] 196253
```

Note how the number of rows in our data set (returned from the function `nrow`) has now decreased by more than half. One may also use the `R` function `subset`; however, `filter` tends to be faster for large data sets. Both `subset` and `filter` take a logical expression as instruction on how to filter out unwanted rows. As with `gather`, the column names in the logical expression do not appear as strings. In R this method of providing arguments is known as *non-standard evaluation*, and we shall see several instances of it in the course of the Labs.

Now assume we wish to include minimum temperature and the other variables inside this data frame too. The first thing we need to do is first make sure every measurement z is attributed to a process. In our case, we need to add a column, say proc, indicating what process the measurement relates to. There are a few ways in which to add a column to a data frame; here we shall introduce the function **mutate**, which will facilitate operations in the following Labs.

```
Tmax_long <- mutate(Tmax_long, proc = "Tmax")
head(Tmax_long)

##   julian year month day   id  z proc
## 1 726834 1990     1   1 3804 35 Tmax
## 2 726835 1990     1   2 3804 42 Tmax
## 3 726836 1990     1   3 3804 49 Tmax
## 4 726837 1990     1   4 3804 59 Tmax
## 5 726838 1990     1   5 3804 41 Tmax
## 6 726839 1990     1   6 3804 45 Tmax
```

Now repeat the same procedure with the other variables to obtain data frames Tmin_long, TDP_long, and Precip_long (remember the different codings for the missing values!). To save time, the resulting data frames can also be loaded directly from **STRbook** as follows.

```
data(Tmin_long, package = "STRbook")
data(TDP_long, package = "STRbook")
data(Precip_long, package = "STRbook")
```

We can now construct our final data frame in long format by simply concatenating all these (rowwise) together using the function **rbind**.

```
NOAA_df_1990 <- rbind(Tmax_long, Tmin_long, TDP_long, Precip_long)
```

There are many advantages of having data in long form. For example, it makes grouping and summarizing particularly easy. Let us say we want to find the mean value for each variable in each year. We do this using the functions **group_by** and **summarise**. The function **group_by** creates a *grouped data frame*, while **summarise** does an operation *on each group within the grouped data frame*.

```
summ <- group_by(NOAA_df_1990, year, proc) %>%  # groupings
        summarise(mean_proc = mean(z))          # operation
```

Alternatively, we may wish to find out the number of days on which it did not rain at each station in June of every year. We can first filter out the other variables and then use **summarise**.

```
NOAA_precip <- filter(NOAA_df_1990, proc == "Precip" & month == 6)
summ <- group_by(NOAA_precip, year, id) %>%
        summarise(days_no_precip = sum(z == 0))
head(summ)

## # A tibble: 6 x 3
## # Groups:   year [1]
##     year    id days_no_precip
##    <int> <int>          <int>
## 1   1990  3804             19
## 2   1990  3810             26
## 3   1990  3811             21
## 4   1990  3812             24
## 5   1990  3813             25
## 6   1990  3816             23
```

The median number of days with no recorded precipitation was

```
median(summ$days_no_precip)

## [1] 20
```

In the R code above, we have used the operator %>%, known as the *pipe* operator. This operator has its own nuances and should be used with care, but we find it provides a clear desciption of the processing pipeline a data set is passed through. We shall always use this operator as `x %>% f(y)`, which is shorthand for `f(x,y)`. For example, the June summaries above can be found equivalently using the commands

```
grps <- group_by(NOAA_precip, year, id)
summ <- summarise(grps, days_no_precip = sum(z == 0))
```

There are other useful commands in **dplyr** that we use in other Labs. First, the function **arrange** sorts by a column. For example, `NOAA_df_1990` is sorted first by station ID, and then by time (Julian date). The following code sorts the data first by time and then by station ID.

```
NOAA_df_sorted <- arrange(NOAA_df_1990, julian, id)
```

Calling `head(NOAA_df_sorted)` reveals that no measurements on temperature dew point are available for the first few days of the data set.

Another useful function is **select**, which can be used to select or discard columns. For example, in the following, df1 selects only the Julian date and the measurement while df2 contains all columns except the Julian date.

```
df1 <- select(NOAA_df_1990, julian, z)
df2 <- select(NOAA_df_1990, -julian)
```

At present, our long data frame contains no spatial information attached to it. However, for each station ID we have an associated coordinate in the data frame locs. We can merge locs to NOAA_df_1990 using the function **left_join**; this is considerably faster than the function **merge**. With **left_join** we need to supply the column field name by which we are merging. In our case, the field common to both data sets is "id".

```
NOAA_df_1990 <- left_join(NOAA_df_1990, locs, by = "id")
```

Finally, it may be the case that one wishes to revert from long format to either space-wide or time-wide format. The reverse function of **gather** is **spread**. This also works by identifying the key–value pair in the data frame; the values are then "widened" into a table while the keys are used to label the columns. For example, the code below constructs a space-wide data frame of maximum temperatures, with each row denoting a different date and each column containing data z from a specific station id.

```
Tmax_long_sel <- select(Tmax_long, julian, id, z)
Tmax_wide <- spread(Tmax_long_sel, id, z)
dim(Tmax_wide)

## [1] 1461  138
```

The first column is the Julian date. Should one wish to construct a standard matrix containing these data, then one can simply drop this column and convert as follows.

```
M <- select(Tmax_wide, -julian) %>% as.matrix()
```

Working with Spatio-Temporal Data Classes

Next, we convert the data into objects of class STIDF and STFDF; in these class names "DF" is short for "data frame," which indicates that in addition to the spatio-temporal locations (which only need STI or STF objects), the objects will also contain data. These classes are defined in the package **spacetime**. Since sometimes we construct spatio-temporal objects using spatial objects we also need to load the package **sp**. For details on these classes see **?**.

```
library("sp")
library("spacetime")

## Registered S3 method overwritten by 'xts':
##   method     from
##   as.zoo.xts zoo
```

Constructing an STIDF Object

The spatio-temporal object for irregular data, STIDF, can be constructed using two functions: **stConstruct** and **STIDF**. Let us focus on the maximum temperature in Tmax_long. The only thing we need to do before we call **stConstruct** is to define a formal time stamp from the year,month,day fields. First, we construct a field with the date in year–month–day format using the function **paste**, which concatenates strings together. Instead of typing NOAA_df_1990$year, NOAA_df_1990$month and NOAA_df_1990$day we embed the **paste** function within the function **with** to reduce code length.

```
NOAA_df_1990$date <- with(NOAA_df_1990,
                          paste(year, month, day, sep = "-"))
head(NOAA_df_1990$date, 4)    # show first four elements

## [1] "1990-1-1" "1990-1-2" "1990-1-3" "1990-1-4"
```

The field date is of type character. This field can now be converted into a Date object using **as.Date**.

```
NOAA_df_1990$date <- as.Date(NOAA_df_1990$date)
class(NOAA_df_1990$date)

## [1] "Date"
```

Now we have everything in place to construct the spatio-temporal object of class STIDF for maximum temperature. The easiest way to do this is using **stConstruct**, in which we provide the data frame in long format and indicate which are the spatial and temporal coordinates. This is the bare minimum required for constructing a spatio-temporal data set.

```
Tmax_long2 <- filter(NOAA_df_1990, proc == "Tmax")
STObj <- stConstruct(x = Tmax_long2,             # data set
                     space = c("lon", "lat"),   # spatial fields
                     time = "date")             # time field
class(STObj)

## [1] "STIDF"
## attr(,"package")
## [1] "spacetime"
```

The function class can be used to confirm we have successfully generated an object of class STIDF. There are several other options that can be used with stConstruct. For example, one can set the coordinate reference system or specify whether the time field indicates an instance or an interval. Type help(stConstruct) into the R console for more details.

The function STIDF is slightly different from stConstruct as it requires one to also specify the spatial part as an object of class Spatial from the package **sp**. In our case, the spatial component is simply an object containing irregularly spaced data, which in the package **sp** is a SpatialPoints object. A SpatialPoints object may be constructed using the function SpatialPoints and by supplying the coordinates as arguments. As with stConstruct, several other arguments can also be supplied to SpatialPoints; see the help file of SpatialPoints for more details.

```
spat_part <- SpatialPoints(coords = Tmax_long2[, c("lon", "lat")])
temp_part <- Tmax_long2$date
STObj2 <- STIDF(sp = spat_part,
                time = temp_part,
                data = select(Tmax_long2, -date, -lon, -lat))
class(STObj2)

## [1] "STIDF"
## attr(,"package")
## [1] "spacetime"
```

Constructing an STFDF Object

A similar approach can be used to construct an STFDF object instead of an STIDF object. When the spatial points are fixed in time, we only need to provide as many spatial coordinates as there are spatial points, in this case those of the station locations. We also need to provide the regular time stamps, that is, one for each day between 01 January 1990 and 30 December 1993. Finally, the data can be provided both in space-wide or time-wide format with stConstruct, and in long format with STFDF. Here we show how to use STFDF.

The spatial and temporal parts can be obtained from the original data as follows.

```
spat_part <- SpatialPoints(coords = locs[, c("lon", "lat")])
temp_part <- with(Times,
                  paste(year, month, day, sep = "-"))
temp_part <- as.Date(temp_part)
```

The data need to be provided in long format, but now they must contain all the missing values too since a data point must be provided for every spatial and temporal combination. To get the data into long format we use gather.

```
Tmax_long3 <- gather(Tmax, id, z, -julian, -year, -month, -day)
```

It is very important that the data frame in long format supplied to `STFDF` has the spatial index moving faster than the temporal index, and that the order of the spatial index is the same as that of the spatial component supplied.

```
Tmax_long3$id <- as.integer(Tmax_long3$id)
Tmax_long3 <- arrange(Tmax_long3,julian,id)
```

Confirming that the spatial ordering in `Tmax_long3` is the correct one can be done as follows.

```
all(unique(Tmax_long3$id) == locs$id)

## [1] TRUE
```

We are now ready to construct the `STFDF`.

```
STObj3 <- STFDF(sp = spat_part,
                time = temp_part,
                data = Tmax_long3)
class(STObj3)

## [1] "STFDF"
## attr(,"package")
## [1] "spacetime"
```

Since we will be using `STObj3` often in the Labs we further equip it with a coordinate reference system (see **?**, for details on these reference systems),

```
proj4string(STObj3) <- CRS("+proj=longlat +ellps=WGS84")
```

and replace the missing values (currently coded as -9999) with `NA`s.

```
STObj3$z[STObj3$z == -9999] <- NA
```

For ease of access, this object is saved as a data file in **STRbook** and can be loaded using the command `data("STObj3", package = "STRbook")`.