SSER Workshop on Using Large-scale Survey Data

Notes on R Programming Language

December 16-23, 2016

SSER

SOCIETY
FOR
SOCIAL AND ECONOMIC
RESEARCH

Introduction

Why should you use programming for statistical analysis? Why use R?

- Programming is crucial for reproducibility and verifiability of your research
- R is open-source and free of cost
  - Will always remain available
  - Development is fast; tools for using new, cutting-edge statistical techniques become available as add-on packages.

Importing and Exporting Data

How to read data in R

Reading fixed-width text files

- NSS/ASI data are given as fixed-width, text files.
- First look at the survey schedule and instructions, and understand which variables are of interest to you. The survey schedule is divided into blocks.
- Look at the layout file to see which data files (levels) have the data you are interested in.
- The layout file gives you byte position of each variable in each file.
- In R, read.fwf can be used to read fixed-width files.

Reading other types of files.

- You often need to read spreadsheet files into R. Excel is notoriously messy. Use your spreadsheet application to save such files as csv (comma separated value files), using a proper delimiter (often better to use | instead of a comma), and import into R using read.table command.
- You can also use readxl package to directly read excel files into R
- You can use read.sta to read stata files, read.spss to use spss files directly into R.

Box 1 Reading other types of files

The following command reads a file called R6810L01.TXT into R and saves it as a data frame (called newobj here). Note that everything in R is case sensitive. So use upper and lower cases correctly.

```r
read.fwf("R6810L01.TXT",
           c(3,5,2,3,1,1,3,2,2,2,1,1,1,4,1,1,2,2,5,2,1,1,1,
             6,6,3,3,1,1,1,1,2,55,3,3,10,3,3,10),
           col.names = c("roundcode","fsunumber","round","schedule_number",
                         "sample", "sector","state_region","district",
                         "stratum_number", "sub_stratum","filler1",
                         "sub_round","sub_sample","fod_sub_region",
                         "hamlet_group","second_stage_stratum","hhs_no",
                         "level","filler2","informant_sno","response_code",
                         "survey_code","subsitution_code","survey_date",
                         "despatch_date","canvas_time","canvas_time_block8",
                         "remarks1_block9","remarks2_block9",
                         "remarks1_elsewhere","remarks2_elsewhere",
                         "special_characters","blank","NSS","NSC","MLT",
                         "nss_sr","nsc_sr","mlt_sr"))->level1
```

- The command has three arguments: name of the file to be read, the width of each column in the file, and variable name to be given to each column.
- The width of each column is given in the layout file for NSS/ASI data
- There are no strict naming conventions for variable names. But give variable names sensibly. Do not use spaces and other special characters in names. Variable names are also case sensitive.

Box 2 Reading other types of files

The following command reads a file called filename.csv into R and saves it as a data frame (called newobj here)

```r
read.table("filename.csv",sep="|",header=T)->newobj
read.sta("filename.sta")->newobj ; reads stata files
read.spss("filename.spss")->newobj ; reads spss files
readxl::read_excel("filename.xlsx")->newobj ; reads excel files
readRDS("filename.rds")->newobj ; reads rds, an R data format
```

- argument sep tells the command which character should be used as a delimiter.
- argument “header=T” tells the command that the first line of the file has the variable names.

How to export data from R

Box 3 Exporting data to csv and rds files

The following command converts a data frame into different types of files.

```
write.table(data1,file="datafile.csv",sep="|",row.names=F)
; rds files are R data files, good for sharing with other R uses
saveRDS(data1,file="datafile.rds")
library(openxls)
```

Box 4 Exporting data to excel files

Library openxlsx can be used to export dataframes to excel files.

```
createWorkbook()->wb
addWorksheet(wb,"soybean_entity")
addWorksheet(wb,"soybean_aggregates")
addWorksheet(wb,"soybean_results")
writeData(wb,"soybean_entity",x=fbs1,colNames=TRUE,rowNames = FALSE)
writeData(wb,"soybean_aggregates",x=fbs2,colNames=TRUE,rowNames = FALSE)
writeData(wb,"soybean_results",x=fbs3,colNames=TRUE,rowNames=FALSE)
saveWorkbook(wb,file="soybean_trade.xlsx",overwrite=TRUE)
```

Working with Dataframes

Dataframes and vectors

- Dataframes are the main objects in which you store data.
- Dataframes are data tables with variables in columns and observations in rows.
- Each column/variable is of a particular type. There are three main types: numeric, character and factor.
  - character variables are text strings,
  - numeric variables are numbers
  - factors are categorical variables, in which each observation belongs to a category. Category labels may be numeric or characters. Even when they are numeric, they are just labels for categories and should not be confused as numbers.

Box 5 Changing variable type

Unless you specify while importing data into R, R guesses the type of vector for each variable. Sometimes the guessed data type may not be what you want, and you may have to change it. Typical use cases are as follows.

```
1  class(level1$sector)
2  as.character(level1$sector)->level1$sector
3  as.numeric(level1$sector)->level1$sector
4  as.numeric(as.character(level1$sector))->level1$sector
```

- Line 1 tells you vector type of variable sector in level1
- Line 2 converts variable sector in level1 to character and overwrites the original vector
- Line 3 converts character variable sector in level1 to numeric and overwrites the original vector
- Line 4 converts a factor with numeric labels into character, and then converts it into a numeric vector.

Box 6 Logical operators in R

```
ifelse(level3$age<18,"Child","Adult")->level3$agegroup1
ifelse(level3$age<18,"Child",
       ifelse(level3$age<60,"Working-age adult",
              "Old person"))->level3$agegroup2
```

Operator	Meaning	Example
==	equal to	level3$age==10
!=	equal to	level3$age!=10
>, >=	greater than/equal	level3$age>=60
<, <=	less than/equal	level3$age<18
is.na	is blank	is.na(level3$age)
!is.na	is not blank	!is.na(level3$age)
%in%	is part of the sequence	level3$state %in% c("Haryana","Punjab")
|	OR	level3$age<18 \| level3$age>=60
&	AND	level3$age<18 & level3$sex==1

Merging data

Different items of data collected from a sample unit (household or firm, for example) are disaggregated at different levels.

- household-level information,
- information on individuals,
- information on each occupation of an individual,
- information on each plot of land,

- information on each item of consumption of a household,
- information on each input used by a firm

Such data are provided in separate levels/files.

When you need to put together information provided in different data tables, you would need to merge data frames that have those tables.

There are three types of merges/joins

- inner: matching rows from two data frames are merged; non-matching rows are dropped.
- one-sided: matching rows from two data frames are merged; non-matching rows from one data frame are included; non-matching rows from the other data frame are dropped.
- outer: all rows are included; where possible, rows are matched; where matching information is not found, blank cells are created.

Box 7 Merging dataframes

```
; inner join
merge(level2,level3,
    by=c("fsunumber","hamlet_group","second_stage_stratum","hhs_no"))->emp68

; left join
merge(level2,level3,
      by=c("fsunumber","hamlet_group","second_stage_stratum","hhs_no"),
      all.x=T)->emp68

; right join
merge(level2,level3,
      by=c("fsunumber","hamlet_group","second_stage_stratum","hhs_no"),
      all.y=T)->emp68

; outer join
merge(level2,level3,
      by=c("fsunumber","hamlet_group","second_stage_stratum","hhs_no"),
      all.x=T)->emp68
```

Summarising data

Aggregate

- Description

  Splits the data into subsets, computes summary statistics for each, and returns the result in a convenient form.

- Usage

  aggregate(x, …)

  ## Default S3 method:

  `aggregate(x, ...)`

  ## S3 method for class 'data.frame'

  `aggregate(x, by, FUN, ..., simplify = TRUE, drop = TRUE)`

  ## S3 method for class 'formula'

  `aggregate(formula, data, FUN, ...,subset, na.action = na.omit)`

  ## S3 method for class 'ts'

  ```
  aggregate(x, nfrequency = 1, FUN = sum, ndeltat = 1,
            ts.eps = getOption("ts.eps"), ...)
  ```

Box 8 Using aggregate to summarise data

```
aggregate(level2$hhd_size,
          list(level2$sector,level2$social_group),FUN=mean)

; or the syntax below
aggregate(hhd_size~sector+social_group,level2,FUN=mean)
```

plyr

reshape2

- Flexibly reshape data.
- `melt` turns a wide data frame into a long-form data frame.
  - `id` variables Variables you need to group observations for creating any statistical summaries
  - `measure` or `m` variables Variables of which you want the summaries (income, consumption, days of work, wage, etc)
  - The output is a dataframe which has all the `id` variables, and two extra variables: a variable called "variable", and a variable called "value"
  - The output dataframe has n*m rows where n is the number of rows in the original database, and m is the number of `measure` variables.
  - For each row in original data, m rows are created in the output data, one for each `measure` variable.

  - In each row in the output data, `variable` contains the label of the `measure` variable, and `value` contains its value.

- `dcast` turns the molten database into a desired format, taking whatever statistical summary you want.

Box 9 Reshaping and summarising data using reshape2

```
library(reshape2)
ifelse(level2$NSS==level2$NSC,level2$MLT/100,level2$MLT/200)->level2$weight
level2$hhd_size*level2$weight->level2$weightedsize
melt(level2,id=c("social_group","sector"),m=c("weightedsize","weight"))->a
ifelse(a$sector==1,"Rural","Urban")->a$sector
dcast(a,social_group~sector+variable,sum)->a
# names(a)
# [1] "social_group"  "Rural_weightedsize" "Rural_weight"  "Urban_weightedsize"
# [5] "Urban_weight"
a[,2]/a[,3]->a$Rural_mean
a$Urban_weightedsize/a$Urban_weight->a$Urban_mean
a[,c("social_group","Rural_mean","Urban_mean")]
```

data.table

library(data.table) provides an alternative to data frames. It creates a different kind of object, data tables, that allow you to manipulate data flexibly and efficiently. data.tables can do almost everything you do with aggregate, plyr or reshape2. We are explaining the basic syntax here. See data.table vignettes for details.

Box 10 Creating a data table object

```
fread("filename.csv")->dt ; reads a csv file and creates a datatable called dt
as.data.frame(level2)->level2.dt ; converts a data frame to a data table
```

The basis syntax of data.table is as follows: `DT[i,j,by]`

- subset DT using i, calculate j, group using by
- i (subsetting rows)
  - a set of logical expressions to subset DT
  - columns can be referred to directly by their names
    `level2.dt[sector==1]` ; subsets rural households
- j (selecting columns, summarising)
  - `[,columnname]` selects a column and returns it as a vector

- `[,.columnname]` selects a column and returns the output as datatable
- `[,.(columnname1, columnname2)]` or [,list(columnname1, columnname2)]~ return two columns
- `[,.(colname1=columnname1, colname2=columnname2)]` simultaneously renames columns
- `[,.(col1sum=mean(columnname1),col2sum=med(columnname2))]` calculates mean and median.

Box 11 Computing weighted measures using data.table

```
level2.dt[,.(mean=weighted.mean(hhd_size,weight)),social_group]
```

Statistical graphs with ggplot2

Published by
Society for Social and Economic Research, New Delhi.
E-mail: office@sser.in