UNIVERSITÉ
DE GENÈVE
FACULTÉ DES SCIENCES
ÉCONOMIQUES ET SOCIALES
Département d'économétrie

Mining sequence data in R with the TraMineR package: A user's guide[1]

(for version 1.8)

Alexis Gabadinho, Gilbert Ritschard, Matthias Studer
and Nicolas S. Müller

Department of Econometrics and Laboratory of Demography
University of Geneva, Switzerland

http://mephisto.unige.ch/traminer/

March 18, 2011

[1] This work is part of the research project "Mining event histories: Towards new insights on personal Swiss life courses" supported by the Swiss National Science Foundation under grants FN-100012-113998 and FN-100015-122230.

Acknowledgments: TraMineR was mainly developed on a Ubuntu/Linux system with several open-source free tools and programs, including of course R and the LaTeX language used to write this manual. We would like to thank all the contributors to those free softwares. We also would like to thank Cees Elzinga for providing us the code of his CHESA software for sequence analysis, which was helpful to program some of the metrics he introduced to compute distances between sequences. Thanks also to the participants of the Research Seminar in Statistics for the Social Sciences and Demography in Geneva as well as to the participants of the Workshop on Sequential Data Analysis held in Lund, Sweden, May 8-9 2008, for their useful remarks and for β-testing earlier versions of the package. Thanks also to the Swiss Household Panel who authorized us to use a sample of their data, and to D. McVicar and M. Anyadike-Danes for the permission regarding the *mvad* data set they used in an article of the Journal of the Royal Statistical Society. Those data sets are included in the TraMineR package and are used for illustrating this user's guide.

Reporting bugs: We have indeed carefully tested the package. Nevertheless, we cannot exclude that there remain programming errors and encourage you to report any bugs you may encounter to the package maintainer who is presently `alexis.gabadinho@unige.ch`. You will thus contribute to improve the package.

Referencing TraMineR: Thank you for citing this User's guide, i.e.

> Gabadinho, A., G. Ritschard, M. Studer and N. S. Müller
> Mining sequence data in R with the TraMineR package: A user's guide
> University of Geneva, 2010. (http://mephisto.unige.ch/traminer)

when presenting analyses realized with the help of TraMineR.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

TraMineR is a R-package for mining and visualizing sequences of categorical data. Its primary aim is the knowledge discovery from event or state sequences describing life courses, although most of its features apply also to non temporal data such as text or DNA sequences for instance. The name TraMineR is a contraction of Life Trajectory Miner for R. Indeed, as some may suspect, it was also inspired by the authors' taste for Gewurztraminer wine. This guide is essentially a tutorial that describes the features and usage of the TraMineR package. It may also serve, however, as an introduction to sequential data analysis. The presentation is illustrated with data from the social sciences. Illustrative data sets and R scripts (sequence of R-commands) [1] are included in the TraMineR distribution package.

The functions and options used in the guide as well as their displayed output correspond to the version indicated on the title page. Though the guide discusses the major functionalities of the package, it is not exhaustive. For a full list and description of available functions, see the Reference Manual of the current version that can be found on the CRAN (http://cran.r-project.org/web/packages/TraMineR/). Check also the 'History' tab on the package web page (http://mephisto.unige.ch/traminer) for the latest added features.

For newcomers to R, a short introduction to the R-environment is given in Appendix A in which the reader will learn where R can be obtained as well as its basic commands and principles. Chapter 3 shortly explains how to use the package and describes the illustrative data sets provided with it.

1.1 Aims and features of the **TraMineR** package

Some of the features of TraMineR can be found in other statistical programs handling sequential data. For instance, TDA (Rohwer and Pötter, 2002), which is freely available at http://www.stat.ruhr-uni-bochum.de/tda.html, the T-COFFEE/SALTT program by Notredame et al. (2006), the dedicated CHESA program by Elzinga (2007) freely downloadable at http://home.fsw.vu.nl/ch.elzinga/ and the add-on Stata package by Brzinsky-Fay et al. (2006) freely available for licensed Stata users all compute the optimal-matching edit distance between sequences and each of them offers specific useful facilities for describing sets of sequences. TraMineR is to our knowledge the first such toolbox for the free R statistical and graphical environment. Our objective with TraMineR is to put together most of the features proposed separately by other softwares as well as offering original tools for extracting useful knowledge from sequence data. Its salient characteristics are

- R and TraMineR are free.

[1] R demo scripts named *Rendering*, *Seqdist* and *Events* are in the *demo* directory of the package tree and can be run by means of the `demo()`, for instance `demo("Describing_visualizing",package="TraMineR")` for the first one.

- Since TraMineR is developed in R, it takes advantage of many already optimized procedures of R as well as of its powerful graphics capabilities.
- R runs under several OS including Linux, MacOS X, Unix and Windows. A same R program runs unmodified under all operating systems[2]. The same is indeed true for R-packages and hence for TraMineR.
- TraMineR features a unique set of procedures for analysing and visualizing sequence data, such as
  - handling a large number of state and time stamped event sequence representations, simple functions for transforming to and from different formats;
  - individual sequence summaries and summaries of sequence sets;
  - selecting and displaying the most frequent sequences or subsequences;
  - various metrics for evaluating distances between sequences;
  - aggregated and index plots of sets of sequences.
- Specific TraMineR functions can be combined in a same script with any of the numerous basic statistical procedures of R as well as with those of any other R-package.

Before describing the usage of the TraMineR package for R, a few remarks are worth on the nature of sequence data considered in the particular field of social sciences. In the social sciences, sequence data represent typically longitudinal biographical data such as employment histories or family life courses. Following for instance Brzinsky-Fay et al. (2006) we may simply define a *sequence* as an ordered list of states (employed/unemployed) or events (leaving parental home, marriage, having a child). For now let us just retain that there are multiple other ways of representing longitudinal data that will be discussed in more details in Chapter 4 and that TraMineR will prove useful for converting from one form to the other.

[2] Minor changes may be needed in case of references to file names and paths or other interactions with the OS.

Chapter 2

A short example to begin with

Nothing is better than an example to present the features of TraMineR. We will use for this purpose an example data set from McVicar and Anyadike-Danes (2002) which has been included with the package (see Section 3.2). The data stems from a survey on transition from school to work and contains 72 monthly activity state variables from July 1993 to June 1999 for 712 individuals.

All the following commands show the process of analysing a sequence data set and can be issued by a user who has R and TraMineR installed [1] on his computer.

2.1 State sequence analysis

1. Loading the TraMineR library and the *mvad* example data set

```
R> library(TraMineR)
R> data(mvad)
```

2. Defining a vector containing the legends for the states to appear in the graphics and creating a sequence object which will be used as argument to the next functions (see Chapter 6)

```
R> mvad.labels <- c("employment", "further education", "higher education",
+       "joblessness", "school", "training")
R> mvad.scode <- c("EM", "FE", "HE", "JL", "SC", "TR")
R> mvad.seq <- seqdef(mvad, 17:86, states = mvad.scode,
+       labels = mvad.labels, xtstep = 6)
```

3. Drawing in a single figure [2] (Fig. 2.1)

   - the index plot of the first 10 sequences (see Section 7.3)

   ```
   R> seqiplot(mvad.seq, withlegend = F, title = "Index plot (10 first sequences)",
   +       border = NA)
   ```

   - the sequence frequency plot of the 10 most frequent sequences with bar width proportional to the frequencies (see Section 7.2)

   ```
   R> seqfplot(mvad.seq, withlegend = F, border = NA, title = "Sequence frequency plot")
   ```

   - the state distribution by time points (see Section 7.2.2)

   ```
   R> seqdplot(mvad.seq, withlegend = F, border = NA, title = "State distribution plot")
   ```

[1] To download R, go to http://www.r-project.org/. Installing TraMineR is as straightforward as typing `install.packages("TraMineR")` within a R console

[2] The following command is issued first to set the graphical display `par(mfrow=c(2,2))`

- the legend as a separate graphic since several plots use the same color codes for the states

```
R> seqlegend(mvad.seq, fontsize = 1.3)
```

4. Plot the entropy of the state distribution at each time point (Fig. 2.2)

```
R> seqHtplot(mvad.seq, title = "Entropy index")
```

5. Compute, summarize and plot the histogram (Fig. 2.2) of the sequence turbulences (see Section 7.3).

```
R> Turbulence <- seqST(mvad.seq)
R> summary(Turbulence)
R> hist(Turbulence, col = "cyan", main = "Sequence turbulence")
```

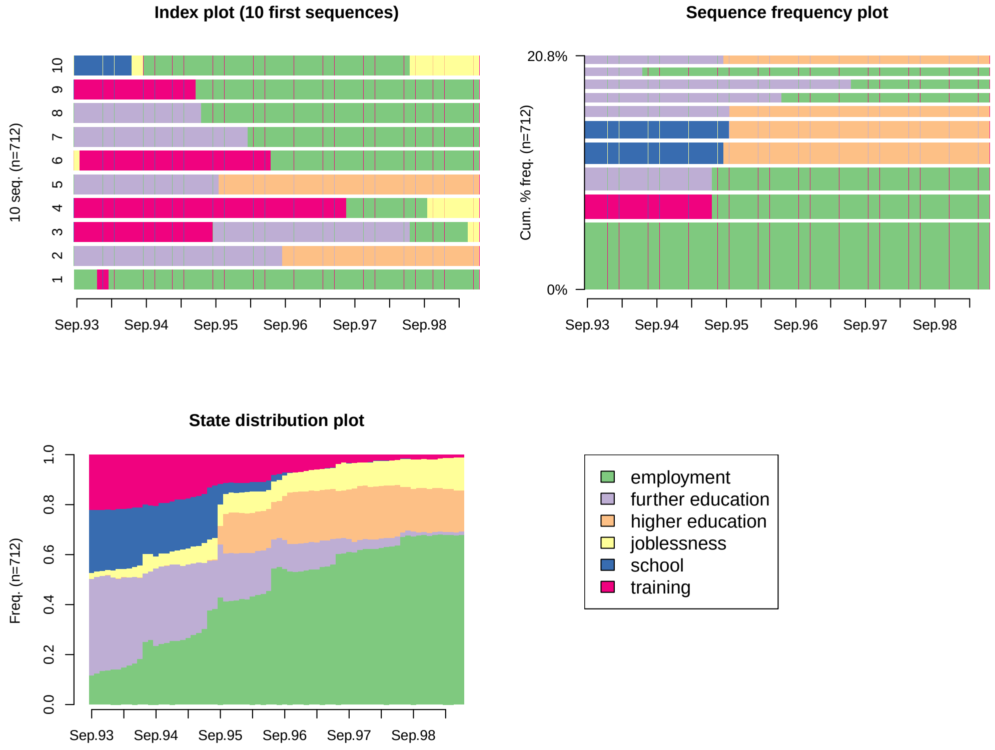


Figure 2.1: A short example - Plot of 10 first sequences (top-left), plot of 10 most frequent sequences (top-right) and state distribution plot (bottom-left) - *mvad* data set

6. Compute the optimal matching distances using substitution costs based on transition rates observed in the data and a 1 indel cost (see Section 9.4). The resulting distance matrix is stored in the `dist.om1` object.

```
R> submat <- seqsubm(mvad.seq, method = "TRATE")
R> dist.om1 <- seqdist(mvad.seq, method = "OM", indel = 1,
+       sm = submat)
```

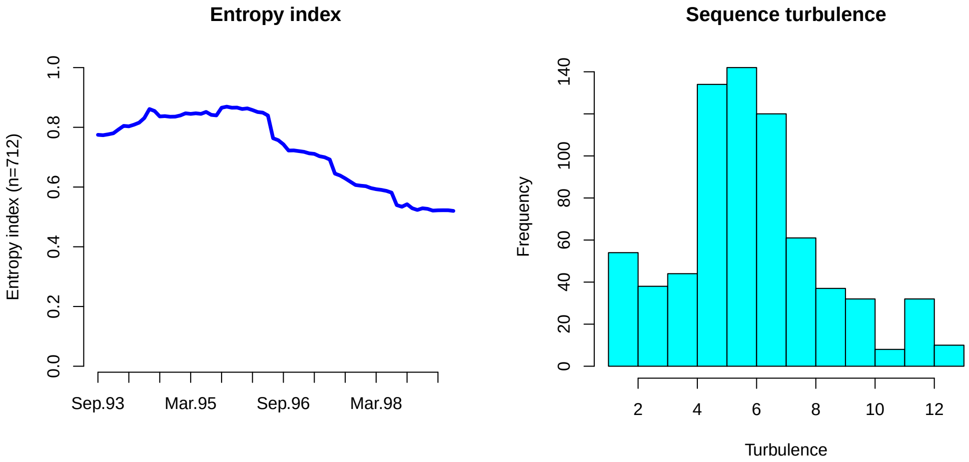


Figure 2.2: A short example - Entropy of the state distribution (left) and and histogram of sequence turbulence (right) - *mvad* data set

7. Make a typology of the trajectories: load the `cluster` library, build a Ward hierarchical clustering of the sequences from the optimal matching distances and retrieve for each individual sequence the cluster membership of the 4 class solution (see Section 9.5). We do not show here the dendrogram produced by `plot(clusterward1)` which, indeed, is not a TraMineR feature.

```
R> library(cluster)
R> clusterward1 <- agnes(dist.om1, diss = TRUE, method = "ward")
R> plot(clusterward1)
R> cl1.4 <- cutree(clusterward1, k = 4)
R> cl1.4fac <- factor(cl1.4, labels = paste("Type", 1:4))
```

8. Plot the state distribution at each time point within each cluster (Fig. 2.3, see Section 9.5)

```
R> seqdplot(mvad.seq, group = cl1.4fac, border = NA)
```

9. Plot the sequence frequencies within each cluster (Fig. 2.4, see Section 9.5)

```
R> seqfplot(mvad.seq, group = cl1.4fac, border = NA)
```

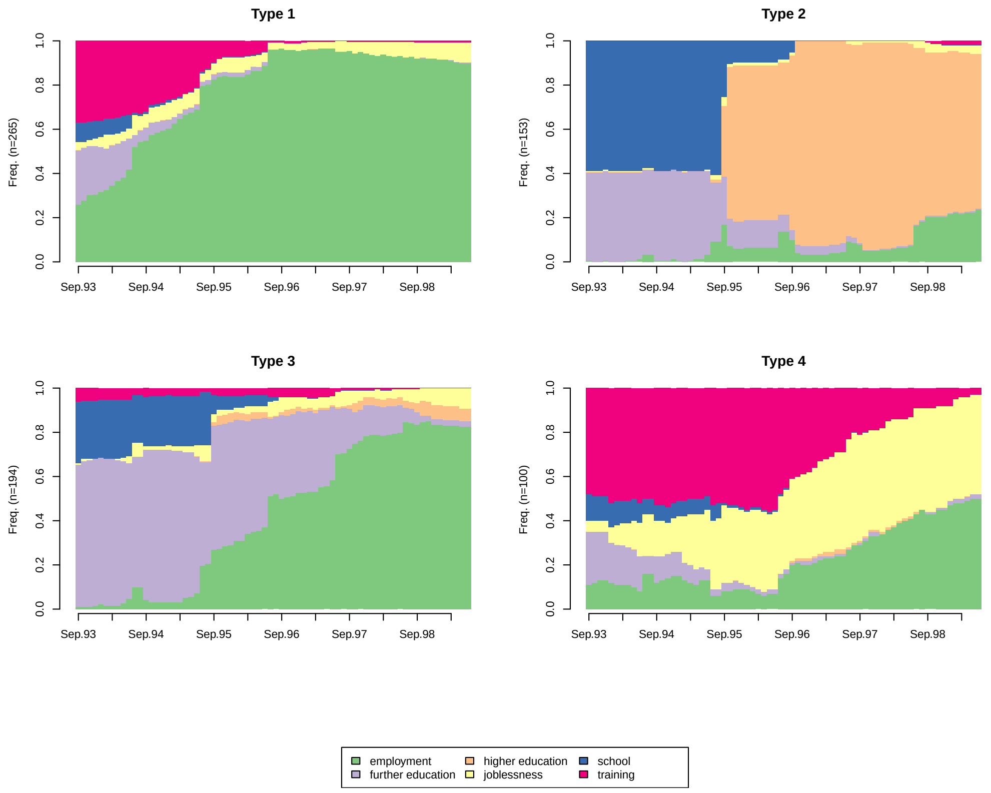


Figure 2.3: A short example - State distribution within each cluster (*mvad* data)

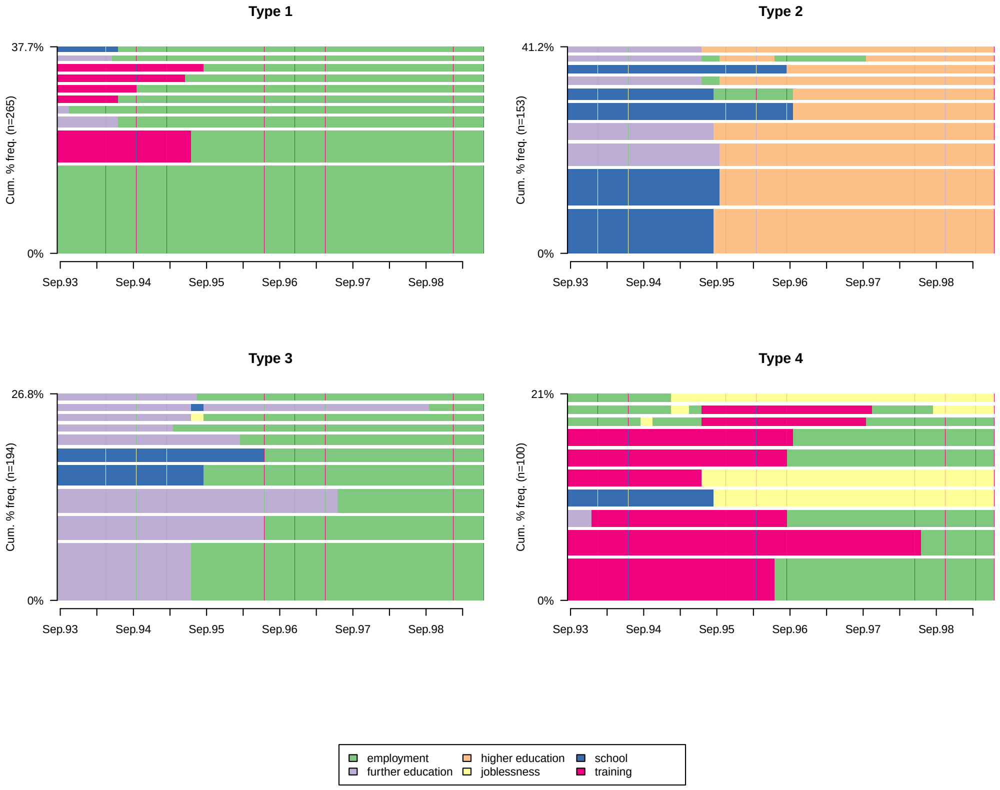


Figure 2.4: A short example - Sequence frequencies whithin each cluster (*mvad* data)

2.2 Event sequence analysis

Instead of focusing on sequences of states, we can look at sequences of transitions or events . TraMineR offers specific tools to deal with such kind of data. For dealing with such event sequences, we can:

1. Define the sequences of transitions (see Section 10.5.4)

   ```
   R> mvad.seqe <- seqecreate(mvad.seq)
   ```

2. Look for frequent event subsequences and plot the 15 most frequent ones (Fig. 2.5, see Section 10.2)

   ```
   R> fsubseq <- seqefsub(mvad.seqe, pMinSupport = 0.05)

   R> plot(fsubseq[1:15], col = "cyan")
   ```

3. Determine the most discriminating transitions between clusters and plot the frequencies by cluster of the 6 first ones (Fig. 2.6, see Section 10.4)

   ```
   R> discr <- seqecmpgroup(fsubseq, group = cl1.4fac)

   R> plot(discr[1:6])
   ```

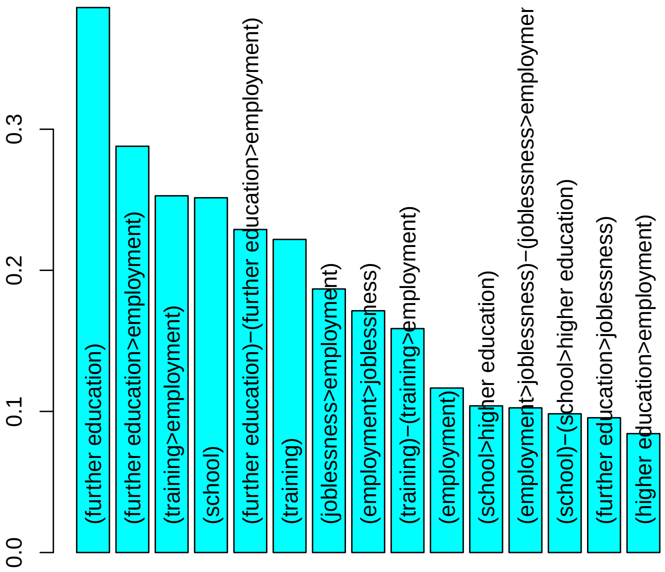


Figure 2.5: A short example - Frequencies of most frequent transitions (*mvad* data)

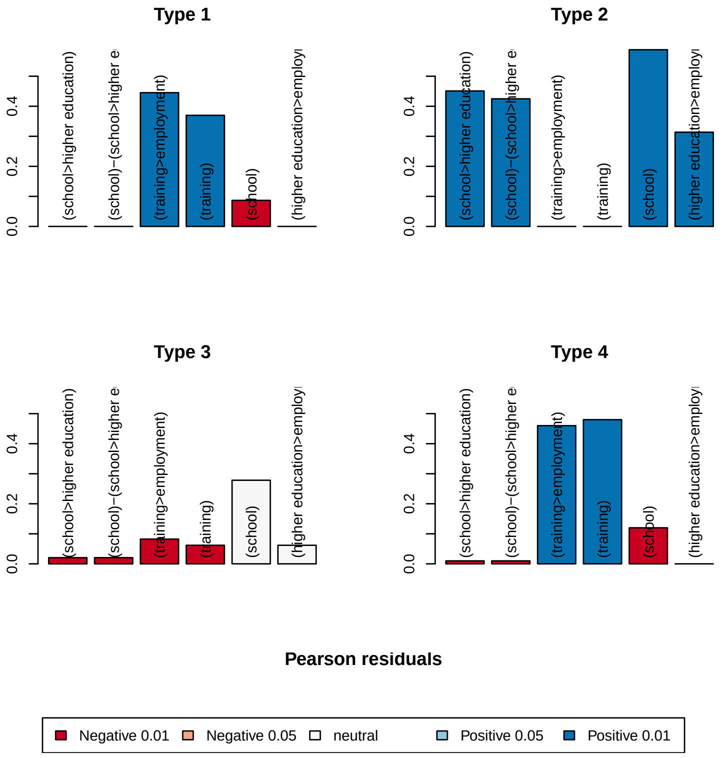


Figure 2.6: A short example - Most discriminating transitions between clusters (*mvad* data)

Chapter 3

The TraMineR package

TraMineR is an add-on package to R, providing a set of functions for describing, visualizing and analysing sequence data, together with example data sets. The latter are used in this manual to demonstrate the multiple powerful features offered by the package.

Depending on your system, TraMineR can be installed either from a precompiled binary package (Windows and Mac OS/X) or from source files (Linux and other UNIXes). The installation of the latest version of the package can be done within an R console by typing:

```
R> install.packages("TraMineR", repos="http://mephisto.unige.ch/traminer/R")
```

The required files are automatically downloaded from our local repository. If you are running the latest R version, you can also install from the Comprehensive R Archive Network (CRAN) http://cran.r-project.org/ by typing:

```
R> install.packages("TraMineR")
```

For more detail on how to install or update TraMineR, see the instructions here: http://mephisto.unige.ch/traminer/download.shtml.

This chapter describes the basic use of TraMineR and presents the included data sets that will be used in this manual to demonstrate the package capabilities.

3.1 Loading, using and getting help

Loading Once you have installed TraMineR on your system you have to load it to access its functionalities. This is done by means of the `library()` command. Typing

```
R> library(TraMineR)
```

gives you access to the functions and data sets provided by the library. This command has to be issued each time you start a new R session, but needs to be issued only once by session. All the examples in the remaining of this manual assume that the TraMineR library is already loaded.

You get information about the installed package such as the version number and the list of functions and data sets provided by issuing the command

```
R> library(help = TraMineR)
```

The above command opens a help window. The content of the obtained help window is shown in Appendix B.

Using the functions TraMineR functions are just like other R functions. To call them, you just type in the function name and the requested arguments surrounded with parentheses. Most TraMineR functions require at least the name of a sequence object created with the `seqdef()` or the `seqecreate()` functions (see Chapters 5 and 10) and (optionally) the values for some specific arguments.

If the arguments are given in the order expected by the function, you can omit the argument names before their values. Arguments with assigned default values can be omitted, unless you want to specify a different value. However, always specifying the names of the arguments is more secure since:

- Adding a new optional argument to a function in a new version of TraMineR may change the order of the arguments, in which case your programs would fail when the names of the arguments are not specified.
- Scripts are easier to understand (by you and by others) when the name of each used argument is explicitly specified.

The `seqdef()` function is used to illustrate how to specify arguments. This command is one of the first you will issue since it defines the sequence object requested by most of the other functions provided by the TraMineR package. The main arguments of `seqdef()` are[1]:

- `data`, the name of a data frame;
- `var`, which specifies the variables (names or index numbers of columns) containing the sequence information (default value is 'NULL', meaning all the variables in the data set);
- `informat`, which specifies the format of the sequences (default value is 'STS', the most common sequence format).

The function `seqdef()` accepts additional arguments (`stsep`, `alphabet`, `states`, `start`, `missing`, `cnames`) that are described later in this manual (see Chapter 5). The name of the data frame is mandatory, but the other arguments have default values and can be omitted if their values are suitable to you. The options can be given in any order if you specify the argument names before their values:

```
R> data(actcal)
R> actcal.seq <- seqdef(var = 13:24, data = actcal)
```

In this example, not specifying the argument names `var=` and `data=` generates an error message

Getting help To get help about a specific function, `seqdef` for instance, type

```
R> `?`(seqdef)
```

or

```
R> help(seqtab)
```

Updating and new features The `update.packages()` function can be used to automatically compare the version numbers of installed packages with the newest available version on the repositories and update outdated packages on the fly.

Informations on new features added to updated versions of the package are described in the `NEWS` file (see http://cran.r-project.org/web/packages/TraMineR/index.html).

[1] you can use `?seqdef` or `help(seqdef)` or the reference manual to see what the expected arguments are

3.2 Data sets included in the TraMineR package

Several sequence data sets used in this manual are included in the TraMineR package and can be loaded in memory using the `data()` function. The *actcal* and *biofam* data sets were created from the Swiss Household Panel[2], SHP, data (http://www.swisspanel.ch/.)

3.2.1 The *actcal* data set

The next example shows how to load the *actcal* data set, list the names of its columns and display the content of the first row. You may get an overview and summary statistics of the whole *actcal* data set by issuing the `summary(actcal)` command (output not shown).

```
R> data(actcal)
R> names(actcal)
```

```
 [1] "idhous00" "age00"    "educat00" "civsta00" "nbadul00" "nbkid00"
 [7] "aoldki00" "ayouki00" "region00" "com2.00"  "sex"      "birthy"
[13] "jan00"    "feb00"    "mar00"    "apr00"    "may00"    "jun00"
[19] "jul00"    "aug00"    "sep00"    "oct00"    "nov00"    "dec00"
```

```
R> actcal[1, ]
```

```
     idhous00 age00 educat00 civsta00 nbadul00 nbkid00 aoldki00 ayouki00
2848    60671    47 maturity  married        3       2       17       14
                         region00                              com2.00
2848 Middleland (BE, FR, SO, NE, JU) Industrial and tertiary sector communes
       sex birthy jan00 feb00 mar00 apr00 may00 jun00 jul00 aug00 sep00 oct00
2848 woman   1953     B     B     B     B     B     B     B     B     B     B
     nov00 dec00
2848     B     B
```

This data set contains a sample of 2000 records of individual monthly activity statuses from January to December 2000, with the activity statuses coded as described in Table 3.1. In addition, it contains also (first 12 columns) some covariates gathered at the individual and household level. The variables in the data set are listed in Table 3.2. Sequences are in the columns named ‘jan00’, ‘feb00’, etc... The row labels are just id numbers. Notice that the numbering is not consecutive. This is because cases were randomly selected.

Each row contains a sequence of states, i.e. activity statuses, reported by a respondent to the wave of year 2000 of the SHP survey. The respondent whose activity calendar is in row 1 stayed in a part-time (19-36 hours per week) payed job during the whole period. The respondent in row 2 (labeled 1230) had no job between January and April 2000, then worked full-time between May and November 2000, and had no remunerated job in December 2000. Note that row names are arbitrary character strings that can be easily modified (we explain how in the appendix; see paragraph A.3.4, p. 117).

3.2.2 The *biofam* data set

The *biofam* data set was constructed by Müller et al. (2007) from the data of the retrospective biographical survey carried out by the Swiss Household Panel in 2002. In includes only individuals who were at least 30 years old at the time of the survey for whom we consider sequences of their

[2] Those example data sets are random samples drawn from the original files and are only used for documenting the package. Persons interested in using the data from the Swiss Household Panel for their research must sign a data protection contract to get access to the complete and original files.

family life states between ages 15 and 30. The *biofam* data set is a random sample of size 2000 of the original data set. It describes the family life courses of individuals born between 1909 and 1972. The possible states are numbered from 0 to 7 and were derived from time stamped event sequences using the coding of Table 3.3. The list of variables is shortly described in Table 3.4.

3.2.3 The *mvad* data set

This data set used and described by McVicar and Anyadike-Danes (2002) is now included in the TraMineR package with permission of the authors and the Journal of the Royal Statistical Society. The data covers 712 individuals. Each individual is characterized by 14 variables, including a unique identifier (id), and 72 monthly activity state variables from July 1993 to June 1999. The complete list of variables is given in Table 3.5. Here we show the first row of the *mvad* data frame

```
R> data(mvad)
R> mvad[1, ]
```

```
  id weight male catholic Belfast N.Eastern Southern S.Eastern Western Grammar
1  1   0.33   no       no      no        no       no        no     yes      no
  funemp gcse5eq fmpr livboth   Jul.93   Aug.93     Sep.93     Oct.93
1     no      no  yes     yes training training employment employment
      Nov.93     Dec.93   Jan.94   Feb.94     Mar.94     Apr.94     May.94
1 employment employment training training employment employment employment
```

Table 3.1: State definition for the activity calendar (*actcal* data set)

Code	Status
A	full-time paid job (37 hours or more per week)
B	part-time paid job (19-36 hours per week)
C	part-time paid job (1-18 hours per week)
D	no work / unemployment / other

Table 3.2: Covariates and state variables of the activity calendar (*actcal* data set)

Variable	Label
age00	age in 2000
educat00	education level in 2000
civsta00	civil status of the respondent in 2000
nbadul00	number of adults in the household
nbkid00	number of children under 15 in the household
aoldkid00	age of the oldest kid in the household
ayoukid00	age of the youngest kid in the household
region00	region the household is living in
com2.00	type of community the household is living in
sex	sex of the respondent
birthy	birth year of the respondent
jan00	activity status for January 2000
:	:
dec00	activity status for December 2000

Table 3.3: State definition for the *biofam* data set

State	Leaved parental home	Married	Children	Divorce
0	no	no	no	no
1	yes	no	no	no
2	no	yes	yes/no	no
3	yes	yes	no	no
4	no	no	yes	no
5	yes	no	yes	no
6	yes	yes	yes	no
7	yes/no	yes/no	yes/no	yes

Table 3.4: List of Variables in the *biofam* data set

Variable	Label
idhous	household number
sex	sex of the respondent
birthy	birth year of the respondent
nat_1_02	first nationality of the respondent
plingu02	interview language
p02r01	Confession or religion
p02r04	Participation in religious services: Frequency
cspfaj	Swiss socio-professional category: Fathers job
cspmoj	Swiss socio-professional category: Mothers job
a15	family formation status at age 15
:	:
a30	family formation status at age 30

```
      Jun.94     Jul.94     Aug.94     Sep.94     Oct.94     Nov.94     Dec.94
1 employment employment employment employment employment employment employment
      Jan.95     Feb.95     Mar.95     Apr.95     May.95     Jun.95     Jul.95
1 employment employment employment employment employment employment employment
      Aug.95     Sep.95     Oct.95     Nov.95     Dec.95     Jan.96     Feb.96
1 employment employment employment employment employment employment employment
      Mar.96     Apr.96     May.96     Jun.96     Jul.96     Aug.96     Sep.96
1 employment employment employment employment employment employment employment
      Oct.96     Nov.96     Dec.96     Jan.97     Feb.97     Mar.97     Apr.97
1 employment employment employment employment employment employment employment
      May.97     Jun.97     Jul.97     Aug.97     Sep.97     Oct.97     Nov.97
1 employment employment employment employment employment employment employment
      Dec.97     Jan.98     Feb.98     Mar.98     Apr.98     May.98     Jun.98
1 employment employment employment employment employment employment employment
      Jul.98     Aug.98     Sep.98     Oct.98     Nov.98     Dec.98     Jan.99
1 employment employment employment employment employment employment employment
      Feb.99     Mar.99     Apr.99     May.99     Jun.99
1 employment employment employment employment employment
```

Table 3.5: List of Variables in the *MVAD* data set

id	unique individual identifier
weight	sample weights
male	binary dummy for gender, 1=male
catholic	binary dummy for community, 1=Catholic
Belfast	binary dummies for location of school, one of five Education and Library Board areas in Northern Ireland
N.Eastern	”
Southern	”
S.Eastern	”
Western	”
Grammar	binary dummy indicating type of secondary education, 1=grammar school
funemp	binary dummy indicating father's employment status at time of survey, 1=father unemployed
gcse5eq	binary dummy indicating qualifications gained by the end of compulsory education, 1=5+ GCSEs at grades A-C, or equivalent
fmpr	binary dummy indicating SOC code of father?s current or most recent job,1=SOC1 (professional, managerial or related)
livboth	binary dummy indicating living arrangements at time of first sweep of survey (June 1995), 1=living with both parents
jul93	Monthly Activity Variables are coded 1-6, 1=school, 2=FE, 3=employment, 4=training, 5=joblessness, 6=HE
:	”
jun99	”

3.2.4 Other data sets borrowed from the literature

The *famform* data set is a small illustrative data set of family forms used by Elzinga (2008). It consists in 5 sequences of length 5, some having missing values (NA). The states are: single (‘S’), with unmarried partner (‘U’), married (‘M’), married with a child (‘MC’), single with a child (‘SC’). The five sequences in the data are

```
v  "S"  "U"
w  "S"  "U"  "M"
x  "S"  "U"  "M"   "MC"
y  "S"  "U"  "M"   "MC"  "SC"
z  "U"  "M"  "MC"
```

where the first column contains case labels.

3.3 Performance and memory usage

Depending on your system and the size of your data, some functions for sequence data analysis may have a consequent time and memory consumption, especially the computation of distances between sequences. However, as the critical functions are written in C, the speed performance of the functions in TraMineR compares quite advantageously with other packages that deal with sequence analysis. For instance, it is almost as efficient as TDA and outperforms Brzinsky-Fay et al. (2006)'s package for Stata. Nonetheless, the number of distances to compute increases rapidly with the size of the dataset. For a 4000 sequences dataset, the number of distances to compute is $(4000 * 3999)/2 = 7'998'000$ whereas it is $(10000 * 9999)/2 = 49'995'000$ for 10000 sequences [3].

With R, the size of the data you can handle is limited by the available memory size on your system (at least on Linux systems). Remember that from the moment that you compute a distance matrix, the requested memory size increases dramatically. TraMineR has been succesfull in computing the distance matrix for as much as 30'328 sequences, but the size of the (half) distance matrix was 6.85GB. To give a more common example, computing optimal matching distances for the 4'318 sequences of length 16 (841 distinct sequences) of the original data set from which *biofam* was extracted takes less than 15 seconds on a dual core processor. The resulting 4318 × 4318 distance matrix has a size of 142Mb. Table 3.6 gives computation time and memory usage for some typical examples. The reported computation times concern version 1.0 of TraMineR. Improvements in version 1.1 permitted to reduce the indicated times by a factor of at least 10 for large data sets.

If you get some message claiming about a lack of memory, you should try `gc()` to free memory from 'garbages' that may be produced by some memory consuming functions. The computation of distances between sequences was faster with version 2.6 and 2.7 of R compared with version 2.5.

Table 3.6: Performance and memory usage

Number of seq.	Seq. length	System	Time	Matrix size
712	72	Intel Core 2 @ 2.13GHz / 2Gb RAM	21 s	3.9Mb
4'318	16	Intel Core 2 @ 2.13GHz / 2Gb RAM	15 s	142Mb
30'328	77	4x Quad Core 64-bit Xeon CPUs @ 2.4 GHz / 64GB RAM	54 mn	6.85Gb

[3] To reduce the number of distances to compute, TraMineR first selects the set of unique sequences

Chapter 4

Definition and representation of longitudinal data formats

In Section 1.1, we defined sequences as ordered lists of states or events. However, sequence representation in data files can vary a lot, depending on the way data were collected and the way information is organized. In numerous cases, sequences are even not present “as such” in the data but can be reconstructed from data originally collected as spells, time stamped events or other forms.

Hence, a crucial preliminary step in sequential data analysis is preparing the data to organize it in the form expected by the functions we want to use. This is often a cumbersome discouraging task and the literature does not offer much to help identifying the main types of sequential data organization and formats, Giele and Elder (1998) being one of the rare exception. Conscious of the importance of the issue, we devoted a lot of effort on these aspects when developing the package.

TraMineR provides thus a unique set of features for handling and converting data to and from several different formats. This Chapter describes these formats and Chapter 5 details the data management tools available in TraMineR.

4.1 Ontology

Before defining and describing the main formats and representations of sequence data, we begin with an ontology of longitudinal data. This ontology describes the main attributes we can use to identify the various formats and characterize the nature of the sequences the user will have to deal with.

4.1.1 States and events

One first distinction between the several data types is whether the basic information they contain are states or events. Broadly, in a longitudinal framework, each change of state is an event and each event implies a change of state. However, the state that results from an event may also depend on the previous state, and hence of which other events already occurred. The states of the *biofam* data set were for instance derived from the combination of 4 events as described in Table 3.3 page 22. Conversion between state sequences and event sequences is thus not always straightforward.

Figure 4.1 shows a graphical representation for 10 sequences. Here the sequences are ordered list of states, with the states being the work status of the corresponding respondent at each time unit, i.e. months from January to December 2000. Though the sequences are ordered lists of states, they provide also some information about events, especially if we consider events as simple changes of states. In sequence number 1 (first one from the bottom), no event occurred during the observation period since the respondent stays in the same state during the whole sequence. In sequence 2 (second from bottom), two events occurred:

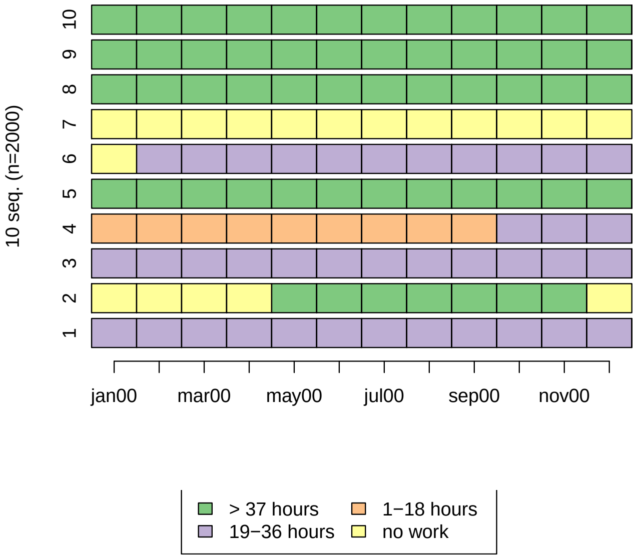


Figure 4.1: First 10 sequences of the *actcal* data (first at bottom)

- The respondent changed his work status between time unit 4 (April 2000) and time unit 5 (May 2000), from ‘no work’ to ‘full time paid work’.
- Then, the respondent changed again his work status between time unit 11 (November 2000) and time unit 12 (December 2000), from ‘full time paid work’ to ‘no work’.

States or events can be coded with letters, character strings or digits. The ***alphabet*** is the list of all possible states or events appearing in the data. In the following example taken from Aassve et al. (2007), states are coded with character strings of length 3 and separated by the ‘-’ character. We will see other formats to represent such sequences in the following sections.

```
R> seq.ex1[, 10:25]

    Sequence
[1] 000-000-000-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0WU-0WU-0WU-0WU
[2] 000-000-000-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0
```

For each state in the sequence, the first character stands for the number of children (0=no children, 1=1 children, etc...), the second character for the work status (0=not working, W=working) and the third character for the union status (0=not in union, U=in union). The alphabet contains 16 distinct states (see Table 1, page 376 in Aassve et al., 2007).

4.1.2 Single or multichannel

In the previous example, each distinct state is actually a combination of states pertaining to different domains: work status, number of children and union status. The combination of all possible states in each domain yields an alphabet of 16 distinct states. As mentioned by Aassve et al., 2007, “the number of possible states available in different time periods implies that the frequency of any specific sequence will be very low”.

An alternative is to handle sequences of each domain separately. This is called *multichannel sequences.*

4.1.3 Time reference: Internal and external clocks

Unlike biological sequences for instance, trajectories in social sciences are usually defined on a time axis. The information about time is an important part of sequence data when timing and/or duration is a concern as in life course analysis.

In the case of sequences of states, it is important to know whether the alignment of states is done according to

- an internal time reference (e.g. age of the individual, such as in the *biofam* dataset)

- or to an external time reference (e.g. January to December 2000, such as in the *actcal* dataset).

One typology of the discrete time axis on which the sequences of states are defined has been proposed by Rohwer and Pötter (2002, Sec. 3.4.1). The authors distinguish between

- a **calendar** time axis which does not have a natural origin. Fixing an origin is simply a convention for providing time points.

- a **process** time axis where the origin represents the date of a starting event.

4.1.4 One or several rows per individual

The most natural way of presenting sequence data is to use one row per case. However, using several rows for data belonging to a same individual may also have its advantages. A first example is provided by the multichannel context in which it may be worth to explicitly distinguish between sequences belonging to different domains or aspects (living arrangement, civil status, education, professional, ...).

In longitudinal analysis it is also sometimes more convenient to use a distinct row

- by time unit lived by each individual: States of the different channels will be in columns; such data presentation is commonly called *person-period data.*

- by spell lived by each individual: Each rows defines the states in which the individual is during the spell; this presentation is called *spell data* and requires indeed to specify the spell start and end time, or equivalently start time and duration.

- by episode lived by each individual, i.e. a row for each date at which one or more events occur. In this case, the row contains the time stamp and the list of events that occur; this kind of presentation is for instance useful for mining frequent event sub-sequences.

4.1.5 Ontology

An ontology of sequence data formats can be defined by a nested suite of ‘yes/no’ questions about properties of the format. Figure 4.2 shows an ontology of types of longitudinal data, i.e. data organized according to time.

4.2 Longitudinal data representations

Using some elements of the ontology, Table 4.1 defines several data formats. The basic information used to identify them is whether the elements are states or events, and whether the format uses a single row or more than one for each case. Table 4.2 gives examples of the listed formats. The latter as well as some other formats are described in details below in the present Section with indication of whether they are supported by TraMineR.

4.2.1 The 'states-sequence' (STS) format

The 'STates-Sequence' (STS) format is the internal format used by TraMineR (in TraMineR, sequences are stored in sequence objects, see next section). It is one of the most intuitive and common way of representing a sequence. In this format, the successive states (statuses) of an individual are given in consecutive columns. Each column is supposed to correspond to a predetermined time unit, but sequences of states with no time reference can be handled as well using the same format. In the *actcal* data set previously described (see Sec. 3.2.1), sequences are in columns 13 to 24 representing the monthly activity statuses from January to December 2000.

```
R> actcal[1:6, 13:24]
```

```
     jan00 feb00 mar00 apr00 may00 jun00 jul00 aug00 sep00 oct00 nov00 dec00
2848     B     B     B     B     B     B     B     B     B     B     B     B
```

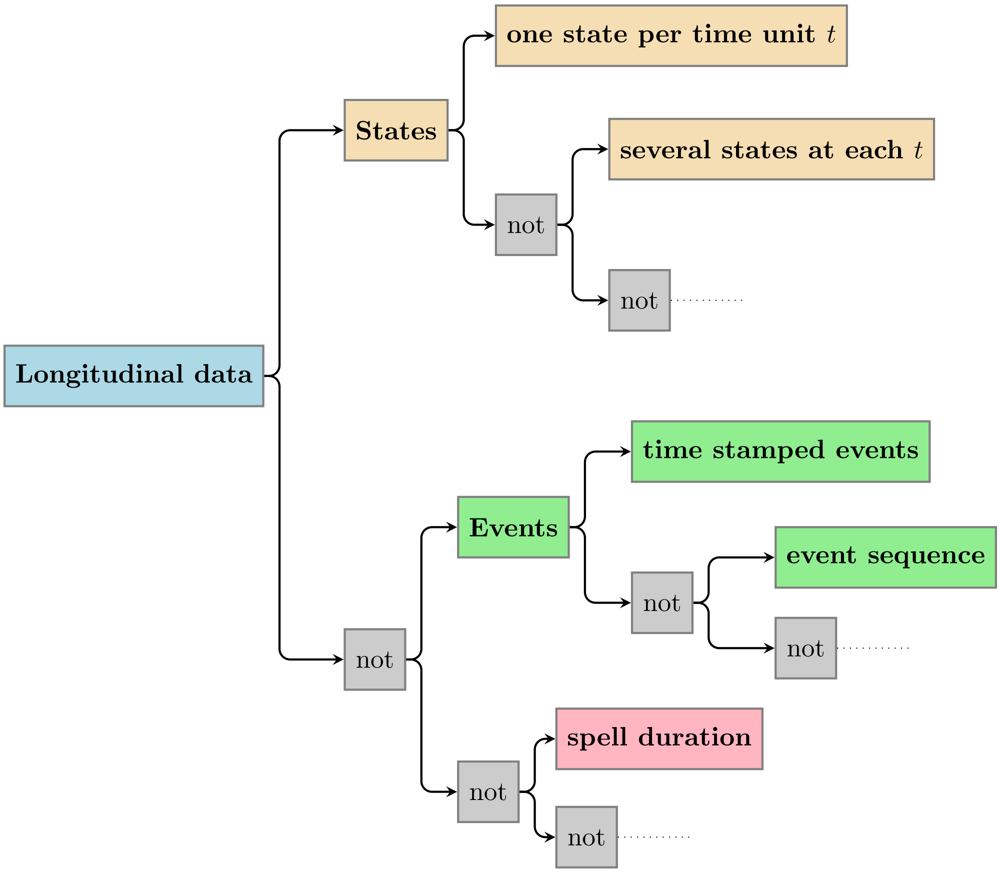


Figure 4.2: Ontology of types of longitudinal data

Table 4.1: Sequence data representations

Code	Data type	(S)tates or (E)vents	One (1) or several (M) rows for each individual	Import into a sequence object
STS	State-sequence	S	1	Yes
SPS	State-permanence (1)	S	1	Yes
DSS	Distinct-State-Sequence	S	1	Yes (use STS)
TSE	Time-stamped event	E	M	Yes (event sequence)
SPELL	Spell	S	M	Yes
	Person-period		M	

Table 4.2: Sequence data representations: Examples

Code	Example										
STS	**Id**	**18**	**19**	**20**	**21**	**22**	**23**	**24**	**25**	**26**	**27**
	101	S	S	S	M	M	MC	MC	MC	MC	D
	102	S	S	S	MC	MC	MC	MC	MC	MC	MC
SPS (1)	**Id**	**State 1**	**State 2**	**State 3**	**State 4**	**State 5**					
	101	(S,3)	(M,2)	(MC,4)	(D,1)						
	102	(S,3)	(MC,7)								
SPS (2)	**Id**	**State 1**	**State 2**	**State 3**	**State 4**	**State 5**					
	101	S/3	M/2	MC/4	D/1						
	102	S/3	MC/7								
DSS	**Id**	**State 1**	**State 2**	**State 3**	**State 4**	**State 5**					
	101	S	M	MC	D						
	102	S	MC								
TSE	**id**	**time**	**event**								
	101	21	Marriage								
	101	23	Child								
	101	27	Divorce								
	102	21	Marriage								
	102	21	Child								
SPELL	**id**	**index**	**from**	**to**	**status**						
	101	1	18	20	Single						
	101	2	21	22	Married						
	101	3	23	26	Married w Children						
	101	4	27	..	Divorced						
	102	1	18	20	Single						
	102	2	21	27	Married w Children						

```
1230    D     D     D     D     A     A     A     A     A     A     A     D
2468    B     B     B     B     B     B     B     B     B     B     B     B
654     C     C     C     C     C     C     C     C     C     B     B     B
6946    A     A     A     A     A     A     A     A     A     A     A     A
1872    D     B     B     B     B     B     B     B     B     B     B     B
```

4.2.2 The ‘state-permanence-sequence’ (SPS) format

The ‘SPS’ format, whose name ‘State-Permanence-Sequence’ is due to Aassve et al., 2007, is for instance used by Elzinga (2008). In this format, each successive distinct state in the sequence is given together with its duration. In one variant, each state/duration couple is enclosed into parentheses. The example below is taken from Aassve et al., 2007.

```
R> print(seq.ex1, format = "SPS")

    Sequence
[1] (000,12)-(0W0,9)-(0WU,5)-(1WU,2)
[2] (000,12)-(0W0,14)-(1WU,2)
```

This format is an alternative way of representing ‘STS’ sequences. Here are the same sequences as they are internaly stored in a sequence object by TraMineR

```
R> print(seq.ex1, ext = TRUE)

    [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17]
[1] 000 000 000 000 000 000 000 000 000  000  000  000  0W0  0W0  0W0  0W0  0W0
[2] 000 000 000 000 000 000 000 000 000  000  000  000  0W0  0W0  0W0  0W0  0W0
    [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28]
[1]  0W0  0W0  0W0  0W0  0WU  0WU  0WU  0WU  0WU  1WU  1WU
[2]  0W0  0W0  0W0  0W0  0W0  0W0  0W0  0W0  0W0  1WU  1WU
```

4.2.3 The vertical ‘time-stamped-event’ (TSE) format

A time-stamped-event representation consists in listing the events experienced by each individual together with the time at which the events occurred. Sequences of events can easily be constructed from this representation. It is also possible in TraMineR to translate sequence data into such time-stamped event (TSE) representation at the cost, however, of providing event definition information (see Section 5.2.2 page 40). Each record of the TSE representation usually contains a case identifier, a time stamp and codes identifying the event occurring. In the following example, 3 events, coded 5, 7 and 9, are observed at age (time) 25 for the individual 70102. Individual 215102 experiences one event (1) at age 6, two events (5, 17) at age 21, two events (7, 18) at age 22 and two events (8, 13) at age 25.

```
R> TSE.ex1

       id time event
1   70102   25     5
2   70102   25     7
3   70102   25     9
4  215102    6     1
5  215102   21     5
6  215102   21    17
7  215102   22     7
8  215102   22    18
9  215102   25     8
10 215102   25    13
```

Table 4.3: Living arrangements - SHP

State	Description
1	with both natural parents
2	with one parent and his/her new partner
3	with one parent alone
4	with relatives or in a foster family
5	with partner (married or not)
6	with friends or in a flat share
7	alone
8	other situation
9	with both natural parents and the partner (married / married
10	with both natural parents and (friends or flat share)
11	with partner (married or not) and (friends or flat share)

4.2.4 The spell (SPELL) format

In the spell format there is one line for each spell. Each spell is characterized by the states (supposed constant during the spell) and the spell start and end times. Hence 'STS' sequences can easily be constructed from this representation. The following example is an extract of data drawn from the retrospective questionnaire of the Swiss Household Panel[1] about living arrangements. Statuses are described in Table 4.3. The first respondent (id 2713) lived with both natural parents from 1965 to 1989, then with a partner from 1989 to 1990 and again with a partner from 1990 to 1991 and from 1991 to 2002 (here we have multiple consecutive spells for the same status; this is because statuses are aggregated from more detailed ones).

```
R> SPELL.ex1
```

```
   idpers index from until status
1    2713     1 1965  1989      1
2    2713     2 1989  1990      5
3    2713     3 1990  1991      5
4    2713     4 1991  2002      5
5    2714     1 1968  1985      1
6    2714     2 1985  1988      7
7    2714     3 1989  1990      5
8    2714     4 1990  1991      5
9    2714     5 1991  2002      5
10   3713     1 1961  1978      1
11   3713     2 1978  1983      3
12   3713     3 1983  1984      4
13   3713     4 1984  1985      3
14   3713     5 1985  1999      4
15   3713     6 1999  2001      7
16  11714     1 1973  1993      1
17  11714     2 1993  2002      5
```

4.2.5 The 'person-period' format

This format is for instance used for discrete-time logistic regressions. Each line contains information about an individual at a different time unit. There is one line for each time unit where the individual

[1] Original personal identification numbers have been modified.

is under observation. Such data presentation is mainly used for discrete survival models where the focus is on a specific event (leaving home, childbirth, death, end of job, etc.) and the time-periods considered are those where the cases are under risk of experimenting the event. In that case, each record contains at least the time stamp and a status variable indicating if the event under study occurred in this time interval, and may possibly be completed with the values of some covariates.

4.2.6 The 'shifted-replicated-sequence' format (SRS)

This data presentation is intended for mobility analysis where the concern is the transition from the states observed at previous time points, $t-1, t-2, \ldots$, to the one observed at time t. Consider for example the sequence A, A, C, D, D where the first element in the sequence corresponds to year 2000 and the last one to year 2004. The shifted-replicated-sequence representation of this sequence is obtained as follows:

```
R> seqs <- data.frame(y2000 = "A", y2001 = "A", y2002 = "C", y2003 = "D",
+       y2004 = "D")
R> seqs
```

```
  y2000 y2001 y2002 y2003 y2004
1     A     A     C     D     D
```

```
R> seqformat(seqs, from = "STS", to = "SRS")
```

```
  id idx  T-4  T-3  T-2  T-1 T
1  1   1 <NA> <NA> <NA> <NA> A
2  1   2 <NA> <NA> <NA>    A A
3  1   3 <NA> <NA>    A    A C
4  1   4 <NA>    A    A    C D
5  1   5    A    A    C    D D
```

In this presentation we collect in the columns named 'T-1' and 'T' all subsequences between $t-1$ and t, and hence all observed transitions between $t-1$ and t, . This is useful when we want t to be a relative time point rather than an absolute date.

4.3 Definition and properties of categorical sequences

The next parts of this manual are dedicated to the analysis of categorical sequences. We define here more precisely as well as some important concepts such as subsequences.

4.3.1 Categorical sequences

For formal definition, we may follow for example Elzinga and Liefbroer (2007). First, define an *alphabet* A as the list of possible states or events. A sequence x of length k is then an ordered list of k successively chosen elements of A. It is often represented by the concatenation of the k elements. A sequence can thus be written as $x = x_1x_2 \ldots x_k$ with $x_i \in A$. We use commas when necessary for distinguishing successive elements in a sequence. For instance, $x = S, U, M, MC$ stands for the sequence *single*, *with unmarried partner*, *married*, *married with a child*.

4.3.2 Time axis

In addition to the sequencing of states or events that the above definitions account for, the information about sequences, especially those describing life courses, includes often a time dimension. When necessary we should then also account either for the time stamp of the states or events, or for the duration of either the states or the time between events. For state sequences over time it is often assumed that each state corresponds to periodic dates (years, months, ...). For event sequences over time, a specific time stamp is most often assigned to each event.

4.3.3 Subsequences

A sequence u is a *subsequence* of x if all successive elements u_i of u appear in x in the same order, which we simply denote by $u \subset x$. According to this definition, unshared states can appear between those common to both sequences u and x. For example, $u = S, M$ is a subsequence of $x = S, U, M, MC$.

Chapter 5

Importing and handling longitudinal data with TraMineR

Results shown in this chapter are obtained with:
TraMineR version 1.6-2
R version 2.9.2 (2009-08-24)-platform: i486-pc-linux-gnu.

Two main preliminary steps are needed for the user to visualize and analyse sequence data with the functions provided by the TraMineR package:

- Import the data into R.

- Create a sequence object (either a state sequence object as described in Chapter 6, or an event sequence object as explained in Section 10.5.4).

In this chapter we first describe shortly how to import data coming from other statistical packages or text files and the way (imported) data is stored in R objects. If your data is already in one of the formats supported by the function that creates sequence objects, you may want to skip the remainder of the chapter and proceed directly to Chapter 6. However, in the second part of the chapter you will learn more about the functions offered by TraMineR for converting to and from several longitudinal data formats. Such transformations may prove useful not only for TraMineR but also for applying other statistical methods to your data such as for instance survival analysis or classification trees.

5.1 Importing data sets into R

Data files generated by statistical programs such as SPSS, SAS and Stata can be directly imported into R by using the `foreign`[1] library and assigned to R objects. We briefly explain hereafter the `read.spss()` command for importing SPSS files and the `read.dta()` command for importing Stata files. Additional details can be found in the R-data manual `http://cran.r-project.org/doc/manuals/R-data.pdf` which provides also explanations regarding other file formats. Data in the form of text files or spreadsheets can also be easily imported.

[1] On Ubuntu Linux (and maybe on other Linux distributions), the `foreign` library is not installed with the basic R installation. You have to install it explicitly on your system with the package manager.

5.1.1 Reading data from other statistical packages

Preliminary remarks. When importing SPSS or Stata files, variables having attached values labels are converted into R factors[2] with levels set to the value labels in the original files. For example, a variable containing states 1, 2, 3, 4 with value labels "single", "living with a partner", "married", "divorced" will be converted into a factor with the four levels "single", "living with a partner", "married", "divorced". Hence the original numerical coding is lost. If you prefer preserving the numerical coding and losing the labels, use the `convert.factors = FALSE` option.

Stata ('.dta') format Here is an example of how to import the living arrangement history data from the biographic questionnaire of the Swiss Household Panel (SHP). We use for that a truncated version of the original *shp0_bvla_user.dta* file that can be found on the SHP CD. This CD can be obtained on request from the SHP, `www.swisspanel.ch`. The R function to import data sets saved in the Stata ('.dta') format is provided by the *foreign* library and reads `read.dta()`. It returns a data frame obect. The `head()` function shows the first 6 rows of the imported data set.

```
R> library(foreign)
R> LA <- read.dta("data/shp0_bvla_user.dta")
R> head(LA)
```

```
  idpers q_source bvla_idx bvla013 bvla014                        bvla100
1   4101     2002        1    1965    1989     with both natural parents
2   4101     2002        2    1989    1990 with partner (married or not)
3   4101     2002        3    1990    1991 with partner (married or not)
4   4101     2002        4    1991    2002 with partner (married or not)
5   4102     2002        1    1968    1985     with both natural parents
6   4102     2002        2    1985    1988                          alone
```

The summary of the *LA* data frame shows that some variables, such as the begin (bvla013) and end of the spell (bvla014) were imported as numeric variables (distribution summarized by quantiles) while the type of living arrangement (bvla100) has been imported as a factor (distribution summarized by a frequency table).

```
R> summary(LA)
```

```
     idpers                    q_source         bvla_idx           bvla013
 Min.   :     4101   2001 (pretest): 2627   Min.   : 0.000   Min.   :  -2
 1st Qu.: 3515102   2002           :18484   1st Qu.: 1.000   1st Qu.:1962
 Median : 7344101                           Median : 3.000   Median :1977
 Mean   : 7286883                           Mean   : 2.885   Mean   :1963
 3rd Qu.:10820101                           3rd Qu.: 4.000   3rd Qu.:1989
 Max.   :14676102                           Max.   :13.000   Max.   :2002

    bvla014                               bvla100
 Min.   :  -2   with partner (married or not)  :7438
 1st Qu.:1974   with both natural parents      :6240
 Median :1989   alone                          :2738
 Mean   :1974   other situation                :1731
 3rd Qu.:2001   with one parent alone          : 961
 Max.   :2002   with friends or in a flat share: 948
                (Other)                        :1055
```

[2] see Appendix A or an introduction to R manual to see what a factor is.

SPSS ('.sav') format Here we read the same data file as in the previous example but from the SPSS version, which is also provided on the SHP CD. The `to.data.frame=TRUE` is specified so that the `read.spss()` function returns a data frame, otherwise it would return a list.

```
R> library(foreign)
R> LA <- read.spss("data/shp0_bvla_user.sav", to.data.frame = TRUE)
```

5.1.2 Reading data from text files

Several functions are available for reading data in various text format: `read.table`, `read.csv`, `read.delim`, `read.fwf`. See `http://cran.r-project.org/doc/manuals/R-data.pdf` for details. An example on how to read a comma separated (CSV) text file is given below with the *mvad* data set described in Section 3.2.4, p. 23. The file can be freely downloaded from `http://www.blackwellpublishing.com/rss/Volumes/Av165p2.htm`. Though the data set is provided with TraMineR as an R data frame, we show below how it was converted and prepared. The steps are the following:

1. Convert the downloaded '.xls' file into a '.csv' (Comma Separated Values) file, using for example Excel or OpenOffice.

2. Run R, and type

   ```
   R> mvad <- read.csv(file = "data/McVicar.csv", header = TRUE)
   ```

   where you should indeed adapt the path "data" to the '.csv' file.

The text file contains only variables with numeric values but most of them are indeed binary indicator variables (see Table 3.5). Let us take an example with the `male` indicator variable. For the moment, this variable is stored as numeric and summarizing it yields quantiles of its distribution.

```
R> summary(mvad$male)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.0000  0.0000  1.0000  0.5197  1.0000  1.0000
```

Hence we convert all indicator variables into factors.

```
R> yn <- c("no", "yes")
R> mvad$male <- factor(mvad$male, labels = yn)
R> mvad$catholic <- factor(mvad$catholic, labels = yn)
R> mvad$Belfast <- factor(mvad$Belfast, labels = yn)
R> mvad$N.Eastern <- factor(mvad$N.Eastern, labels = yn)
R> mvad$Southern <- factor(mvad$Southern, labels = yn)
R> mvad$S.Eastern <- factor(mvad$S.Eastern, labels = yn)
R> mvad$Western <- factor(mvad$Western, labels = yn)
R> mvad$Grammar <- factor(mvad$Grammar, labels = yn)
R> mvad$funemp <- factor(mvad$funemp, labels = yn)
R> mvad$gcse5eq <- factor(mvad$gcse5eq, labels = yn)
R> mvad$fmpr <- factor(mvad$fmpr, labels = yn)
R> mvad$livboth <- factor(mvad$livboth, labels = yn)
```

Now we summarize the data frame

```
R> summary(mvad[, 1:17])
```

```
       id            weight          male      catholic  Belfast   N.Eastern
 Min.   :  1.0   Min.   :0.1300   no :342   no :368   no :624   no :503  
 1st Qu.:178.8   1st Qu.:0.4500   yes:370   yes:344   yes: 88   yes:209  
 Median :356.5   Median :0.6900                                          
 Mean   :356.5   Mean   :0.9994                                          
 3rd Qu.:534.2   3rd Qu.:1.0700                                          
 Max.   :712.0   Max.   :4.4600                                          
 Southern  S.Eastern Western   Grammar   funemp    gcse5eq    fmpr    
 no :497   no :629   no :595   no :583   no :595   no :452   no :537  
 yes:215   yes: 83   yes:117   yes:129   yes:117   yes:260   yes:175  
```

```
 livboth       Jul.93          Aug.93         Sep.93     
 no :261   Min.   :1.000   Min.   :1.00   Min.   :1.000  
 yes:451   1st Qu.:2.000   1st Qu.:2.00   1st Qu.:1.000  
           Median :3.000   Median :3.00   Median :2.000  
           Mean   :3.176   Mean   :3.15   Mean   :2.381  
           3rd Qu.:5.000   3rd Qu.:4.00   3rd Qu.:3.000  
           Max.   :5.000   Max.   :5.00   Max.   :5.000  
```

5.1.3 Data storage in R

A set of sequences, i.e. vectors or strings of states or events, can be stored in several kinds of R objects, namely vectors, matrices, or data frames.

1. A *vector* is a one dimensional object (its size is just its length). Sequences stored in vectors are typically defined as character strings, each sequence being an element of the vector.

2. A *matrix* is a two dimensional object (the two dimensions are rows and columns) containing elements of the same type. Sequences are typically defined as the rows of the matrix, each column giving the state or event at a given time point.

3. *Data frame* is the most common object for storing sequences. It is like a matrix, but can contain objects from different types, for example one or more variables representing sequences (as character strings or vectors of states or events) and covariates. Data sets imported from other statistical packages (See Section 5.1.1) are stored as data frames. The *actcal*, *biofam* and *mvad* data sets are each a data frame object.

5.1.4 Compressed and extended format

In data files, sequences may appear as character strings (what we call the compressed format) or as vectors (what we call the extended format). TraMineR can handle both formats and provides a function to convert between them. For instance, the `seqdef()` and `seqformat()` functions check first whether the data you send them as argument are in the compressed or extended format.[3]

The extended format In the extended format, sequences are given as vectors of states or events, where each state or event is stored in a separate column (variable). Each variable usually corresponds to a time unit as in the example below. The *actcal* data set accompanying the TraMineR package is in the extended format. Each column (variable) contains one state and represents a month of the activity calendar.

[3] This is done by means of the `seqfcheck()` function that searches for the presence of any separator in the data.

```
R> head(actcal[, 13:24])
```

```
     jan00 feb00 mar00 apr00 may00 jun00 jul00 aug00 sep00 oct00 nov00 dec00
2848     B     B     B     B     B     B     B     B     B     B     B     B
1230     D     D     D     D     A     A     A     A     A     A     A     D
2468     B     B     B     B     B     B     B     B     B     B     B     B
654      C     C     C     C     C     C     C     C     C     B     B     B
6946     A     A     A     A     A     A     A     A     A     A     A     A
1872     D     B     B     B     B     B     B     B     B     B     B     B
```

The compressed format In the compressed format, a sequence is represented as a character string. A single string variable is used for storing the sequence. States or events are represented by words or numerical codes separated by a specific separator character[4]. The handling of sequences as character strings without separator is also possible. However, in that case states or events should be represented by single characters or digits. Below the six above sequences from *actcal* are displayed in the compressed format. They were compressed with the `seqconc()` function that is explained below.

```
    Sequence
[1] "B-B-B-B-B-B-B-B-B-B-B-B"
[2] "D-D-D-D-A-A-A-A-A-A-A-D"
[3] "B-B-B-B-B-B-B-B-B-B-B-B"
[4] "C-C-C-C-C-C-C-C-C-B-B-B"
[5] "A-A-A-A-A-A-A-A-A-A-A-A"
[6] "D-B-B-B-B-B-B-B-B-B-B-B"
```

5.2 Converting between formats

Data conversion is done with the `seqformat()`, `seqconc()` and `seqdecomp()` functions described in this section. If you just want to analyse your data with the functions provided by TraMineR, you can directly use the `seqdef()` function described in Chapter 6 and specify the input format. The function `seqdef()` will then automatically call `seqformat()` if necessary. If you want to create event sequences from state sequences for analyzing them with the TraMineR functions dedicated to event sequences, see Chapter 10 for details on how to make such state to event conversions.

5.2.1 Converting between compressed and extended formats

The `seqconc()` and `seqdecomp()` functions convert between compressed and extended representations of sequence data. The following command was used for creating the compressed string vector `actcal.comp` shown above

```
R> actcal.comp <- seqconc(actcal, 13:24)
```

The `seqdecomp()` function makes the reverse transformation to the original uncompressed format. Notice that we do not need to specify the names or column indexes of the variables containing the sequence in the previously created `actcal.comp` data set. Indeed, `actcal.comp` contains a single string variable, namely the one that stores the compressed sequences.[5]

```
R> actcal.ext <- seqdecomp(actcal.comp)
R> head(actcal.ext)
```

[4] In TraMineR, the default separator is '-', but other user specified separators can be specified.

[5] By default, when no `var` option is specified, the function assumes that the data set contains only sequence data and hence retains all columns, i.e. here the single column of the `actcal.comp` object.

```
    [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12]
[1] "B" "B" "B" "B" "B" "B" "B" "B" "B" "B"  "B"  "B"
[2] "D" "D" "D" "D" "A" "A" "A" "A" "A" "A"  "A"  "D"
[3] "B" "B" "B" "B" "B" "B" "B" "B" "B" "B"  "B"  "B"
[4] "C" "C" "C" "C" "C" "C" "C" "C" "C" "B"  "B"  "B"
[5] "A" "A" "A" "A" "A" "A" "A" "A" "A" "A"  "A"  "A"
[6] "D" "B" "B" "B" "B" "B" "B" "B" "B" "B"  "B"  "B"
```

In the next example taken from Aassve et al. (2007) and introduced in Section 4.1.1, states are coded with character strings of length 3 and separated with the '-' symbol. Each sequence is transformed into a (row) vector, where each element is a state associated with its duration

```
R> seqdecomp(seq.ex1)
```

```
    [1]        [2]        [3]       [4]
[1] "(000,12)" "(0W0,9)"  "(0WU,5)" "(1WU,2)"
[2] "(000,12)" "(0W0,14)" "(1WU,2)" NA
```

To translate compressed sequences with no separator, the sep option can be set to an empty string as in the following example. In that case, every character in the string is assumed to represent a state or event.

```
R> seqdecomp("aaaaaa", sep = "")
```

```
    [1] [2] [3] [4] [5] [6]
[1] "a" "a" "a" "a" "a" "a"
```

5.2.2 The seqformat function

The seqformat function takes as main arguments the name of the sequence data, the names or column indexes of the variables containing the sequences, the input and the output formats. We describe below the various formats that seqformat() can handle. By default, the output returned by the function is in the so called STS compressed format, in which the sequences are stored as character strings. Note that for translating the seqformat() uses the STS format as internal intermediate format. Hence some information can be lost depending on the input and output formats.

Converting to and from the SPS format The seqformat() function allows to convert from and to the state-permanence-sequence SPS format (see Section 4.2.1). In the next example, we translate the sequences contained in the *actcal* data frame to SPS format and store the result in the actcal.SPS object.

```
R> actcal.SPS <- seqformat(actcal, 13:24, from = "STS", to = "SPS")
R> head(actcal.SPS)
```

```
    [1]      [2]      [3]     [4] [5] [6] [7] [8] [9] [10] [11] [12]
[1] "(B,12)" NA       NA      NA  NA  NA  NA  NA  NA  NA   NA   NA
[2] "(D,4)"  "(A,7)"  "(D,1)" NA  NA  NA  NA  NA  NA  NA   NA   NA
[3] "(B,12)" NA       NA      NA  NA  NA  NA  NA  NA  NA   NA   NA
[4] "(C,9)"  "(B,3)"  NA      NA  NA  NA  NA  NA  NA  NA   NA   NA
[5] "(A,12)" NA       NA      NA  NA  NA  NA  NA  NA  NA   NA   NA
[6] "(D,1)"  "(B,11)" NA      NA  NA  NA  NA  NA  NA  NA   NA   NA
```

Here are the same sequences, but in the compressed form

```
R> actcal.SPS.comp <- seqformat(actcal, 13:24, from = "STS", to = "SPS",
+       compressed = TRUE)
R> head(actcal.SPS.comp)
```

```
    Sequence
[1] "(B,12)"
[2] "(D,4)-(A,7)-(D,1)"
[3] "(B,12)"
[4] "(C,9)-(B,3)"
[5] "(A,12)"
[6] "(D,1)-(B,11)"
```

Converting to TSE format In order to extract time stamped events from a sequence of statuses (which is the internal format used by TraMineR), a matrix of size $ns \times ns$ must be given, where ns is the number of distinct states appearing in the sequences. In this matrix, the cell (a, b), where a is the row index and b the column index contains a comma separated list of all events associated with a transition from state a to state b. The diagonal of this matrix has a special meaning. It defines the initial event of the sequence. For example, the position (a, a) gives the event generated when the sequence starts with state a.

The exact design of this matrix can be tricky since a transition may imply several events and the same event may appear in several transitions. However, TraMineR implements several basic generic methods to build this matrix with the function `seqetm()`. You can then adapt the generated matrix to your need by editing the appropriate cells. However, if you create your own matrix from scratch, you should be aware that row and column names of the matrix MUST BE (in a one to one mapping) the states appearing in the data set since they are used to retrieve the events associated with transitions from one state to the other.

The first generic method to generate this matrix is named “transition”. In this case, we simply generate a distinct event for each possible transitions. The diagonal is set to the different possible states.

```
R> data(actcal)
R> actcal.seq <- seqdef(actcal, 13:24, labels = c("FullTime", "PartTime",
+       "LowPartTime", "NoWork"))
R> transition <- seqetm(actcal.seq, method = "transition")
R> transition
```

```
  A                      B                      C
A "FullTime"             "FullTime>PartTime"    "FullTime>LowPartTime"
B "PartTime>FullTime"    "PartTime"             "PartTime>LowPartTime"
C "LowPartTime>FullTime" "LowPartTime>PartTime" "LowPartTime"
D "NoWork>FullTime"      "NoWork>PartTime"      "NoWork>LowPartTime"
  D
A "FullTime>NoWork"
B "PartTime>NoWork"
C "LowPartTime>NoWork"
D "NoWork"
```

The second generic method called “period” generates a “begin” event and an “end” event for each spell. The diagonal gives the “sequence initiating” event that we represent by the first state of the state sequence. By setting `bp=begin`, each initiating event (diagonal element) will be different from the “begin” event that starts other spells in the same state. Here we do not use this option and the same event is used for say starting a full time job at position 1 and at position 4.

```
R> transition <- seqetm(actcal.seq, method = "period")
R> transition
```

```
  A                          B                          C
A "FullTime"                 "endFullTime,PartTime"     "endFullTime,LowPartTime"
B "endPartTime,FullTime"     "PartTime"                 "endPartTime,LowPartTime"
C "endLowPartTime,FullTime"  "endLowPartTime,PartTime"  "LowPartTime"
D "endNoWork,FullTime"       "endNoWork,PartTime"       "endNoWork,LowPartTime"
  D
A "endFullTime,NoWork"
B "endPartTime,NoWork"
C "endLowPartTime,NoWork"
D "NoWork"
```

However, most of the time, we are interested in specific events. For instance, we may be interested in the following events in the in the activity calendar (*actcal* data set).

Table 5.1: Considered events of the activity calendar (*actcal* data set) data set

Code	**Status**
Increase	Increasing activity rate
Decrease	Decreasing activity rate
Start	Starting an activity
Stop	Stopping an activity
FullTime	Starting a full-time paid job (37 hours or more per week)
PartTime	Starting a part-time paid job (19-36 hours per week)
LowPartTime	Starting a part-time paid job (1-18 hours per week)
NoActivity	Starting a period without activity

Table 5.2: Events associated to each state transition

	To state			
From	Full time	Part time	Low part time	No work
state	A	B	C	D
A	FullTime	Decrease PartTime	Decrease LowPartTime	Stop
B	Increase FullTime	PartTime	Decrease LowPartTime	Stop
C	Increase FullTime	Increase PartTime	LowPartTime	Stop
D	Start FullTime	Start PartTime	Start LowPartTime	NoActivity

We may thus define the following matrix: Remember that the events given on the diagonal of this matrix are not associated to the transition from a state to each self, but are just the starting event of the sequence. If we omit this step, information about the beginning of the event sequence will be omitted. In our case, we insert for example the event “FullTime” to each event sequence that begins with the state ”A”.

To generate our own matrix, we first use `seqetm()` to assign correct column and rows names, and then enter the content of our own matrix.

```
R> transition <- seqetm(actcal.seq, method = "transition")
R> transition[1, 1:4] <- c("FullTime", "Decrease,PartTime", "Decrease,LowPartTime",
+      "Stop")
R> transition[2, 1:4] <- c("Increase,FullTime", "PartTime", "Decrease,LowPartTime",
+      "Stop")
R> transition[3, 1:4] <- c("Increase,FullTime", "Increase,PartTime",
+      "LowPartTime", "Stop")
R> transition[4, 1:4] <- c("Start,FullTime", "Start,PartTime", "Start,LowPartTime",
+      "NoActivity")
R> transition
```

```
  A                   B                   C                      D
A "FullTime"          "Decrease,PartTime" "Decrease,LowPartTime" "Stop"
B "Increase,FullTime" "PartTime"          "Decrease,LowPartTime" "Stop"
C "Increase,FullTime" "Increase,PartTime" "LowPartTime"          "Stop"
D "Start,FullTime"    "Start,PartTime"    "Start,LowPartTime"    "NoActivity"
```

Once we have our event matrix, we can convert our state sequence data set into the time stamped event (TSE) form by means of `seqformat()`.

```
R> actcal.tse <- seqformat(actcal[1:100, ], var = 13:24, from = "STS",
+      to = "TSE", tevent = transition)
R> head(actcal.tse)
```

```
  id time      event
1  1    0   PartTime
2  2    0 NoActivity
3  2    4      Start
4  2    4   FullTime
5  2   11       Stop
6  3    0   PartTime
```

Looking at the first record for individual 2 (id number have been created from the sequences order), we see that the events “Start” and “FullTime” occur at time 4, and therefore that individual number 2 started a full time job at time 4. This individual then stops working (“Stop”) at time 11.

Note that the times at which the events occur are computed as the number of positions in the sequences before the new resulting state.

Converting from SPELL format The following command translates the `LA` data set described above (page 48) to the STS state-sequence format. The `from` option of the `seqformat()` function is set to ‘SPELL’. However, since the variable containing the states is here a factor with very long labels, we first create a new variable containing numeric codes only. This is done with the `as.integer()` function, which returns the numeric codes associated with each factor level. We then add this new variable to the `LA` data frame. We also recode missing start times (stored in the `bvla013` variable) that are originally coded as -2 into NA’s.

```
R> levels(LA$bvla100)
```

```
 [1] "other error"
 [2] "filter error"
 [3] "inapplicable"
 [4] "no answer"
 [5] "does not know"
 [6] "with both natural parents"
 [7] "with one parent and his/her new partner"
 [8] "with one parent alone"
 [9] "with relatives or in a foster family"
[10] "with partner (married or not)"
[11] "with friends or in a flat share"
[12] "alone"
[13] "other situation"
[14] "with both natural parents and the partner (married / married"
[15] "with both natural parents and (friends or flat share)"
[16] "with partner (married or not) and (friends or flat share)"
[17] "missing values"
```

```
R> bvla100_rec <- as.integer(LA$bvla100)
R> table(bvla100_rec)
```

```
bvla100_rec
   4    6    7    8    9   10   11   12   13   14   15   16
 110  875   49  148   14 1066   96  393  226   12    2    7
```

```
R> LA <- data.frame(LA, bvla100_rec)
R> LA$bvla013[LA$bvla013 == -2] <- NA
```

We now convert the SPELL data into state sequences. The minimal informations needed for importing data in SPELL format are described in table 5.3. If no options is specified, the input data is supposed to comply this structure. The user can alternatively specify which columns in the input data set contain the mandatory variables using the `id`, `begin`, `end` and `status` option, or select the variables in the required order using the `var` option.

Table 5.3: Structure for the spell format

Position	Variable	Option name
1	Personal identification number	id
2	Start time	begin
3	End time	end
4	Status	status

Other options pertaining to the time axis definition and the handling of overlaps in the beginning and ending times of the successive spells are also available. In the first example below, we import the data with the default options, that is, the time axis is a calendar time axis defined taking the minimum and maximum years at which an episode begins and ends.

```
R> LA.sts <- seqformat(LA, id = "idpers", begin = "bvla013", end = "bvla014",
+       status = "bvla100_rec", from = "SPELL", to = "STS", process = FALSE)
```

The resulting STS data contains the living arrangements from the birth of the respondents to the year of the survey (2002). Hence the time at which the first spell begins is the birth year of the respondent. Since the oldest respondent in our sample was born in 1914, our time axis is defined from 1914 to 2002. The first case was born in 1965, hence the first valid state appears in the column named y1965

```
R> LA.sts[1, ]
```

```
     y1914 y1915 y1916 y1917 y1918 y1919 y1920 y1921 y1922 y1923 y1924 y1925
4101    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
     y1926 y1927 y1928 y1929 y1930 y1931 y1932 y1933 y1934 y1935 y1936 y1937
4101    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
     y1938 y1939 y1940 y1941 y1942 y1943 y1944 y1945 y1946 y1947 y1948 y1949
4101    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
     y1950 y1951 y1952 y1953 y1954 y1955 y1956 y1957 y1958 y1959 y1960 y1961
4101    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA    NA
     y1962 y1963 y1964 y1965 y1966 y1967 y1968 y1969 y1970 y1971 y1972 y1973
4101    NA    NA    NA     6     6     6     6     6     6     6     6     6
     y1974 y1975 y1976 y1977 y1978 y1979 y1980 y1981 y1982 y1983 y1984 y1985
4101     6     6     6     6     6     6     6     6     6     6     6     6
     y1986 y1987 y1988 y1989 y1990 y1991 y1992 y1993 y1994 y1995 y1996 y1997
4101     6     6     6    10    10    10    10    10    10    10    10    10
     y1998 y1999 y2000 y2001 y2002
4101    10    10    10    10    10
```

It is also possible to convert directly into the more concise ‘state-permanence’ format by setting the to option to ‘SPS’. Using the compressed=TRUE option produces compressed sequences (character strings). We can see that the first sequence begins with 51 (1965-1914) missing states (coded as ’*’)

```
R> LA.sps <- seqformat(LA, var = c("idpers", "bvla013", "bvla014",
+       "bvla100_rec"), from = "SPELL", to = "SPS", compressed = TRUE,
+       process = FALSE)
R> head(LA.sps)
```

```
    Sequence
[1] "(*,51)-(6,24)-(10,14)"
[2] "(*,54)-(6,17)-(12,4)-(10,14)"
[3] "(*,47)-(6,17)-(8,5)-(9,1)-(8,1)-(9,14)-(12,3)"
[4] "(*,59)-(6,20)-(10,10)"
[5] "(*,89)"
[6] "(*,11)-(14,28)-(4,50)"
```

Now we convert our data using a process time axis. We need therefore some additional informations, namely the respondents’ birth years, in order to compute the ages at which the spells begin and end. Those informations are provided as a separate data set, containing only one row for each individual. The data contains the respondents’ id as well so as to match the information on birth year with the spell data. In addition to the birth year, it contains the birth month and the sex of each respondent. Here is an extract of this data set

```
R> head(shp.birthyr)
```

```
  idpers birthm birthy sex
1   4101      7   1965   1
2   4102     11   1968   2
3   4103     -3   1991   1
4   4104     -3   1993   1
5   4105     -3   1996   2
6   5101      6   1961   1
```

The pdata option is used to specify the name of the data frame, the pvar option is used to specifiy the names of the columns containing the respondents’ id and birth year.

```
R> LA.sts.process <- seqformat(LA, id = "idpers", begin = "bvla013",
+       end = "bvla014", status = "bvla100_rec", from = "SPELL",
+       to = "STS", process = TRUE, pdata = shp.birthyr, pvar = c("idpers",
+           "birthy"))
```

In the output data, each sequence now begins at age 1. The first sequence shows the living arrangement history of the first respondent. He was in the state 6 (with both natural parents) from his birth to age 23 (the 24th year of life), and then in state 10 (with partner, married or not) from ages 24 to 37 [6]

```
R> LA.sts.process[1, ]
```

```
     a1 a2 a3 a4 a5 a6 a7 a8 a9 a10 a11 a12 a13 a14 a15 a16 a17 a18 a19 a20 a21
4101  6  6  6  6  6  6  6  6  6   6   6   6   6   6   6   6   6   6   6   6   6
     a22 a23 a24 a25 a26 a27 a28 a29 a30 a31 a32 a33 a34 a35 a36 a37 a38 a39
4101   6   6   6  10  10  10  10  10  10  10  10  10  10  10  10  10  10  NA
     a40 a41 a42 a43 a44 a45 a46 a47 a48 a49 a50 a51 a52 a53 a54 a55 a56 a57
4101  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA
     a58 a59 a60 a61 a62 a63 a64 a65 a66 a67 a68 a69 a70 a71 a72 a73 a74 a75
4101  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA
     a76 a77 a78 a79 a80 a81 a82 a83 a84 a85 a86 a87 a88 a89 a90 a91 a92 a93
4101  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA  NA
     a94 a95 a96 a97 a98 a99 a100
4101  NA  NA  NA  NA  NA  NA   NA
```

Here are the same sequences converted in the compressed SPS format

```
R> LA.sps.process <- seqformat(LA, from = "SPELL", to = "SPS", compressed = TRUE,
+       id = "idpers", begin = "bvla013", end = "bvla014", status = "bvla100_rec",
+       process = TRUE, pdata = shp.birthyr, pvar = c("idpers", "birthy"))
R> head(LA.sps.process)
```

```
    Sequence
[1] "(6,24)-(10,14)"
[2] "(6,17)-(12,4)-(10,14)"
[3] "(6,17)-(8,5)-(9,1)-(8,1)-(9,14)-(12,3)"
[4] "(6,20)-(10,10)"
[5] "(*,100)"
[6] "(14,28)-(4,50)"
```

[6] In our example many successive episodes are overlapping since only the start and end years were collected. In this case, with the default `overwrite=TRUE` option, the most recent episode overwrites the older one if they overlap each other.

Chapter 6

Creating state sequence objects

Once your data is imported into R, the next step to work with most of the functions provided by TraMineR is to create an object containing the sequence data. Such objects are created with the `seqdef()` function. This function stores the sequences in the TraMineR internal object type[1] together with some of their attributes.

The `seqdef()` function accepts input data stored in several of the formats described in Chapter 4. The ontology and formats presented in the previous chapter should help the user in identifying the original format of the data he wants to analyse with TraMineR. Some examples showing how to create a sequence object from sequence data in several input formats are provided below.

6.1 Creating a state sequence object

In the example below, we load the *actcal* data set and create a sequence object named 'actcal.seq' with the sequences contained in columns 13 to 24.

```
R> data(actcal)
R> actcal.seq <- seqdef(actcal, var = 13:24)
```

The `var` argument specifying the variables that define the sequences can be a single variable or column index number, a set of variables, or a set of column index numbers. In the next example, the `seqdef()` command is used with the variable names as `var` argument. The `names()` function returns the names of the variables in the data frame and can be used to locate the corresponding column numbers. In the *actcal* data set, the sequences are in the variables "jan00" to "dec00" corresponding to columns 13 to 24.

```
R> names(actcal)
```

```
 [1] "idhous00" "age00"    "educat00" "civsta00" "nbadul00" "nbkid00"
 [7] "aoldki00" "ayouki00" "region00" "com2.00"  "sex"      "birthy"
[13] "jan00"    "feb00"    "mar00"    "apr00"    "may00"    "jun00"
[19] "jul00"    "aug00"    "sep00"    "oct00"    "nov00"    "dec00"
```

Notice that the column names are grouped into a vector with the `c()` function.

```
R> actcal.seq <- seqdef(actcal, var = c("jan00", "feb00", "mar00",
+       "apr00", "may00", "jun00", "jul00", "aug00", "sep00", "oct00",
+       "nov00", "dec00"))
```

[1] The class of this object is 'stslist'.

Using variable names instead of the column index numbers is more secure, because if you delete a variable from the data frame the index numbers can change, while names remain unchanged. One of the attributes stored in the sequence object is the alphabet, i.e. the list of distinct states an individual may be in. In the previous example, the alphabet is taken from the data, that is, we suppose that all possible states appear in the imported sequences. Some options to specify manually the alphabet and other attributes will be described later.

In the *actcal* data set, sequences are in the STS format (see Section 4.2.1), the beloved format used by TraMineR to store data in sequence objects. If your data is already in this format, you can omit the `informat` option because STS is its default value. You just issue the `seqdef()` function and specify the columns containing the sequence data with the `var` option (if your data contain only sequences and no covariate, you can also omit this option).

As discussed in the previous chapter, state sequences may be presented in some non STS format such as SPS for example. Even more, in some cases, sequences are not directly defined as such but can be derived from data originally collected as spells or time stamped events. We describe hereafter the formats that TraMineR can read and convert into a sequence object, using some 'real-life' example data sets. The `informat` option of the `seqdef()` function is used to specify the original format of the input data. Refer to Section 4.2 for identifying the actual format of your data.

6.1.1 Creating a sequence object from SPS-formatted data

In the SPS format (Section 4.2.2), sequences are defined with state-duration couples. The next example shows the content of a text file with such data and some covariates

```
***06 0.896 20 2 0 4 4 M/44 MC/9 SC/91
***07 0.967 20 1 0 4 1 S/66 U/10 M/12 MC/56
***08 0.967 20 1 0 4 4 S/72 U/5 M/67
***10 0.896 20 2 0 4 1 S/10 U/1 UC/133
***27 0.967 20 1 0 4 4 S/54 U/18 S/15 U/11 M/29 MC/17
***30 0.896 20 2 0 4 2 S/10 U/14 M/8 MC/112
```

The first step is to import the text file into an R data frame. We specify that there are no variable names in the first row with the `header=FALSE` option, that the rows may have unequal length with the `fill` option and that empty strings should be treated as missing values with the `na.strings=""` option.

```
R> sweden <- read.table(file = "data/sweden.txt", header = FALSE,
+       sep = " ", na.strings = "", fill = TRUE)
```

The sequence data is contained in columns 8 to 13. Note that sequences are stored in an uneven number of variables, depending on the number of distinct states the individuals passed through.

```
R> head(sweden)
```

```
     V1    V2 V3 V4 V5 V6 V7   V8   V9    V10    V11  V12   V13
1 ***06 0.896 20  2  0  4  4 M/44 MC/9  SC/91   <NA> <NA>  <NA>
2 ***07 0.967 20  1  0  4  1 S/66 U/10   M/12  MC/56 <NA>  <NA>
3 ***08 0.967 20  1  0  4  4 S/72  U/5   M/67   <NA> <NA>  <NA>
4 ***10 0.896 20  2  0  4  1 S/10  U/1 UC/133   <NA> <NA>  <NA>
5 ***27 0.967 20  1  0  4  4 S/54 U/18   S/15   U/11 M/29 MC/17
6 ***30 0.896 20  2  0  4  2 S/10 U/14    M/8 MC/112 <NA>  <NA>
```

Now importing this data into a sequence object is very straightforward. We set the `informat="SPS"` option since the data is in the SPS format. The additional `SPS.in` option, which is passed to the `seqformat` function, is used to specify which characters are surrounding each state/duration couple (here there are no characters) and which character is used as separator between each state and its associated duration (here '/'). The length of the sequences is 144 but here we display the first 30 statuses only in the STS representation.

```
R> sweden.seq <- seqdef(data = sweden, var = 8:13, informat = "SPS",
+       SPS.in = list(xfix = "", sdsep = "/"))
R> sweden.seq[1:6, 1:30]
```

```
    Sequence
[1] M-M-M-M-M-M-M-M-M-M-M-M-M-M-M-M-M-M-M-M-M-M-M-M-M-M-M-M-M-M
[2] S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S
[3] S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S
[4] S-S-S-S-S-S-S-S-S-S-U-UC-UC-UC-UC-UC-UC-UC-UC-UC-UC-UC-UC-UC-UC-UC-UC-UC-UC-UC
[5] S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S-S
[6] S-S-S-S-S-S-S-S-S-S-U-U-U-U-U-U-U-U-U-U-U-U-U-U-M-M-M-M-M-M
```

Here is another example with SPS formatted sequences taken from Aassve et al. (2007). We first create the sequences as character strings and assemble them with the `rbind` function

```
R> seq1 <- "(000,12)-(0W0,9)-(0WU,5)-(1WU,2)"
R> seq2 <- "(000,12)-(0W0,14)-(1WU,2)"
R> seq.ex1 <- rbind(seq1, seq2)
```

The seq.ex1 is just a vector with 2 character strings. Now we turn it into a sequence object using the `seqdef` function

```
R> seq.ex1 <- seqdef(seq.ex1, informat = "SPS")
R> seq.ex1[, 10:25]
```

```
    Sequence
[1] 000-000-000-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0WU-0WU-0WU-0WU
[2] 000-000-000-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0
```

By default, sequence objects are displayed in STS format when typing their name (the `print` method is called with default parameters). At first glance, the two sequences do not seem to be very different. However, the difference shows up clearly when displaying them in the SPS format

```
R> print(seq.ex1, format = "SPS")
```

```
    Sequence
[1] (000,12)-(0W0,9)-(0WU,5)-(1WU,2)
[2] (000,12)-(0W0,14)-(1WU,2)
```

6.1.2 Creating a sequence object from SPELL-formatted data

Data in the SPELL format can be directly converted into a sequence object with the `informat="SPELL"` option. The required data structure and options for importing spell data are described in more detail in section 5.2.2. The same SPELL data extracted from the Swiss Household Panel retrospective survey will be used here as an example. The original data containing living arrangement history (see Table 4.3 on page 31 for the state description) has been imported into R (see Section 5.1.1). The living arrangement histories for the first two individuals (id = 2713 and 2714) are displayed below

```
R> LA[1:9, ]
```

```
  idpers bvla_idx bvla013 bvla014                      bvla100
1   2713        1    1965    1989     with both natural parents
2   2713        2    1989    1990 with partner (married or not)
3   2713        3    1990    1991 with partner (married or not)
4   2713        4    1991    2002 with partner (married or not)
5   2714        1    1968    1985     with both natural parents
6   2714        2    1985    1988                         alone
7   2714        3    1989    1990 with partner (married or not)
8   2714        4    1990    1991 with partner (married or not)
9   2714        5    1991    2002 with partner (married or not)
```

The variables needed to create the sequence object are an identification number to group all rows pertaining to the same individual (idpers), a starting and ending time for the spells (bvla013 and bvla014), and a status variable (bvla100). Note that the statuses in variable 'bvla100' appear as labels since this variable was imported as a factor from the SPSS data file.

```
R> seqstatl(LA$bvla100)
```

```
 [1] "alone"
 [2] "other situation"
 [3] "with both natural parents"
 [4] "with both natural parents and (friends or flat share)"
 [5] "with both natural parents and the partner (married / married"
 [6] "with friends or in a flat share"
 [7] "with one parent alone"
 [8] "with one parent and his/her new partner"
 [9] "with partner (married or not)"
[10] "with partner (married or not) and (friends or flat share)"
[11] "with relatives or in a foster family"
```

But for more convenience we want shorter codes for the statuses when displaying the sequences. Hence we attribute numeric codes instead of the labels with the `states` option. The original labels are preserved and used as legends for the states which will appear in the graphics (`labels` option). The `process` option, which is passed to the `seqformat` function, is set to `FALSE`, that is, the time axis for the sequences is defined as a calendar time axis whose start and end are the minimum and maximum values found in the `begin` (bvla013) and `end` (bvla013) columns of the input data set

```
R> LA.labels <- seqstatl(LA$bvla100)
R> LA.states <- 1:length(LA.labels)
R> LA.seq <- seqdef(LA, var = c("idpers", "bvla013", "bvla014",
+       "bvla100"), informat = "SPELL", states = LA.states, labels = LA.labels,
+       process = FALSE)
```

Now we can display (in SPS format) the resulting sequences. By setting the `process` option to `FALSE` the sequences have been created using a calendar time axis (see 4.1.3 and 5.2.2), ranging from 1914 to 2002. Hence most of the sequences begin with missing states

```
R> print(LA.seq[1:15, ], format = "SPS")
```

```
     Sequence
[1]  (*,51)-(3,24)-(9,14)
[2]  (*,54)-(3,17)-(1,4)-(9,14)
[3]  (*,47)-(3,17)-(7,5)-(11,1)-(7,1)-(11,14)-(1,3)
[4]  (*,59)-(3,20)-(9,10)
[5]  (*,89)
[6]  (*,11)-(5,28)
```

```
[7]  (*,57)-(3,18)-(1,4)-(9,5)-(1,2)-(9,3)
[8]  (*,67)-(3,5)-(8,11)-(7,3)-(9,3)
[9]  (*,51)-(3,22)-(1,4)-(9,5)-(1,7)
[10] (*,5)-(3,20)-(9,59)-(6,5)
[11] (*,59)-(3,24)-(9,6)
[12] (*,61)-(3,22)-(9,6)
[13] (*,53)-(3,24)-(9,12)
[14] (*,43)-(3,21)-(9,25)
[15] (*,45)-(3,17)-(1,9)-(9,18)
```

If we want the sequences to be defined on a process time axis, we need to provide an additional data file containing the birth years of the respondents, as described before (see 5.2.2).

```
R> head(shp.birthyr)

  idpers birthm birthy sex
1   2713      7   1965   1
2   2714     11   1968   2
3   2715     -3   1991   1
4   2716     -3   1993   1
5   2717     -3   1996   2
6   3713      6   1961   1
```

The sequences are created by using the `pdata` and `pvar` options

```
R> LA.seq <- seqdef(LA, var = c("idpers", "bvla013", "bvla014",
+        "bvla100"), informat = "SPELL", states = LA.states, labels = LA.labels,
+        process = TRUE, pdata = shp.birthyr, pvar = c("idpers", "birthy"))
```

Now the sequence for the first individual in the data begins at age 1 (he was born in 1965 and his living arrangement history begins in 1965). He has been in state 3 (with both natural parents) during 24 years and then in state 9 (with a partner) during 13 years.

```
R> LA.seq[1:2, ]

     Sequence
2713 3-3-3-3-3-3-3-3-3-3-3-3-3-3-3-3-3-3-3-3-3-3-3-3-9-9-9-9-9-9-9-9-9-9-9-9-9
2714 3-3-3-3-3-3-3-3-3-3-3-3-3-3-3-3-1-1-1-1-9-9-9-9-9-9-9-9-9-9-9-9-9-9
```

6.2 Attributes of sequence objects

When creating a sequence object with the `seqdef()` function, several attributes are stored together with the sequence data, namely:

- the alphabet
- the color palette used for representing states in plots
- optional state labels
- the code used for missing values
- the starting time of the sequences

Those attributes are retrieved by other TraMineR functions, for instance the alphabet, color palette and state labels associated to the object are used by the TraMineR sequence plotting functions[2]. If no values for the attributes are provided by the user, those are set automatically. The default values and the user-available options to override them are described below.

[2] The color palette and state labels can be overridden by options to the plotting functions

6.2.1 State codes

In a sequence object, the variables (columns) where the states composing the sequences are stored are R factors. A R factor has an internal numeric code and a label. It resembles the numerically coded variables with value labels we found in SPSS or Stata. When importing data from statistical softwares such as SPSS or Stata all variables with value labels are converted into R factors unless you specify it otherwise. When creating a sequence object, if you do not specify yourself the list of possible states, TraMineR uses the factor levels (i.e. the value labels) to create the alphabet. To illustrate we go back to our SPELL data set described in Section 6.1.2. If we create a sequence object using the state labels present in the data, it would look like this

```
R> print(LA.B.seq[1, ], format = "SPS")
```

```
    Sequence
[1] (with both natural parents,24)-(with partner (married or not),14)
```

The alphabet would be made of the factor levels

```
R> alphabet(LA.B.seq)
```

```
 [1] "alone"
 [2] "other situation"
 [3] "with both natural parents"
 [4] "with both natural parents and (friends or flat share)"
 [5] "with both natural parents and the partner (married / married"
 [6] "with friends or in a flat share"
 [7] "with one parent alone"
 [8] "with one parent and his/her new partner"
 [9] "with partner (married or not)"
[10] "with partner (married or not) and (friends or flat share)"
[11] "with relatives or in a foster family"
```

Hence, if states in the original data set are represented by labels, it may be useful to change the state labels to shorter symbols (in the plots, one can still optionally specify a more descriptive legend of the represented states). This can be done when creating the sequence object with the `states` option. When creating the *La.seq* sequence object, we specified the `states=1:12` option to code the states as numbers ranging from 1 to 12. The sequence object is much more readable when it is displayed

```
R> print(LA.seq[1, ], format = "SPS")
```

```
    Sequence
[1] (3,24)-(9,14)
```

```
R> alphabet(LA.seq)
```

```
 [1]  1  2  3  4  5  6  7  8  9 10 11
```

6.2.2 Alphabet

If you create a sequence object without specifying the alphabet option, all possible states are supposed to be present in the data set and the alphabet is set by listing the distinct states encountered. However, in some cases, we may have to consider states that are not present in the data set used to create the sequence object. Suppose for instance that you want to compare two sequence data sets and that there are some states in one data set that are not present in the other one. Without explicitly specifying the list of the possible states with the alphabet option when creating the sequence objects from these data sets, the missing states will not be accounted for, which may produce misleading results when comparing tabulation of the state frequency of the two data sets. The colors attributed to the states will also be different for each data set which may also be source of confusion. Let us take a short example to illustrate this point. We create two sequence objects, one with the first three sequences of the *actcal* data set

```
R> actcal.s1 <- seqdef(actcal[1:3, 13:24])
R> alphabet(actcal.s1)

[1] "A" "B" "D"
```

and one with sequences 7 to 9.

```
R> actcal.s2 <- seqdef(actcal[7:9, 13:24])
R> alphabet(actcal.s2)

[1] "A" "D"
```

In the first example, the alphabet is set to (A,B,D) while in the second object it is set to (A,D). Since we know that the possible states are (A,B,C,D), we specify manually the alphabet for the first ...

```
R> actcal.s1 <- seqdef(actcal[1:3, 13:24], alphabet = c("A", "B",
+       "C", "D"))
R> alphabet(actcal.s1)

[1] "A" "B" "C" "D"
```

... and the second object

```
R> actcal.s2 <- seqdef(actcal[7:9, 13:24], alphabet = c("A", "B",
+       "C", "D"))
R> alphabet(actcal.s2)

[1] "A" "B" "C" "D"
```

which permits to directly compare plots and tabulations of each sequence object. However, if we had selected the two objects as subsets of the *actcal.seq* sequence object they would have inherited the same alphabet

```
R> actcal.seq <- seqdef(actcal, 13:24)
R> actcal.s1 <- actcal.seq[1:3, ]
R> actcal.s2 <- actcal.seq[7:9, ]
R> alphabet(actcal.s1)

[1] "A" "B" "C" "D"

R> alphabet(actcal.s2)

[1] "A" "B" "C" "D"
```

6.2.3 Color palette

The color palette attached to a sequence object is used by default in the graphical functions provided by TraMineR. If no optional argument is provided, a color palette is created with the dedicated RColorBrewer R package, which is loaded at start-up by TraMineR. The default color palette is `Accent`. It can be overridden by the user with the `cpal` option. The awaited argument is a character vector containing a color for each state in the alphabet. The `colors()` function provides the list of color names available in R.

```
R> actcal.seq <- seqdef(actcal, 13:24, cpal = c("red", "blue", "green",
+       "yellow"))
```

The color palette for an existing sequence object may be modified by providing a vector with color names ...

```
R> attr(actcal.seq, "cpal") <- c("pink", "purple", "cyan", "yellow")
```

... or by retrieving the colors from a color palette. In the example below, we retrieve 4 colors from the “Dark2” color palette provided by the RColorBrewer package.

```
R> attr(actcal.seq, "cpal") <- brewer.pal(4, "Dark2")
```

6.2.4 State labels

State labels are used as legends by the TraMineR plot functions. If not specified, labels are set with the state codes. Use the `labels` option to define state labels. The `labels` option expects a vector containing a character string for each state in the alphabet. The order of the labels in the vector must match the order of the states as returned by the `seqstatl` function [3] .

```
R> actcal.seq <- seqdef(actcal, 13:24, labels = c("> 37 hours",
+       "19-36 hours", "1-18 hours", "no work"))
```

6.2.5 Starting time

The `start` option specifies the starting time of the sequences. Since the value yields for all the sequences in the data, this information makes sense only when states are dated and when all sequences have the same starting time (are left aligned). Otherwise, you can safely ignore this option and the value will be set to 1. This attribute is used for instance for creating column names of the sequence object when there are no column names in the input data and no names are provided by the user. This attribute is also updated when selecting subscripts of a sequence object (see Section 6.4).

6.3 Summarizing sequence objects

The generic R `summary()` function will display some information when the name of a sequence object is given as its argument.

```
R> summary(actcal.seq)
```

[3] IMPORTANT: the order of the states returned by the `seqstatl()` function may not be the same on Mac OS-X systems as on Linux and Windows systems.

```
[>] sequence object created with TraMineR version 1.8
[>] 2000 sequences in the data set, 186 unique
[>] min/max sequence length: 12/12
[>] alphabet (state labels):
    1=A (> 37 hours)
    2=B (19-36 hours)
    3=C (1-18 hours)
    4=D (no work)
[>] dimensionality of the sequence space: 36
[>] colors: 1=#7FC97F 2=#BEAED4 3=#FDC086 4=#FFFF99
```

The dimensionality is the number of dimensions necessary for constructing the sequence space Haubold and Wiehe (2006), i.e.,

$$d = (|A| - 1)\ell$$

where $|A|$ is the size of the alphabet and ℓ the (maximal) length of the sequences.

6.4 Indexing and printing sequence objects

Displaying a sequence object is as simple as typing its name. However, displaying a sequence object containing 2000 rows such as `actcal.seq` for instance is not very interesting. Subscripts can be used to display only selected rows and/or columns of the data. Subscripts and indexes work the same way as for matrices and data frames.

In the next example, we display only the first 5 sequences and columns 3 to 8 (March to August) of the previously created `actcal.seq` sequence object. Typing a sequence object name, with or without subscripts, is equivalent to issuing the `print()` command with the object name as argument

```
R> actcal.seq[1:5, 3:8]
```

```
     Sequence
2848 B-B-B-B-B-B
1230 D-D-A-A-A-A
2468 B-B-B-B-B-B
654  C-C-C-C-C-C
6946 A-A-A-A-A-A
```

Note that the sequences are displayed in a compressed format, i.e. as character strings where the states are separated with the '-' symbol. But internally, each state is still stored in a single variable, as shown with the `print` command with the '`extended=TRUE`' option

```
R> print(actcal.seq[1:5, 3:8], ext = TRUE)
```

```
     mar00 apr00 may00 jun00 jul00 aug00
2848     B     B     B     B     B     B
1230     D     D     A     A     A     A
2468     B     B     B     B     B     B
654      C     C     C     C     C     C
6946     A     A     A     A     A     A
```

We get a more concise view of sequences with the SPS state-permanence representation. Obviously, the SPS format yields shorter and more readable sequences. We obtain the SPS representation with the `format="SPS"` option

```
R> print(actcal.seq[1:5, 3:8], format = "SPS")
```

```
    Sequence
[1] (B,6)
[2] (D,2)-(A,4)
[3] (B,6)
[4] (C,6)
[5] (A,6)
```

When using subscripts to select only parts of sequence objects, the result is still a sequence object and all attributes of the parent object are preserved (inherited). As an example, the sequences for the summer months only are selected from the previously created *actcal.seq* sequence object. We see that the color palette (`cpal` attribute) and state labels (`label` attribute) have been preserved, while the `start` attribute originally set to 1 (default value) has been updated to 6.

```
R> actcal.summer <- actcal.seq[, 6:9]
R> attr(actcal.summer, "cpal")
```

```
[1] "#7FC97F" "#BEAED4" "#FDC086" "#FFFF99"
```

```
R> attr(actcal.summer, "labels")
```

```
[1] "> 37 hours"  "19-36 hours" "1-18 hours"  "no work"
```

```
R> attr(actcal.summer, "start")
```

```
[1] 6
```

The column names are retrieved with the `names` function

```
R> names(actcal.summer)
```

```
[1] "jun00" "jul00" "aug00" "sep00"
```

6.5 Truncations, gaps and missing values

The handling of truncations, gaps and missing values in sequence data received only little attention the literature. In this section we present the effort made to consider this topic and the available features in TraMineR.

6.5.1 Introduction

To outline how we can handle missing values, truncations and gaps in sequences with TraMineR, we focus on sequences stored in the extended STS format (see Chapter 4). This is one of the most common way of storing sequences that TraMineR users may encounter and also the internal storage format for sequence objects in TraMineR. Each sequence is stored in a row of a rectangular matrix, and each row has the same number of elements. However, for several reasons, sequences in a data set may have different lengths or may not begin and end at the same column positions in the matrix. For example:

- Sequences defined as the list of successive states without duration information are typically of varying length.

- In event sequences, the number of events experienced by each individual differs from one individual to the other.
- The length of the follow up is not the same for all individuals or sequences may be right or left censored.
- Sequences may not be left aligned depending on the time axis on which they are defined.
- Data may not be available for all measuring points yielding internal gaps in the sequences.

We consider the *famform* data set coming with TraMineR that contains sequences with unequal lengths. Indeed, the sequences contain only the distinct states that the individuals passed through. The sequences are recorded in the compressed format, i.e. as character strings.

```
R> data(famform)
R> famform
```

```
     Sequence
[1,] "S-U"
[2,] "S-U-M"
[3,] "S-U-M-MC"
[4,] "S-U-M-MC-SC"
[5,] "U-M-MC"
```

When translating the *famform* data set into the extended STS format (where sequences are stored in a matrix), missing values are generated to fill the empty rows

```
R> seqdecomp(famform)
```

```
    [1] [2] [3]  [4]  [5]
[1] "S" "U" NA   NA   NA
[2] "S" "U" "M"  NA   NA
[3] "S" "U" "M"  "MC" NA
[4] "S" "U" "M"  "MC" "SC"
[5] "U" "M" "MC" NA   NA
```

Varying lengths of follow up. Censored data.

Time axis The discrete time axis on which the sequences are defined can be a **calendar** time axis or a **process** time axis (see 4.1.3). A calendar time axis does not have a natural origin and fixing an origin is simply a convention for providing time points. On a process time axis, the origin represents the date of a starting event. Suppose that we follow up respondents after they experienced some event (an accident, ending education, ...) and information is collected during 10 years. If the respondents entered the study at different points in time and we represent the data on a calendar time axis, the data could look like this

```
R> ex2.cal
```

```
   1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001
s1    1    1    1    1    1    1    1    3    3    3   NA   NA
s2   NA    2    2    2    2    2    2    2    3    3    3   NA
s3   NA   NA    1    1    1    2    2    2    2    2    2    2
```

Table 6.1: Start and end of the sequences in the *ex1* data set

Sequence	Start position	End position	Gap positions
s1	4	13	
s2	1	10	
s3	2	11	
s4	1	10	(3,4)
s5	1	10	(2,7)
s6	1	10	(1,2,3)

The meaning of the states does not matter here. Respondent 1 entered the study in 1990 and was followed up until 1999, while respondent 3 entered in 1992 and was followed up until 2001. In this case it may be more appropriate to represent the data on a process time axis where all sequences would be left aligned, meaning that their common origin is not a specific year but the beginning of the observed 10 year spell.

```
R> ex2.proc

   T1 T2 T3 T4 T5 T6 T7 T8 T9 T10
s1  1  1  1  1  1  1  1  3  3   3
s2  2  2  2  2  2  2  2  3  3   3
s3  1  1  1  2  2  2  2  2  2   2
```

Internal gaps. Sequences may also contain “gaps”, i.e. some unknown statuses inside the sequence due to non-response or other reasons.

6.5.2 Handling the different kinds of missing values

From the above discussion, we may distinguish three types of elements in the matricial representation of sequence data: 1) statuses composing the sequence, 2) missing values and 3) empty cells used for adjustment when the sequence is shorter than the row length. To illustrate we load the example data frame *ex1*. In this example all elements that are not valid statuses are coded as NA’s, the usual way of representing missing values in R. Hence we do not distinguish between missing values and empty cells.

```
R> ex1

    [1]  [2]  [3]  [4] [5] [6]  [7] [8] [9] [10] [11] [12] [13]
s1 <NA> <NA> <NA>    A   A   A    A   A   A    A    A    A    A
s2    D    D    D    B   B   B    B   B   B    B <NA> <NA> <NA>
s3 <NA>    D    D    D   D   D    D   D   D    D    D <NA> <NA>
s4    A    A <NA> <NA>   B   B    B   B   D    D <NA> <NA> <NA>
s5    A <NA>    A    A   A   A <NA>   A   A    A <NA> <NA> <NA>
s6 <NA> <NA> <NA>    C   C   C    C   C   C    C <NA> <NA> <NA>
```

The sequences are stored in a 13 columns matrix but we state that the real length of each of them is in actually 10. The positions at which each sequence starts and ends are summarized in Table 6.1. Some sequences also contain gaps corresponding to unknown states. Now the question is how are those sequences handled by TraMineR when we create a sequence object and later on when computing distances between sequences. And which control do we have on this process. To describe this we divide the sequence in three distinct parts and define three associated vectors with the indexes of the missing values in each of the three parts.

Table 6.2: Indexes of missing values in the three parts of the sequences

Sequence	vl	vg	vr
s1	$(1, 2, 3)$	$(\emptyset)$	$(\emptyset)$
s2	$(\emptyset)$	$(\emptyset)$	$(11, 12, 13)$
s3	(1)	$(\emptyset)$	$(12, 13)$
s4	$(\emptyset)$	$(3, 4)$	$(11, 12, 13)$
s5	$(\emptyset)$	$(2, 7)$	$(11, 12, 13)$
s6	$(1, 2, 3)$	$(\emptyset)$	$(11, 12, 13)$

- The first part of the sequence is made of the missing values appearing before the first (leftmost) valid state element. This part can be void if the sequence begins with a valid state. The associated vector vl contains the indexes of all missing values appearing before the first (leftmost) valid state in a sequence. Hence $vl = (1, 2, 3)$ for $s1$ and $s6$, $vl = (\emptyset)$ for $s2$, $s4$ and $s5$ and $vl = (1)$ for $s3$. Settings for handling missing values in this part of the sequence are defined with the `left` option.

- The second part of the sequence begins with the first (leftmost) valid state and ends with the last (rightmost) valid state. The associated vector vg contains the indexes of all missing values appearing in this part of the sequence. Hence $vg = (3, 4)$ in $s4$, $vg = (2, 7)$ in $s5$ and $vg = (\emptyset)$ in $s1$, $s2$, $s3$ and $s6$. Settings for handling missing values in this part of the sequence are defined with the `gaps` argument.

- The third part of the sequence is made of the missing values appearing after the last (rightmost) valid state (element). The associated vector vr contains the indexes of all missing values appearing in this part of the sequence. Hence $vr = (11, 12, 13)$ for $s2$, $s4$, $s5$ and $s6$, $vr = (12, 13)$ for $s3$ and $vr = (\emptyset)$ for $s1$. Settings for handling missing values in this part of the sequence are defined with the `right` option.

When defining a sequence object, the user can specify the way he wants to handle the elements indexed by each of these three vectors. The options for each part are set with the arguments `left`, `gaps` and `right`. Each of them accepts the following values

- `"DEL"` for deleting the NA's, meaning that they do not belong to the sequence. Missing values become thus void. When necessary for maintaining the row length, a special character ("%" by default) will be inserted on the right of the sequence for each such deleted missing value.

- `NA`, nothing is done and the each missing value is left as an explicit missing element. For the output, missing values are coded with a special character ("*" by default) that is more convenient than NA for displaying the sequences.

Default values are `left=NA`, `gaps=NA` and `right="DEL"`, We demonstrate how the different options work on our example data. First, we leave the default settings unchanged. With those settings, all missing values encountered after the last valid state[4] in a sequence are considered as void elements, i.e. the sequence is considered as ending after the last valid state[5].

```
R> ex1.A <- seqdef(ex1, 1:13)
R> ex1.A
```

[4] 'last' means the rightmost.

[5] The code used in the input data for missing values can be set with the `missing` option. The default is NA, the usual way of representing missing values in R.

```
   Sequence
s1 *-*-*-A-A-A-A-A-A-A-A-A-A
s2 D-D-D-B-B-B-B-B-B-B
s3 *-D-D-D-D-D-D-D-D-D-D
s4 A-A-*-*-B-B-B-B-D-D
s5 A-*-A-A-A-A-*-A-A-A
s6 *-*-*-C-C-C-C-C-C-C
```

By printing the sequence object in its internal matrix representation, we see that all the end trailing positions are occupied by "%", the default TraMineR character code for void elements.

```
R> print(ex1.A, ext = TRUE)
```

```
   [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13]
s1  *   *   *   A   A   A   A   A   A    A    A    A    A
s2  D   D   D   B   B   B   B   B   B    B    %    %    %
s3  *   D   D   D   D   D   D   D   D    D    D    %    %
s4  A   A   *   *   B   B   B   B   D    D    %    %    %
s5  A   *   A   A   A   A   *   A   A    A    %    %    %
s6  *   *   *   C   C   C   C   C   C    C    %    %    %
```

Void elements are for instance not taken into account when computing the length of the sequences.

```
R> seqlength(ex1.A)
```

```
   Length
s1     13
s2     10
s3     11
s4     10
s5     10
s6     10
```

Nonetheless, the computed length is not 10 for all sequences. The left part of sequences $s1$ and $s3$ which do not begin in the first column of the matrix, has been considered as part of them. To remedy to this problem, we could use the `left="DEL"` option. With this option, all missing values at the beginning of a sequence are considered as void elements and the sequence is shifted to the left so that it begins with its first valid status.

```
R> ex1.B <- seqdef(ex1, left = "DEL")
R> ex1.B
```

```
   Sequence
s1 A-A-A-A-A-A-A-A-A-A
s2 D-D-D-B-B-B-B-B-B-B
s3 D-D-D-D-D-D-D-D-D-D
s4 A-A-*-*-B-B-B-B-D-D
s5 A-*-A-A-A-A-*-A-A-A
s6 C-C-C-C-C-C-C
```

To preserve the number of elements of the rows in the matrix, void elements are added at the end (right side) of the sequence.

```
R> print(ex1.B, ext = TRUE)
```

```
   [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13]
s1   A   A   A   A   A   A   A   A   A    A    %    %    %
s2   D   D   D   B   B   B   B   B   B    B    %    %    %
s3   D   D   D   D   D   D   D   D   D    D    %    %    %
s4   A   A   *   *   B   B   B   B   D    D    %    %    %
s5   A   *   A   A   A   A   *   A   A    A    %    %    %
s6   C   C   C   C   C   C   C   %   %    %    %    %    %
```

Now the lengths of all sequences appears to be 10, except for $s6$. This is due to the fact that $s6$ begins with missing values that are indeed part of the sequence, which have been deleted. We can see that it is not possible to disentangle in that case the ”void” and the ”real” missing values at the begining of a sequence.

```
R> seqlength(ex1.B)
```

```
   Length
s1     10
s2     10
s3     10
s4     10
s5     10
s6      7
```

The same options are available for the gaps in a sequence, as previously defined. In the next example we ask to delete also the missing values encountered in the center part of the sequence. Sequences have been reduced to their valid statuses only.

```
R> ex1.C <- seqdef(ex1, left = "DEL", gaps = "DEL")
R> ex1.C
```

```
   Sequence
s1 A-A-A-A-A-A-A-A-A-A
s2 D-D-D-B-B-B-B-B-B-B
s3 D-D-D-D-D-D-D-D-D-D
s4 A-A-B-B-B-B-D-D
s5 A-A-A-A-A-A-A-A
s6 C-C-C-C-C-C-C
```

```
R> seqlength(ex1.C)
```

```
   Length
s1     10
s2     10
s3     10
s4      8
s5      8
s6      7
```

Now we set all options to NA and all missing values are considered as part of the sequences

```
R> ex1.D <- seqdef(ex1, left = NA, gaps = NA, right = NA)
R> ex1.D
```

```
   Sequence
s1 *-*-*-A-A-A-A-A-A-A-A-A-A
s2 D-D-D-B-B-B-B-B-B-B-*-*-*
s3 *-D-D-D-D-D-D-D-D-D-D-*-*
s4 A-A-*-*-B-B-B-B-D-D-*-*-*
s5 A-*-A-A-A-A-*-A-A-A-*-*-*
s6 *-*-*-C-C-C-C-C-C-C-*-*-*
```

```
R> seqlength(ex1.D)
```

```
   Length
s1     13
s2     13
s3     13
s4     13
s5     13
s6     13
```

Here is the SPS representation of the previous example

```
R> print(ex1.D, format = "SPS")
```

```
    Sequence
[1] (*,3)-(A,10)
[2] (D,3)-(B,7)-(*,3)
[3] (*,1)-(D,10)-(*,2)
[4] (A,2)-(*,2)-(B,4)-(D,2)-(*,3)
[5] (A,1)-(*,1)-(A,4)-(*,1)-(A,3)-(*,3)
[6] (*,3)-(C,7)-(*,3)
```

Chapter 7

Describing and visualizing state sequences

Results shown in this chapter are obtained with:
TraMineR version 1.8
R version 2.9.2 (2009-08-24)-platform: i486-pc-linux-gnu.

This chapter presents the main TraMineR tools for describing and visualizing sequences. We first briefly explain in Section 7.1 the general plotting philosophy adopted in TraMineR. Section 7.2 presents then tools for describing and visualizing set properties of the sequences from an aggregated standpoint and Section 7.3 focuses on the characterization of individual sequence properties and their summary.

7.1 General principle of TraMineR sequence plots

TraMineR provides three basic plotting functions for visualizing sequence characteristics: `seqdplot()` for plotting the state distribution at each time point, `seqfplot()` for plotting the frequencies of the most frequent sequences and `seqiplot()` for plotting all or a selection of individual sequences.

7.1.1 Color palette representing the states

The before-mentioned plot functions have in common to use a specific color for each state. The choice of the colors is done by selecting a color palette. Indeed, for facilitating readability it is important to use the same color palette for all plots based on a same alphabet. The philosophy retained in TraMineR is therefore to attach the alphabet and the color palette as attributes of the sequence object (see Section 6.2) and letting the plotting functions retrieve these attributes when generating the plots. The same is true also for the labels of the time axis ticks and the labels of the states.

7.1.2 Plotting the legend separately

To be understandable, a plot must be accompanied by the legend of the used state colors. By default each sequence plot produces therefore the legend on the top of the graphic using the attributes of the plotted sequence object.

In some cases, especially when you generate multiple plots, for instance a state distribution plot and an sequence frequency plot, it may be preferable to generate plots without legends and produce the legend only once separately. For doing so, TraMineR provides the `seqlegend()` function that generates the legend has a separate graphic, and a `withlegend=FALSE` option for the `seqdplot()`, `seqfplot` and `seqiplot()` functions.

For example, the following code generates three plots and a legend side by side as shown in Figure 7.1.

```
R> par(mfrow = c(2, 2))
R> seqiplot(biofam.seq, title = "Index plot (first 10 sequences)",
+       withlegend = FALSE)
R> seqdplot(biofam.seq, title = "State distribution plot", withlegend = FALSE)
R> seqfplot(biofam.seq, title = "Sequence frequency plot", withlegend = FALSE,
+       pbarw = TRUE)
R> seqlegend(biofam.seq)
```

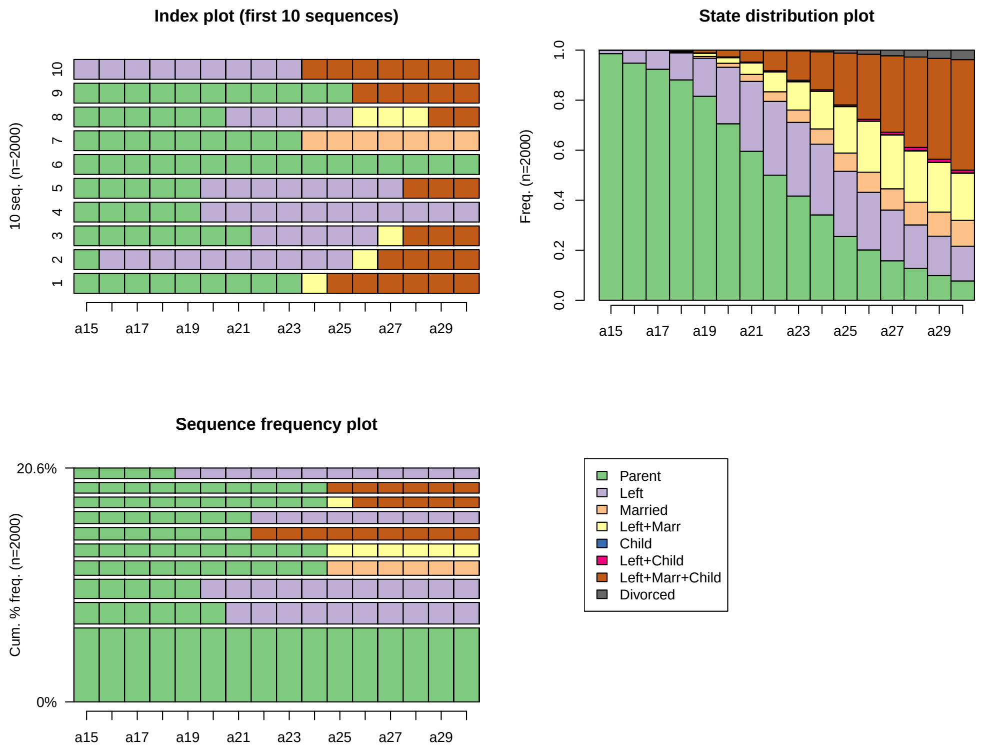


Figure 7.1: Legend plotted as an additional graphic

7.2 Describing and visualizing sequence data sets

In this section we present functions for visualizing and describing sequences at the aggregate level.

7.2.1 List of states present in sequence data

A first result we may want is just the list of states present in the data set. This is obtained with the `alphabet()` function when the list of states has not been explicitly specified by the user.[1] The `alphabet()` function returns the list of the possible states for a sequence object. In the following example, we see that the alphabet for the *actcal.seq* sequence object contains 4 distinct states: A, B, C and D (see Table 3.1 page 21 for their description).

```
R> data(actcal)
R> actcal.lab <- c("> 37 hours", "19-36 hours", "1-18 hours", "no work")
R> actcal.seq <- seqdef(actcal, 13:24, labels = actcal.lab)
R> alphabet(actcal.seq)
```

```
[1] "A" "B" "C" "D"
```

To get the list of all distinct states appearing in a data set containing sequences not converted into a sequence object, use the `seqstatl()` function. You tell `seqstatl()` which variables define the sequence data by providing with the `var` argument either their names or their column index numbers. For specifying the columns by their names, you have to group them into a vector with the `c()` function. By default, the `seqstatl()` function expects a STS formatted data set as input. If the sequences in your data are in another format, you should specify it with the `format` option. In the following example, we retrieve the alphabet for the two sequences of the sp.ex1[2] data set.

```
R> sp.ex1 <- rbind("(000,12)-(0W0,9)-(0WU,5)-(1WU,2)", "(000,12)-(0W0,14)-(1WU,2)")
R> sp.ex1
```

```
     [,1]
[1,] "(000,12)-(0W0,9)-(0WU,5)-(1WU,2)"
[2,] "(000,12)-(0W0,14)-(1WU,2)"
```

```
R> seqstatl(sp.ex1, format = "SPS")
```

```
[1] "000" "0W0" "0WU" "1WU"
```

7.2.2 State distribution

State distribution plot The `seqdplot()` function plots a graphic showing the state distribution at each time point (the columns of the sequence object). The state distribution itself is obtained with the `seqstatd()` command described below. In the next example we plot the state distribution for the *mvad* data set. We first define a `mvad.labels` vector of state labels to be used for the legend of the colors used in the plot. This vector has six elements since there are six different states in the alphabet. The `xtstep` option sets the step between displayed tick-marks and labels on the x-axis of state sequence plots. This parameter is retrieved from the sequence object by the plotting functions but can be overriden by the user.

```
R> data(mvad)
R> mvad.labels <- c("employment", "further education", "higher education",
+       "joblessness", "school", "training")
R> mvad.seq <- seqdef(mvad, 15:86, labels = mvad.labels, xtstep = 6)
```

The plot is produced with the following command

```
R> seqdplot(mvad.seq, border = NA)
```

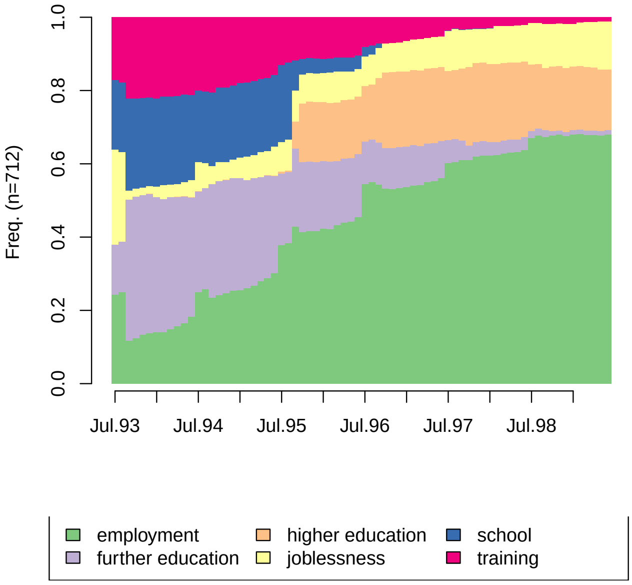


Figure 7.2: Distribution of the statuses by age in the *mvad* data set

The resulting graphic is shown in Figure 7.2. The proportion of individuals who are employed increases to become the most frequent state at the end of the follow-up.

The state distribution plot for the *actcal* data, obtained with the command below, shows a different pattern (Figure 7.3). The distribution of the work statuses looks very stable over time.

```
R> seqdplot(actcal.seq)
```

State distribution table Beside plotting the distribution of the states at each time point, you may want to get the figures of the distribution. The `seqstatd()` function returns the table of the state distributions together with the number of valid states and an entropy measure for each time unit. The state distributions are those visualized by the `seqdplot()`function. The following example shows the family formation state distribution from age 15 to 30 in the (*biofam* data set (see Table 3.3 page 22 for the description of the states). The first argument to the `seqstatd()` function is the previously created `biofam.seq` sequence object.

```
R> seqstatd(biofam.seq)
```

```
 [State frequencies]
    a15   a16    a17    a18    a19   a20    a21    a22    a23    a24   a25
0 0.986 0.948 0.9235 0.8810 0.8155 0.706 0.5955 0.5000 0.4165 0.3410 0.255
1 0.014 0.052 0.0750 0.1080 0.1520 0.226 0.2795 0.2950 0.2945 0.2830 0.261
2 0.000 0.000 0.0005 0.0030 0.0070 0.016 0.0280 0.0385 0.0495 0.0610 0.073
3 0.000 0.000 0.0005 0.0025 0.0130 0.022 0.0460 0.0790 0.1130 0.1505 0.185
```

[1] in that case, some states in the alphabet may not appear in the data. See Sec. 6 for more information on this topic

[2] this data set is created by binding two character strings with the `rbind()` function

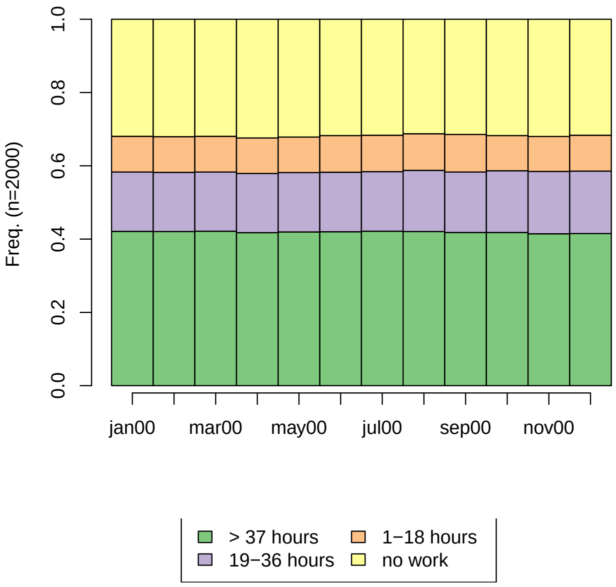


Figure 7.3: Distribution of the work statuses by month in the *actcal* data set (data from the Swiss Household Panel)

```
4 0.000 0.000 0.0000 0.0005 0.0000 0.000 0.0000 0.0005 0.0010 0.0000 0.001
5 0.000 0.000 0.0000 0.0000 0.0015 0.003 0.0035 0.0045 0.0055 0.0060 0.006
6 0.000 0.000 0.0005 0.0050 0.0110 0.027 0.0470 0.0805 0.1175 0.1520 0.207
7 0.000 0.000 0.0000 0.0000 0.0000 0.000 0.0005 0.0020 0.0025 0.0065 0.012
    a26   a27    a28    a29    a30
0 0.201 0.158 0.1275 0.0980 0.0770
1 0.231 0.203 0.1735 0.1580 0.1390
2 0.081 0.085 0.0910 0.0965 0.1035
3 0.203 0.215 0.2050 0.1980 0.1880
4 0.001 0.001 0.0015 0.0015 0.0015
5 0.007 0.011 0.0125 0.0120 0.0115
6 0.260 0.305 0.3615 0.4030 0.4415
7 0.017 0.022 0.0275 0.0330 0.0380

   [Valid states]
   a15  a16  a17  a18  a19  a20  a21  a22  a23  a24  a25  a26  a27  a28  a29
N 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000
   a30
N 2000

   [Entropy index]
    a15   a16  a17 a18  a19  a20  a21  a22  a23  a24  a25 a26 a27  a28  a29
H 0.035 0.098 0.13 0.2 0.29 0.41 0.52 0.61 0.68 0.74 0.78 0.8 0.8 0.79 0.77
   a30
H 0.75
```

In addition to the state distribution at each time point, the `seqstatd()` function provides also for each time point the number of valid states and the Shannon entropy of the observed state distribution. Letting p_i denote the proportion of cases in state i at the considered time point, the entropy is

$$h(p_1, \ldots, p_s) = -\sum_{i=1}^{s} p_i \log(p_i)$$

where s is the size of the alphabet. The `log` is here the natural (base e) logarithm. The entropy is 0 when all cases are in the same state and is maximal when the same proportion of cases are in each state. The entropy can be seen as a measure of the diversity of states observed at the considered time point. Billari (2001) and Fussell (2005) considered for instance such entropy values for studying early life trajectories, the latter author applying the concept on aggregated virtual trajectories derived from transversal data.

Let us look at our example above. At age 15, 99% of the respondents had not leaved parental home (state 0), hence the entropy is very low (0.035). The entropy of the state distribution rises with age and reaches its maximum at age 27. At this age, 16% percent of the respondents had not leaved parental home, 20% had leaved parental home but were not married and had no children (state 1), 1% had one or more children without being married, and 30% had one or more children and were married (state 6).

We can also plot the reported entropy measures. For that we use the `seqHtplot()` function. By the way, we illustrate also how we can save the graphic in a pdf file so that it can for instance be inserted into this manual. To do this, we open a pdf file with the `pdf()` function, create the graphic with the plot command and close the pdf file with the `dev.off()` function. The result is shown in figure 7.4. Of course, if you want to run this program on your system, you should adapt the path to the 'pdf' file to your convenience. Users who prefer to save their graphics in the postscript format can use `postscript()` instead of `pdf()`. There are likewise `png()`, `jpeg()`, ... functions.

```
R> sd <- seqstatd(biofam.seq)
R> pdf(file = "Graphiques/fg_biofam-entropy.pdf", width = 8, height = 8,
+      pointsize = 14)
R> seqHtplot(biofam.seq)
R> dev.off()
```

7.2.3 Sequence frequencies

Sequence frequency plot The `seqfplot()` function plots the most frequent sequences. Each sequence is plotted as a horizontal bar split in as many colorized cells as there are states in the sequence. The sequences are ordered by decreasing frequency from bottom up and the bar widths are set proportional to the sequence frequency (use `pbarw=FALSE` to disable this feature). By default, the 10 most frequent sequences are plotted. However, you can select the number of most frequent sequences to plot with the `tlim` option. The next command plots for instance the 5 most frequent sequences of the `actcal.seq` sequence object. The resulting plot is shown in Figure 7.5. The labels appearing in the plot's legend are those attached to the object page 53. Notice that the legend is plotted on the right using the `withlegend="right"` option. The frequency plot for the `biofam.seq` sequence object is obtained with the following commands and shown in Figure 7.6. The most frequent sequence, living with parents without being in a partnership or having children from age 15 to 30, is shared by less than 8% of the cases. This does not mean that the most frequent case is to live with both parents until age 30. But, because the timing of the events of family formation spreads over many years, its variability is high and the probability of having many individuals with exactly the same calendar, i.e. changing to the same statuses at the same age, is low.

Sequence frequency table Instead of the plot, you may want numerical details (counts and percentage) about the most frequent sequences. The `seqtab()` function returns a frequency table

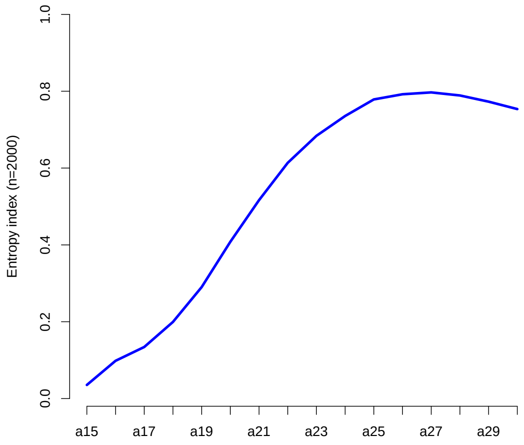


Figure 7.4: Entropy of state distribution by age - *biofam* data set

of the distinct sequences in the data set. Since the number of distinct sequences can be very high, only the first 10 sequences `tlim=1:10` in the list are displayed by default. One can override the default value of `tlim` to select any sequence or group of sequences in the frequency table. The following example shows the frequency table for the `actcal.seq` sequence object created from `actcal` the data set about the activity calendar (the meaning of the states A, B, C, D is given in Table 3.1 on page 21). The most frequent sequence (38%) in the data set is full time paid-job during all the period (January to December 2000) and appears 757 times. The second most frequent sequence (25%) is no-paid job during all the period and appears 508 times. Note that the sequences are displayed in the more readable SPS format.

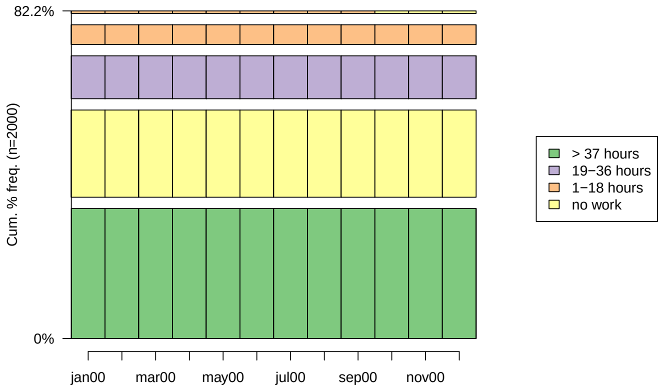


Figure 7.5: Plot of the 10 most frequent sequences in the *actcal* data set

```
R> seqtab(actcal.seq)
```

```
         Freq Percent
A/12      757   37.85
D/12      508   25.40
B/12      250   12.50
C/12      115    5.75
C/9-D/3    15    0.75
A/10-B/2   12    0.60
B/10-C/2    8    0.40
B/11-A/1    8    0.40
D/11-C/1    8    0.40
D/9-C/3     8    0.40
```

We can ask for the sequence frequency table for months July (7) to September (9) only

```
R> seqtab(actcal.seq[, 7:9])
```

```
        Freq Percent
A/3      813   40.65
D/3      581   29.05
B/3      308   15.40
C/3      174    8.70
D/2-C/1   15    0.75
C/2-D/1   11    0.55
A/1-D/2    9    0.45
D/1-C/2    9    0.45
A/2-D/1    8    0.40
D/1-A/2    8    0.40
```

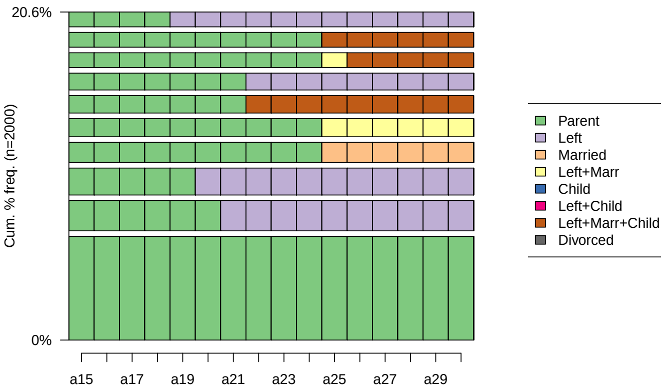


Figure 7.6: Plot of the 10 most frequent sequences in the *biofam* data set (bar widths proportional to the sequence frequencies)

7.2.4 Transition rates

The `seqtrate()` function computes the transition rates between states or events. The outcome is a matrix where each rows gives a transition distribution from associated originating state (or event) in t to the states in $t+1$ (the figures sum to one in each row).

In the following example, the transition rate matrix for the *actcal* (activity calendar) data set is computed. Transition rates from one state to the same state (diagonal elements) have values close to 1, meaning that a person in a given state at time t has a great probability to remain in the same state at time $t+1$. The 'instability' is a bit higher for the state C (part-time paid job from 1 to 18 hours a week), since the probability of staying in that state is 0.94, while the 'instability' of state A is the lowest with a probability of staying in that state of 0.99.

```
R> tr <- seqtrate(actcal.seq)
R> round(tr, 2)
```

```
       [-> A] [-> B] [-> C] [-> D]
[A ->]   0.99   0.01   0.00   0.01
[B ->]   0.01   0.97   0.01   0.01
[C ->]   0.01   0.01   0.93   0.05
[D ->]   0.01   0.01   0.01   0.97
```

Notice that the matrix is not symmetrical. The transition rate between states A and B is 0.005 (0.5%), while the transition rate from B to A is 0.01 (1%). As claimed above, the sum of the transition rates from one state to all other states (including the transition rate between the state and itself) should equal 1. But we don't trust anybody and we want to check it. We therefore apply the `rowSums()` function, which returns the sum of the rows, to the *tr* object containing the transition rates

```
R> rowSums(tr)
```

```
[A ->] [B ->] [C ->] [D ->]
     1      1      1      1
```

Of course there is a shorter way that leads to the same result

```
R> rowSums(seqtrate(actcal.seq))
```

```
[A ->] [B ->] [C ->] [D ->]
     1      1      1      1
```

7.2.5 Mean time spent in each state

We may be interested in the mean time spent in each state. TraMineR provides a special function called `seqmtplot()` to visualize the mean time values. In the next example, we use this function together with the `group` option to visualize the mean times for each sex separately (Figure 7.7). As for the other plotting functions, the colors for representing the states are automatically retrieved from the sequence object.

```
R> seqmtplot(actcal.seq, group = actcal$sex, title = "Mean time")
```

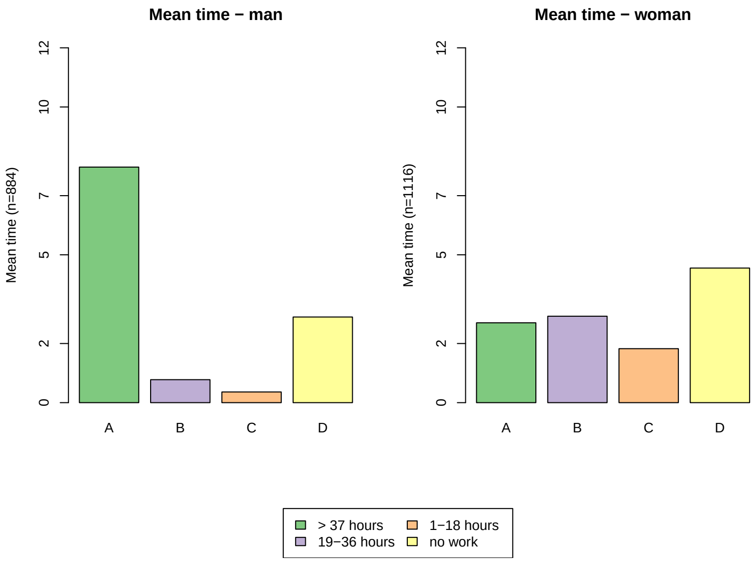


Figure 7.7: Mean time spent in each state, *actcal* data.

7.3 Describing and visualizing individual sequences

7.3.1 Visualizing individual sequences

The `seqiplot()` function renders individual sequences with stacked bars depicting the statuses over time in the same manner as `seqfplot()`. The difference is that `seqiplot()` does neither select nor rank the sequences according to their frequencies. The interest of such plots, known as index-plots, has for instance been stressed by Scherer (2001), Brzinsky-Fay et al. (2006) and Gauthier (2007).

In TraMineR you can select the indexes of the sequences to be plotted with the `tlim` option, which takes 1:10 as default value, i.e. the 10 first rows of the sequence object. Several other options are available to fine tune the graphic. You find their description in the reference manual or in on-line help of the function, which you get by typing `?seqiplot` or `help(seqiplot)`. In the first example below, the 10 first sequences in the `actcal.seq` sequence object are plotted (Figure 7.8). The legend uses the labels attached to the `actcal.seq` object and the color palette is the one set by default. In the next example, we plot all sequences in the previously defined `mvad.seq` sequence object separately for the values of the *gcse5eq* [3] variable using the `group` option. The `border=NA` option specifies that the borders of the bars are not plotted and the `space=0` option that the bars representing individual sequences are plotted without space between them, yielding a more clean graphic when a large number of sequences are plotted.

The `group` option can be useful to distinguish patterns depending on a covariate value. Here the right sequences correspond to young people who gained higher qualifications by the end of compulsory education, and whose large proportion will continue school up to higher education. We observe that the color corresponding to higher education is much more frequent in the right plot, while the colors corresponding to training and employment are much more frequent in the left plot.

[3] binary dummy indicating qualifications gained by the end of compulsory education, yes=5+GCSEs at grades A-C, or equivalent

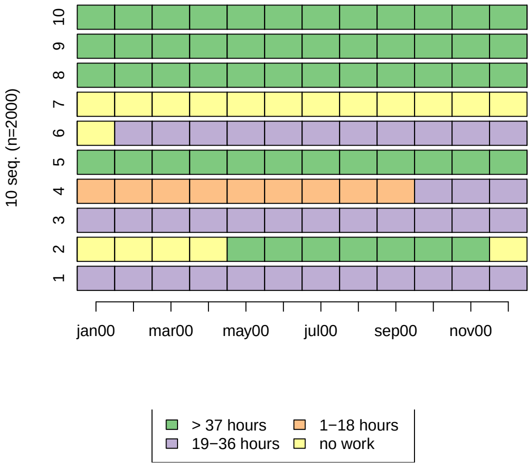


Figure 7.8: Plot of the 10 first sequences of the *actcal* data set

This plot of individual sequences complements the "averaged" representation provided by the state distribution plot by rendering the diversity of the sequences. However, such index plots for thousands of sequences result in very heavy graphic files if they are stored in PDF or POSTSCRIPT format. To reduce the size, we suggest that you save in that case the figures in png format by using `png()` instead of `postscript()` or `pdf()`. Figure 7.9 was produced as a PNG plot with the following commands:

```
R> png(file = "Graphiques/mvad-seqiplot-all.png", width = 1600,
+       height = 1200, pointsize = 50)
R> seqiplot(mvad.seq, group = mvad$gcse5eq, tlim = 0, border = NA,
+       space = 0)
R> dev.off()
```

This results in a degradation of the graphic's quality but permitted, however, to reduce the size of this manual in pdf format by about 5MB.

7.3.2 Finding sequences with a given subsequence

The `seqpm()` function counts the number of sequences that contain a given subsequence and collects their row index numbers. The function returns a list with two elements. The first element, `MTab`, is just a table with the number of occurrences of the given subsequence in the data. Note that only one occurrence is counted per sequence, even when the sub-sequence appears more than one time in the sequence. The second element of the list, `MIndex`, gives the row index numbers of the sequences containing the subsequence. These index numbers may be useful for accessing the concerned sequences (example below). Since it is easier to search a pattern in a character string, the function first translates the sequence data in this format when using the `seqconc` function with the `TRUE` option.

In the following example, we search for the pattern 'DAAD' (see Table 3.1 page 21 for the meaning of the states) into the activity calendar sequence data object.

```
R> seqpm(actcal.seq, "DAAD")
```

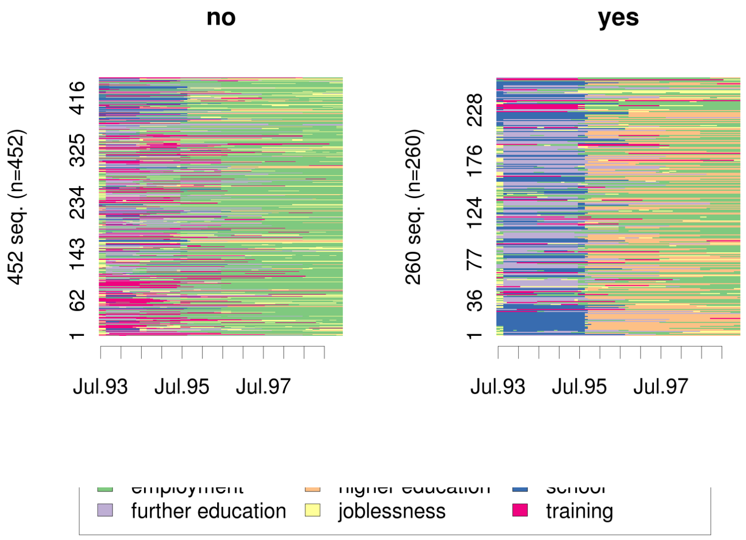


Figure 7.9: Plot of all sequences of the *mvad* data set, grouped according to the gcse5eq variable

```
$MTab
  pattern nbocc
1    DAAD     4

$MIndex
[1]  964  967 1197 1797
```

Four sequences contain the pattern. If we want to look at the sequences containing the 'DAAD' subsequence, we use the '$MIndex' element of the list returned by the `seqpm()` function. We first store the result of the function in an object named `daad` and then access the sequences containing the pattern using `daad$MIndex` as row index for the `actcal.seq` sequence object (since we want all the columns we leave the column index empty).

```
R> daad <- seqpm(actcal.seq, "DAAD")
R> actcal.seq[daad$MIndex, ]
```

```
     Sequence
3660 D-A-A-D-D-D-D-D-D-A-A-A
6829 D-D-A-A-D-D-A-A-A-A-A-A
6040 D-D-B-B-C-D-D-A-A-D-C-C
5489 D-D-D-D-A-A-D-A-B-B-D-D
```

Chapter 8

Sequence characteristics and associated measures

Results shown in this chapter are obtained with:
TraMineR version 1.8
R version 2.9.2 (2009-08-24)-platform: i486-pc-linux-gnu.

This chapter focuses on the characterization of individual sequence properties and their summary.

8.1 Basic sequence characteristics

8.1.1 Sequence length

The `seqlength()` function returns the length of the sequences in a sequence object.

```
R> data(famform)
R> famform.seq <- seqdef(famform)
R> famform.seq
```

```
    Sequence
[1] S-U
[2] S-U-M
[3] S-U-M-MC
[4] S-U-M-MC-SC
[5] U-M-MC
```

```
R> seqlength(famform.seq)
```

```
    Length
[1]      2
[2]      3
[3]      4
[4]      5
[5]      3
```

8.2 Distinct states and durations

A sequence can be considered as an ordered list of the distinct states that an individual has passed through and their associated durations. This is the way the state-permanence SPS format represents sequences as shown here for the first rows of the `actcal.seq` object [1]

```
R> print(head(actcal.seq), "SPS")
```

```
    Sequence
[1] (B,12)
[2] (D,4)-(A,7)-(D,1)
[3] (B,12)
[4] (C,9)-(B,3)
[5] (A,12)
[6] (D,1)-(B,11)
```

The `seqdss()` and `seqdur()` functions are provided to extract distinct states and durations from sequences. Such separated information is required for example for computing sequence turbulence as will be explained below in Section 8.5.1 on page 85. In the following example we extract this separated information from the first six sequences of the `actcal.seq` object. Distinct sequences are obtained with

```
R> seqdss(head(actcal.seq))
```

```
     Sequence
2848 B
1230 D-A-D
2468 B
654  C-B
6946 A
1872 D-B
```

and durations with

```
R> seqdur(head(actcal.seq))
```

```
     DUR1 DUR2 DUR3
2848   12   NA   NA
1230    4    7    1
2468   12   NA   NA
654     9    3   NA
6946   12   NA   NA
1872    1   11   NA
```

Note that durations are stored in a matrix with a number of columns equal to the maximum sequence length encountered. This is because in a sequence of length 12 for instance, there can be at most 12 possible distinct states.

[1] `head(actcal.seq)` is equivalent to `actcal.seq[1:6,]`

8.3 Summarizing the DSS

8.3.1 Number of subsequences

The idea of subsequence is an extension of the notion of substring and is described in detail for instance in Elzinga (2008). While a substring of a sequence is necessarily constituted of adjacent symbols, this requirement is relaxed with the notion of subsequence. Thus if $x = abac$, λ (the empty string), $u = b$, $v = bac$ and $w = bc$ belong to the set of subsequences of x, while only λ, $u = b$ and $v = bac$ are substrings of x.

The `seqsubsn` function returns the number of subsequences contained in a sequence.

```
R> seqsubsn(head(actcal.seq))
```

```
     Subseq.
2848       2
1230       7
2468       2
654        4
6946       2
1872       4
```

8.3.2 Number of transitions

Computing the length of a sequence's DSS yields immediately the number of transitions contained in the sequence. This is illustrated below with the *mvad* data set. We first have a look at the first 10 sequences

```
R> print(head(mvad.seq), format = "SPS")
```

```
    Sequence
[1] (TR,2)-(EM,4)-(TR,2)-(EM,64)
[2] (JL,2)-(FE,36)-(HE,34)
[3] (JL,2)-(TR,24)-(FE,34)-(EM,10)-(JL,2)
[4] (TR,49)-(EM,14)-(JL,9)
[5] (JL,2)-(FE,25)-(HE,45)
[6] (JL,3)-(TR,33)-(EM,36)
```

Next we extract the DSS for each sequence and then compute its length

```
R> mvad.dss <- seqdss(mvad.seq)
R> head(mvad.dss)
```

```
  Sequence
1 TR-EM-TR-EM
2 JL-FE-HE
3 JL-TR-FE-EM-JL
4 TR-EM-JL
5 JL-FE-HE
6 JL-TR-EM
```

```
R> seqlength(head(mvad.dss))
```

```
  Length
1      4
2      3
3      5
4      3
5      3
6      3
```

We can see that substracting 1 to the DSS length gives the number of transition in the sequence.

8.4 Summarizing state durations

8.4.1 Variance of the state durations

8.4.2 Cumulated state durations

The seqistatd() function returns for each sequence the time spent in the different states.

```
R> seqistatd(actcal.seq[1:6, ])
```

```
      A  B C D
2848  0 12 0 0
1230  7  0 0 5
2468  0 12 0 0
654   0  3 9 0
6946 12  0 0 0
1872  0 11 0 1
```

We may be interested in the mean time spent in each state. These mean times can be computed by means of the apply() function, with which we can “apply” the mean() function to each column of the matrix outputted by seqistatd(). In the following example, we first store the outcome of the seqistatd() function in statd, and then compute the mean value by columns (2nd dimension) with the apply function.

```
R> statd <- seqistatd(actcal.seq)
R> apply(statd, 2, mean)
```

```
     A      B      C      D
5.0275 1.9745 1.1780 3.8200
```

TraMineR provides a special function to visualize the mean time values, described in Chapter 7.

8.4.3 Within sequence entropy

In order to measure the diversity of the states in a given sequence, TraMineR offers two functions: The first one measures the entropy of the sequence and the second one, which is discussed later in Section 8.5.1, is the Turbulence.

TraMineR provides the function seqient() that returns the Shannon entropy of each sequence in the data. The entropy of a sequence is computed using the formula

$$h(\pi_1, \ldots, \pi_s) = -\sum_{i=1}^{s} \pi_i \log \pi_i$$

where s is the size of the alphabet and π_i the proportion of occurrences of the ith state in the considered sequence. The log is here the natural (base e) logarithm. The entropy can be interpreted as the 'uncertainty' of predicting the states in a given sequence. If all states in the sequence are the same, the entropy is equal to 0. The maximum entropy for a sequence of length 12 with an alphabet of 4 states is 1.386294 and is attained when each of the four states appears 3 times.

The `seqient()` function returns a vector containing the entropy for each sequence of the provided sequence object. By default, `sequient()` normalizes the entropy by dividing the value of $h(\pi_1, \ldots, \pi_s)$ by the entropy of the alphabet. The latter is indeed an upper bound of the entropy that corresponds to the maximal possible entropy when the sequence length is a multiple of the alphabet size. The normalized entropy has a maximal value of 1. Unstandardized entropies can be obtained with the `norm=F` option.

In the example below, the normalized entropies for the 10 first sequences of the `actcal.seq` object are computed and the results are stored in an object named `actcal.ient`. As expected, the entropy for the first sequence is 0, since it belongs to an individual who worked full-time during all the period. The entropy is higher for sequence number 2, which describes an individual who changed many times his activity status

```
R> actcal.ient <- seqient(actcal.seq)
R> head(actcal.ient)
```

```
       Entropy
2848 0.0000000
1230 0.4899344
2468 0.0000000
654  0.4056391
6946 0.0000000
1872 0.2069084
```

Note that this entropy measure does not account for the ordering of the states in the sequence. To demonstrate this, we construct a small data set containing three sequences with the same states ordered differently. We first construct a vector for each sequence, combine the obtained vectors into a matrix with the `rbind()` function and eventually convert the matrix into a sequence object.

```
R> s1 <- c("A", "A", "A", "B", "B", "B", "C", "C", "C", "D", "D", "D")
R> s2 <- c("A", "D", "A", "B", "C", "B", "C", "B", "C", "D", "A", "D")
R> s3 <- c("A", "B", "A", "B", "A", "B", "C", "D", "C", "D", "C", "D")
R> ex1 <- rbind(s1, s2, s3)
R> ex1
```

```
   [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
s1 "A"  "A"  "A"  "B"  "B"  "B"  "C"  "C"  "C"  "D"   "D"   "D"  
s2 "A"  "D"  "A"  "B"  "C"  "B"  "C"  "B"  "C"  "D"   "A"   "D"  
s3 "A"  "B"  "A"  "B"  "A"  "B"  "C"  "D"  "C"  "D"   "C"   "D"  
```

```
R> ex1 <- seqdef(ex1)
```

Now that the sequence object is created we display its content.

```
R> ex1
```

```
   Sequence
s1 A-A-A-B-B-B-C-C-C-D-D-D
s2 A-D-A-B-C-B-C-B-C-D-A-D
s3 A-B-A-B-A-B-C-D-C-D-C-D
```

We check with the `seqistatd()` that the three sequences in the `ex1` object contain the same number of A,B,C and D states.

```
R> seqistatd(ex1)
```

```
   A B C D
s1 3 3 3 3
s2 3 3 3 3
s3 3 3 3 3
```

Now we are ready to verify that the entropy is the same for all sequences. As shown by the results of the `seqient()` function, our claim is true. The normalized entropy equals the maximum theoretical entropy, i.e. the entropy of a sequence with all states equally frequent. Unlike the entropy, Elzinga (2006)'s turbulence measure, which you may also get with TraMineR (see Section 8.5.1), is sensitive to the state ordering.

```
R> seqient(ex1)
```

```
   Entropy
s1       1
s2       1
s3       1
```

The non normalized entropy is obtained with the `norm=FALSE` option

```
R> seqient(ex1, norm = FALSE)
```

```
    Entropy
s1 1.386294
s2 1.386294
s3 1.386294
```

Now we are very impatient to plot an histogram of the within entropy of the sequences in the *actcal* data set. We thus plot the *actcal.ient* object using the `hist()` function

```
R> hist(actcal.ient, col = "cyan", main = NULL, xlab = "Entropy")
```

The histogram can be seen in Figure 8.1. By the way, we produce some summary statistics using the `summary()` function and learn that the mean and the maximum normalized entropy are respectively 0.07484 and 0.97957.

```
R> summary(actcal.ient)
```

```
    Entropy
 Min.   :0.00000
 1st Qu.:0.00000
 Median :0.00000
 Mean   :0.07484
 3rd Qu.:0.00000
 Max.   :0.97957
```

Now we would like to know what the maximum value of the within sequence entropy is and look at the sequence(s) reaching this maximum value. The `max()` function returns the maximum of the `actcal.ient` vector of within sequence entropies.

```
R> max(actcal.ient)
```

```
[1] 0.979574
```

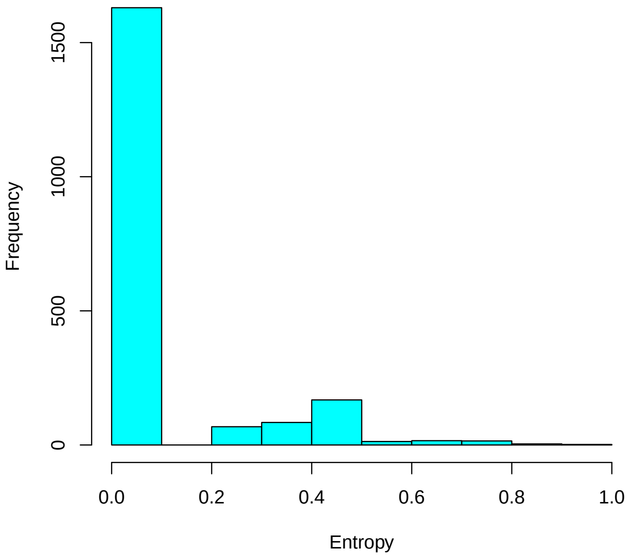


Figure 8.1: Within sequence entropies - *actcal* data set

The `which()` function is used to locate the row index number(s) of the sequences that reach this maximum entropy. It is here the row number 1836

```
R> index <- which(actcal.ient == max(actcal.ient))
R> index
```

```
[1] 1836
```

```
R> actcal.seq[index, ]
```

```
     Sequence
5587 A-B-B-C-D-D-D-D-C-C-A-A
```

The same result can be obtained more simply but also more mysteriously with a single command. Below we display the rows of the `actcal` data frame which contain more information than the sole sequences of the `actcal.seq` object, and we can see that this is a woman aged 37, having two children aged 14 and 10.

```
R> actcal[actcal.ient == max(actcal.ient), ]
```

```
     idhous00 age00       educat00 civsta00 nbadul00 nbkid00 aoldki00 ayouki00
5587   116151    37 apprenticeship  married        2       2       14       10
                        region00                                 com2.00   sex
5587 Lake Geneva (VD, VS, GE) Industrial and tertiary sector communes woman
     birthy jan00 feb00 mar00 apr00 may00 jun00 jul00 aug00 sep00 oct00 nov00
5587   1963     A     B     B     C     D     D     D     D     C     C     A
     dec00
5587     A
```

The distribution of the within sequence entropies looks quite different for the *biofam* data set as shown in Figure 8.2 obtained with the following commands

```
R> biofam.ient <- seqient(biofam.seq)
```

```
R> hist(biofam.ient, main = NULL, col = "cyan", xlab = "Entropy")
```

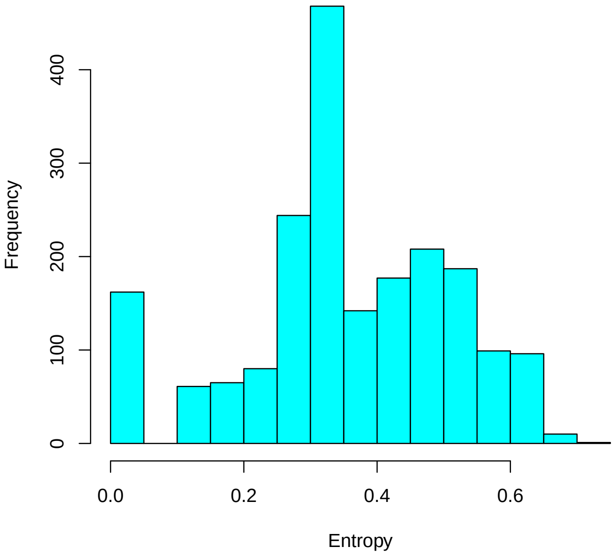


Figure 8.2: Within sequence entropies - *biofam* data set

We would like to compare the values of the entropies conditioned on the value of a covariate. In order to do this, we first add a column with the sequence entropies to the `biofam` data frame.

```
R> biofam <- data.frame(biofam, seqient(biofam.seq))
```

We can check that the `biofam` data frame contains one more variable called `Entropy` and summarize the distribution of the `Entropy` variable.

```
R> names(biofam)
```

```
 [1] "idhous"   "sex"      "birthyr"  "nat_1_02" "plingu02" "p02r01"
 [7] "p02r04"   "cspfaj"   "cspmoj"   "a15"      "a16"      "a17"
[13] "a18"      "a19"      "a20"      "a21"      "a22"      "a23"
[19] "a24"      "a25"      "a26"      "a27"      "a28"      "a29"
[25] "a30"      "wp00tbgp" "wp00tbgs" "Entropy"
```

```
R> summary(biofam$Entropy)
```

```
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
0.0000  0.2987  0.3333  0.3548  0.4729  0.7028
```

Let us have a look at the sequences near the minimum, median and maximum entropy. For that, we draw sets of sequences having an entropy lower or equal to the 1st percentile, an entropy near the median, and an entropy greater than the 99th percentile. We first store the percentiles in the `ient.quant` vector for later usage.

```
R> ient.quant <- quantile(biofam$Entropy, c(0, 0.1, 0.45, 0.55,
+       0.9, 1))
R> ient.quant
```

```
       0%       10%       45%       55%       90%      100% 
0.0000000 0.1124300 0.3295665 0.3738803 0.5565789 0.7028195 
```

Now we can create a categorical variable from the quantiles with the `cut()` function. The `include.lowest` is used to include the values equal to the lowest value of the cutting points, i.e an entropy equal to 0

```
R> ient.group <- cut(biofam$Entropy, ient.quant, labels = c("Min",
+       "q10-45", "Median", "q55-90", "Max"), include.lowest = T)
R> table(ient.group)
```

```
ient.group
   Min q10-45 Median q55-90    Max 
   223    741    176    664    196 
```

The *ient.group* factor has 5 levels corresponding to the five intervals defined by the quantiles. But we are mostly interested in only three of the intervals. A way to select them is to redefine the factor with three levels only (all other values of the factor are converted to NA)

```
R> ient.group <- factor(ient.group, levels = c("Min", "Median",
+       "Max"))
R> table(ient.group)
```

```
ient.group
   Min Median    Max 
   223    176    196 
```

Finally, we plot the three sets of sequences separately using the `group` option. Recall that the `seqiplot()` function plots by default only the 10 first sequences, but this is enough. The result is shown in Figure 8.3. It confirms that the more there are different states in the sequence, the higher the entropy.

```
R> seqfplot(biofam.seq, group = ient.group, pbarw = TRUE)
```

We may want to plot the distribution of the entropy by birth cohorts. It does not make sense to use the individual birth years as there are too many different values. Thus, we want to first group the birth years into ten year classes. To do this, we first look at the distribution of the birth years using the `summary()` function. Then, by means of the `cut()` function, we add the new `ageg` cohort variable to the *biofam* data set. The `cut()` function takes three arguments: The name of the variable from which to create classes of values, the bins for creating the classes, and optionally labels of the classes. The `include.lowest=TRUE` option tells the function that the lowest value (1909) should be included in the first group.

```
R> summary(biofam$birthyr)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max. 
   1909    1935    1944    1943    1951    1957 
```

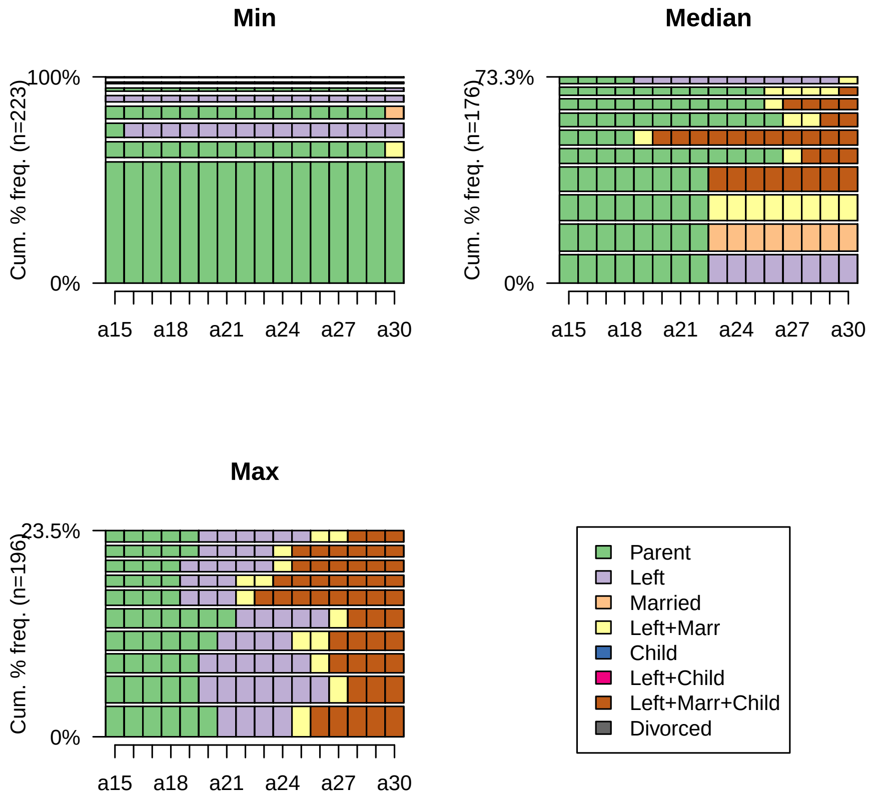


Figure 8.3: Low, median and high sequence entropies - *biofam* data set

```
R> biofam <- data.frame(biofam, ageg = cut(biofam$birthy, c(1909,
+      1918, 1928, 1938, 1948, 1958), label = c("1909-18", "1919-28",
+      "1929-38", "1939-48", "1949-58"), include.lowest = TRUE))
R> table(biofam$ageg)
```

```
1909-18 1919-28 1929-38 1939-48 1949-58 
     35     194     449     620     702 
```

Now we are ready to plot the entropy by ten year age cohorts. We choose the `boxplot()` command. The result is shown in Figure 8.4. The `Entropy` ∼ `ageg` part of the command is a formula syntax widely used in R. Here it means 'plot the entropy by age group'.

```
R> boxplot(Entropy ~ ageg, data = biofam, xlab = "Birth cohort",
+      ylab = "Sequences entropy", col = "cyan")
```

The mean and median entropy are rising in the more recent birth cohorts. Figure (8.4), obtained with the following commands, shows that the entropy is also sligthly higher in the women family formation history when compared to that of the men.

```
R> boxplot(Entropy ~ sex, data = biofam, xlab = "Sex", ylab = "Sequences entropy",
+      col = "cyan")
```

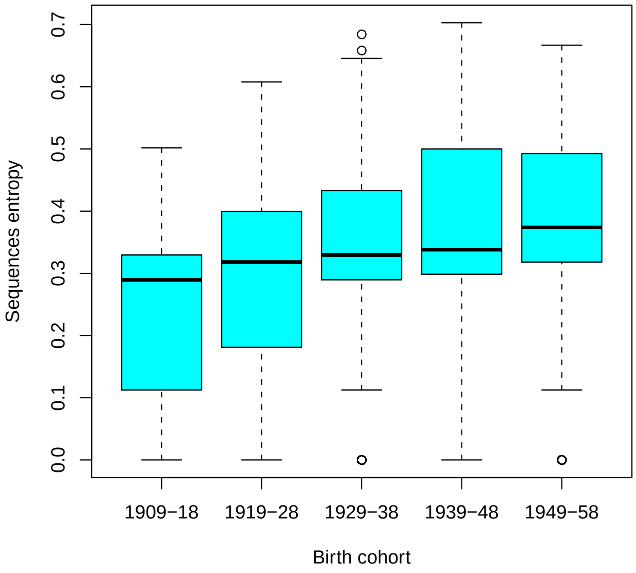


Figure 8.4: Boxplot of the within sequence entropies by birth cohort - *biofam* data set

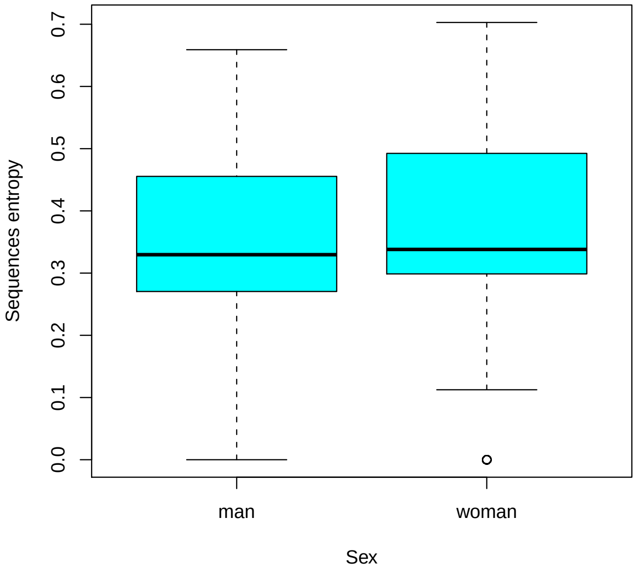


Figure 8.5: Boxplot of the within sequence entropies by sex - *biofam* data set

8.5 Composite measures of sequences complexity

8.5.1 Sequence turbulence

Sequence turbulence is a measure proposed by Elzinga (Elzinga and Liefbroer, 2007; Elzinga, 2006). It is based on the number $\phi(x)$ of distinct subsequences that can be extracted from the distinct state sequence and the variance of the consecutive times t_i spent in the distinct states. For a sequence x, the formula is

$$T(x) = \log_2 \left(\phi(x) \, \frac{s^2_{t,max}(x) + 1}{s^2_t(x) + 1} \right)$$

where s^2_t is the variance of the state-duration for the x sequence and $s^2_{t,max}$ is the maximum value that this variance can take given the total duration of the sequence. This maximum is computed as follows

$$s_{t,max} = (n - 1)(1 - \bar{t})^2$$

where $\bar{t}$ is the mean consecutive time spent in the distinct states, i.e. the sequence duration divided by the number of distinct states in the sequence and n is the length of the distinct state sequence. Elzinga's definition of the turbulence is based on the 'sequence-permanence' SPS data representation of sequences and the number of distinct subsequences considered is that of the distinct state sequences, i.e. the sequence obtained by considering only one of several same consecutive states. In the example below, the x sequence comes from the *actcal* data set and contains 12 elements corresponding to the successive work statuses from January to December 2000. The same sequence formatted in the 'distinct-successive-state' (DSS) format exhibits only 3 elements, as shown by the output of the `seqdss()` function.

```
R> data(actcal)
R> actcal.seq <- seqdef(actcal, 13:24)
R> actcal.seq[2, ]
```

```
     Sequence
1230 D-D-D-D-A-A-A-A-A-A-A-D
```

```
R> seqdss(actcal.seq[2, ])
```

```
     Sequence
1230 D-A-D
```

We can compute the number of distinct subsequences with the `seqsubsn()` function. With the `DSS=FALSE` option, the returned result is 76. With the default `DSS=TRUE` option, the computation is made on the sequence of distinct successive states only ('D-A-D') returning 7 as the number of distinct subsequences.

```
R> seqsubsn(actcal.seq[2, ], DSS = FALSE)
```

```
     Subseq.
1230      76
```

```
R> seqsubsn(actcal.seq[2, ], DSS = TRUE)
```

```
     Subseq.
1230       7
```

The `seqST()` function returns Elzinga's turbulence measure for each sequence of the provided sequence object. We begin with a small example taken from Aassve et al. (2007). The original sequences are defined in SPS format by couples of two character strings[2]. Hence we give the `informat='SPS'` option to the `seqdef()` function for creating the sequence object.

```
R> sp.ex1

     [,1]
[1,] "(000,12)-(0W0,9)-(0WU,5)-(1WU,2)"
[2,] "(000,12)-(0W0,14)-(1WU,2)"

R> sp.ex1 <- seqdef(sp.ex1, informat = "SPS")
```

Now `sp.ex1` is a sequence object. Its content is displayed below in STS format.

```
R> print(sp.ex1, ext = TRUE)

    [1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17]
[1] 000 000 000 000 000 000 000 000 000  000  000  000  0W0  0W0  0W0  0W0  0W0
[2] 000 000 000 000 000 000 000 000 000  000  000  000  0W0  0W0  0W0  0W0  0W0
    [18] [19] [20] [21] [22] [23] [24] [25] [26] [27] [28]
[1]  0W0  0W0  0W0  0W0  0WU  0WU  0WU  0WU  0WU  1WU  1WU
[2]  0W0  0W0  0W0  0W0  0W0  0W0  0W0  0W0  0W0  1WU  1WU
```

We use the `seqST()` function to compute the turbulence

```
R> seqST(sp.ex1)

    Turbulence
[1]   6.813988
[2]   5.292438
```

Let us now compute the turbulence of the sequences in the *biofam* data set. As for the entropy, we add a new `Turbulence` variable with the values of the turbulences to the data frame. Note how this time pass the output of the `seqdef()` function 'on the fly' to the `seqST()` function.

```
R> biofam <- data.frame(biofam, seqST(biofam.seq))
```

To get a first idea of the turbulence distribution we summarize the created variable with the `summary()` function. The mean turbulence is 4.8, with a minimum of 1 and a maximum of 8.807.

```
R> summary(biofam$Turbulence)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  1.000   3.691   5.064   4.800   6.222   8.807
```

We get an histogram for the turbulence of the sequences with the command below, yielding Figure 8.6. Let us mention that the user who does not like the 'cyan' color used in the graphic can indeed use any other color from the list returned by the `colors()` function.

```
R> hist(biofam$Turbulence, main = "Sequence turbulences - biofam data set",
+       col = "cyan", xlab = "Turbulence")
```

The distribution of the turbulences resembles that of the entropy (see Figure 8.2 on page 81). With the following command we look for the most turbulent sequence.

[2] see 7.2.1 for the syntax used to create the *sp.ex1* data set

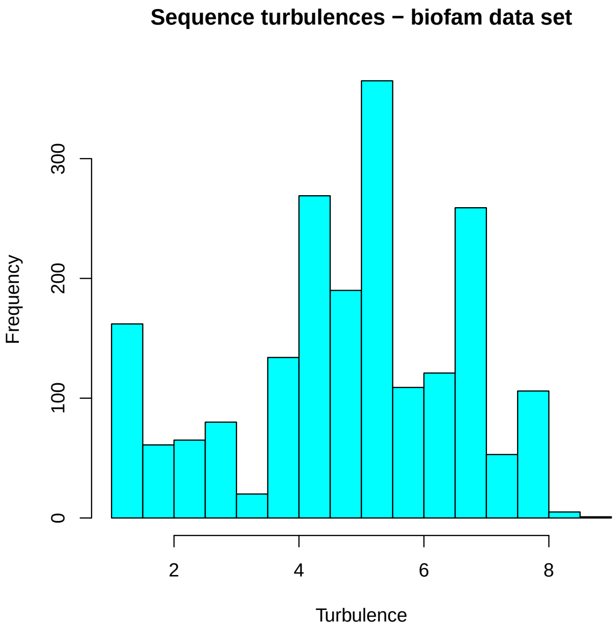


Figure 8.6: Histogram of the sequence turbulences - *biofam* data set

```
R> max.turb <- max(biofam$Turbulence)
R> biofam.tmax <- subset(biofam, Turbulence == max.turb)
R> head(biofam.tmax)
```

```
     idhous   sex birthyr    nat_1_02 plingu02                        p02r01
1098  61871 woman    1953 Switzerland   german Protestant or Reformed Church
                 p02r04              cspfaj cspmoj a15 a16 a17 a18 a19 a20 a21
1098 a few times a year other self-employed   <NA>   0   0   0   0   1   1   1
     a22 a23 a24 a25 a26 a27 a28 a29 a30 wp00tbgp  wp00tbgs   Entropy    ageg
1098   1   3   3   3   3   6   6   6   6 846.5408 0.7512915 0.6666667 1949-58
     Turbulence
1098   8.807355
```

Note the use of the subset() function in the previous command instead of the equivalent command

```
R> biofam[biofam$Turbulence == max.turb, ]
```

The sequence with maximum turbulence is not the same as that with maximum entropy (c.f. Section 8.4.3). It contains four subsequences of equal length. This is best shown using the SPS format.

```
R> max.seq <- which(biofam$Turbulence == max.turb)
R> print(biofam.seq[max.seq, ], format = "SPS")
```

```
    Sequence
[1] (0,4)-(1,4)-(3,4)-(6,4)
```

Nonetheless, the correlation between entropy and turbulence measures is reasonably high, whether we consider the Pearson correlation[3] or the Spearman rank correlation.

[3] The ‘pearson’ method is the default for the cor() function, hence it is not necessary to specify it as an option

```
R> cor(biofam$Turbulence, biofam$Entropy)
```

```
[1] 0.8078864
```

```
R> cor(biofam$Turbulence, biofam$Entropy, method = "spearman")
```

```
[1] 0.731871
```

Figure 8.7 is obtained with the following command and shows the relationship between the two measures.

```
R> plot(biofam$Turbulence, biofam$Entropy, main = "Turbulence vs. Entropy",
+       xlab = "Turbulence", ylab = "Entropy")
```

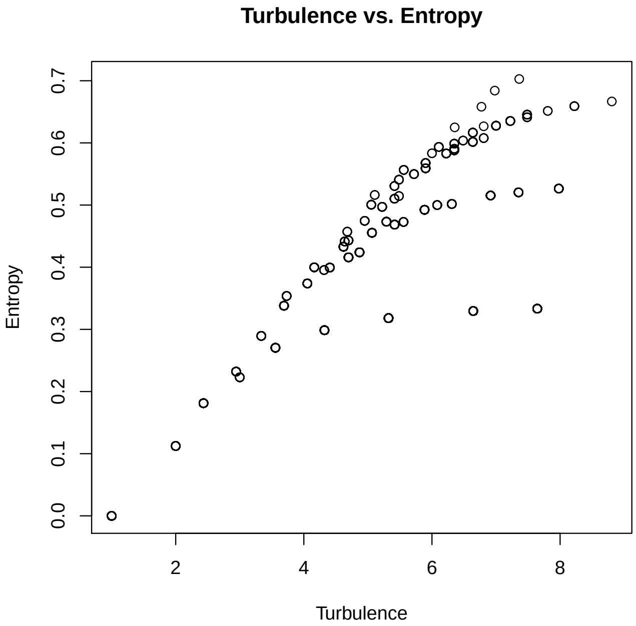


Figure 8.7: Correlation between within sequence turbulence and entropy - *biofam* data set

As previously done with the entropy, we would like to have a look at some sequences having low, medium and high turbulence. This is achieved by first storing the values for the 10, 45, 55 and 90 percentiles

```
R> turb.quant <- quantile(biofam$Turbulence, c(0, 0.1, 0.45, 0.55,
+       0.9, 1))
R> turb.quant
```

```
      0%      10%      45%      55%      90%     100% 
1.000000 2.000000 4.697325 5.321928 6.915230 8.807355 
```

and creating a categorical variable using the percentile values

```
R> turb.group <- cut(biofam$Turbulence, turb.quant, labels = c("Min",
+       "q10-45", "Median", "q55-90", "Max"), include.lowest = T)
R> table(turb.group)
```

```
turb.group
   Min q10-45 Median q55-90    Max 
   223    684    369    542    182 
```

and keeping only the first, third and fith levels of the variable

```
R> turb.group <- factor(turb.group, levels = c("Min", "Median",
+       "Max"))
R> table(turb.group)
```

```
turb.group
   Min Median    Max 
   223    369    182 
```

and next by plotting the frequency plots for each of the three intervals. The plot is shown in figure 8.8

```
R> seqfplot(biofam.seq, group = turb.group, pbarw = TRUE)
```

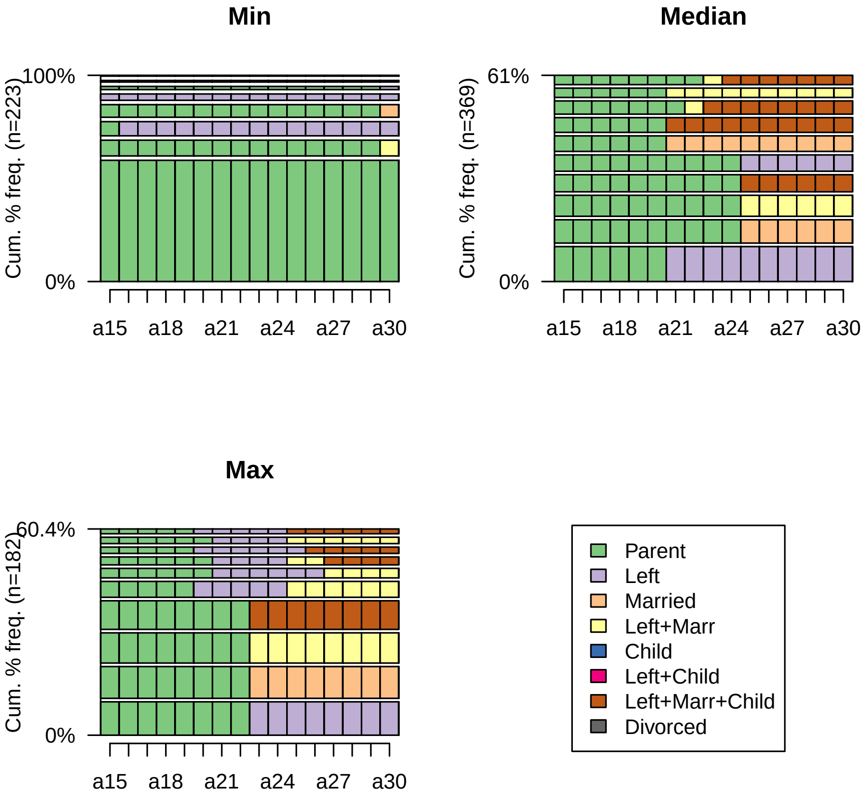


Figure 8.8: Low, median and high sequence turbulences - *biofam* data set

Chapter 9

Measuring similarities and distances between sequences

Results shown in this chapter are obtained with:
TraMineR version 1.8
R version 2.9.2 (2009-08-24)-platform: i486-pc-linux-gnu.

This chapter presents the measures of similarity and distance between sequences available in the TraMineR package. The `seqdist()` function is the main tool provided by the TraMineR package to compute distances between sequences. It can compute the distance matrix, i.e. the distances between all pairs of sequences in the data set, or the distance to a reference sequence, for example to the most frequent sequence. The following metrics are available with `seqdist`:

- the Longest Common Prefix (LCP)
- the Longest Common Subsequence (LCS)
- the Optimal Matching distances (OM)

These metrics and the use of the `seqdist()` are described in the following sections.

9.1 Number of matching positions

The number of matching positions is a simple similarity measure. We get it for a given couple of sequences with the function `seqmpos()` as illustrated below with the `famform` data included in TraMineR.

```
R> data(famform)
R> famform.seq <- seqdef(famform)
R> famform.seq
```

```
    Sequence
[1] S-U
[2] S-U-M
[3] S-U-M-MC
[4] S-U-M-MC-SC
[5] U-M-MC
```

```
R> seqmpos(famform.seq[1, ], famform.seq[2, ])
```

```
[1] 2
```

```
R> seqmpos(famform.seq[2, ], famform.seq[4, ])
```

```
[1] 3
```

9.2 Longest Common Prefix (LCP) distances

One of the measures of similarity/distance between sequences proposed by Elzinga (2008) is based on the length of the longest common prefix (LLCP).

9.2.1 LCP based metric

The prefix of a sequence of characters (states) is defined as follow [1] . Let x be a sequence of length n. The k^{th} prefix of x is defined as $x^k = x_1 ... x_k$, where $0 \leq k \leq n$. If $x = abac$ we have $x^3 = aba$ and $x^4 = x = abac$. The length of a sequence x is written $|x|$ and we have $|x| = 4$, $|x^3| = 3$ and $|x^4| = 4$. Hence x^k is a k-long prefix of x. The empty string λ, of length $|\lambda| = 0$ is a prefix of any sequence and thus $x^0 = \lambda$ for any x.

The LCP based metric uses the length of the longest common prefix of two sequences. Let $\mathcal{P}(x, y)$ be the set of all common nonempty prefixes of a pair of sequence (x, y)

$$\mathcal{P}(x, y) = \{u \neq \lambda : x^{|u|} = u = y^{|u|}\}$$

Since the prefix of any length is unique, the length $A_{\mathcal{P}}(x, y)$ of the longest common prefix of x and y corresponds to the size $|\mathcal{P}(x, y)|$ of this set.

The `seqLLCP()` function returns the value of this measure for a given couple of sequences. Let us take an example with the *famform* data set. We use therefore the *famform.seq* sequence object created in the previous section and compute the LLCP for some of the sequences

```
R> famform.seq
```

```
    Sequence
[1] S-U
[2] S-U-M
[3] S-U-M-MC
[4] S-U-M-MC-SC
[5] U-M-MC
```

```
R> seqLLCP(famform.seq[1, ], famform.seq[2, ])
```

```
[1] 2
```

```
R> seqLLCP(famform.seq[3, ], famform.seq[4, ])
```

```
[1] 4
```

```
R> seqLLCP(famform.seq[3, ], famform.seq[5, ])
```

[1] see Elzinga (2008) for a more complete explanation.

```
[1] 0
```

The LLCP for sequences 1 and 2 is S-U, hence its length is 2. It is S-U-M-MC for sequences 3 and 4, yielding a LLCP of 4. The LLCP is 0 for sequences 3 and 5.

Elzinga proposes as first measure of distance between sequences x and y

$$d_{\mathcal{P}}(x,y) = |x| + |y| - 2A_{\mathcal{P}}(x,y)$$

where $A_{\mathcal{P}}(x,y)$ is the LLCP between sequences x and y.

9.2.2 Computing LCP distances

The LCP distances can be computed with the seqdist() function by specifying the method='LCP' option. The following example reproduces the results shown in the lower triangle of Table 4 in Elzinga (2008):

```
R> seqdist(famform.seq, method = "LCP")

     [,1] [,2] [,3] [,4] [,5]
[1,]    0    1    2    3    5
[2,]    1    0    1    2    6
[3,]    2    1    0    1    7
[4,]    3    2    1    0    8
[5,]    5    6    7    8    0
```

Elzinga suggests also a normalized LCP-metric that is insensitive to the length of the sequences, namely

$$D_{\mathcal{P}}(x,y) = 1 - S_{\mathcal{P}}(x,y)$$

with

$$S_{\mathcal{P}}(x,y) = \frac{A_{\mathcal{P}}(x,y)}{\sqrt{\mid x \mid \cdot \mid y \mid}}$$

This normalized metric is obtained with the option norm=TRUE

```
R> seqdist(famform.seq, method = "LCP", norm = TRUE)

          [,1]      [,2]      [,3]      [,4] [,5]
[1,] 0.0000000 0.1835034 0.2928932 0.3675445    1
[2,] 0.1835034 0.0000000 0.1339746 0.2254033    1
[3,] 0.2928932 0.1339746 0.0000000 0.1055728    1
[4,] 0.3675445 0.2254033 0.1055728 0.0000000    1
[5,] 1.0000000 1.0000000 1.0000000 1.0000000    0
```

Those who prefer similarity measures can easily get them by taking the complement to one of the normalized distance values.

$$S_{\mathcal{P}}(x,y) = 1 - D_{\mathcal{P}}(x,y)$$

```
R> 1 - seqdist(famform.seq, method = "LCP", norm = TRUE)

          [,1]      [,2]      [,3]      [,4] [,5]
[1,] 1.0000000 0.8164966 0.7071068 0.6324555    0
[2,] 0.8164966 1.0000000 0.8660254 0.7745967    0
[3,] 0.7071068 0.8660254 1.0000000 0.8944272    0
[4,] 0.6324555 0.7745967 0.8944272 1.0000000    0
[5,] 0.0000000 0.0000000 0.0000000 0.0000000    1
```

One can check that these values are equal to those in the upper triangle of Table 4 in Elzinga (2008).

9.3 Longest Common Subsequence (LCS) distances

The Longest Common Subsequence, LCS, based distance is another one of the metrics considered by Elzinga (2008) that is available through the `seqdist()` function. The notion of subsequence is described in section 8.3.1.

9.3.1 LCS based metric

Let $S(x, y)$ be the nonempty[2] set of subsequences of sequences x and y. The proposed LCS metric is based on the length of the longest element of S. Let us take the example in Elzinga (2008), consisting of 3 family formation histories with the meaning of the states being the same as in the *famform* data set (see subsection 3.2.4).

```
R> LCS.ex
```

```
    Sequence
[1] S-U-S-M-S-U
[2] U-S-SC-MC
[3] S-U-M-S-SC-UC-MC
```

For convenience we derive from `LCS.ex` 3 distinct sequence objects `x`, `y` and `z` containing each one sequence.

```
R> x <- LCS.ex[1, ]
R> y <- LCS.ex[2, ]
R> z <- LCS.ex[3, ]
```

The length of the longest common subsequence of the first pair of histories (x, y) can be computed with the `seqLLCS()` function.

```
R> seqLLCS(x, y)
```

```
[1] 2
```

The longest common subsequence is indeed `U-S`. Now we compute the longest common subsequence of the pair (x, z)

```
R> seqLLCS(x, z)
```

```
[1] 4
```

The longest common subsequence of (x, z) is `S-U-M-S`. It appears in x as $x_1x_2x_4x_5$ and in z as $z_1z_2z_3z_4$. Now if we define the attribute

$$A_{\mathcal{L}}(x, y) = max\{|u| : u \in S(x, y)\}$$

where $|u|$ is the length of the longest common subsequence for the pair of sequences (x, y), a LCS distance can be defined as

$$d_{\mathcal{L}}(x, y) = |x| + |y| - 2A_{\mathcal{L}}(x, y)$$

and a similarity as

$$s_{\mathcal{L}}(x, y) = \frac{A_{\mathcal{L}}(x, y)}{\sqrt{|x|.|y|}}$$

[2] Since λ, the empty string, is a substring (subsequence) of any sequence, we have $|S(x, y)| \geq 1$

9.3.2 Computing LCS distances

LCS based distances can be computed with the `seqdist()` function using the `method=LCS` option. In the following example the results for the three sequences are those shown in the lower triangle of Table 7 in Elzinga (2008)

```
R> seqdist(LCS.ex, method = "LCS")
```

```
     [,1] [,2] [,3]
[1,]    0    6    5
[2,]    6    0    3
[3,]    5    3    0
```

In the next example, we compute the LCS distances [3] for the *biofam.seq* sequence object previously created from the *biofam* data frame

```
R> biofam.lcs <- seqdist(biofam.seq, method = "LCS")
```

and print the distance matrix for the first 10 sequences

```
R> biofam.lcs[1:10, 1:10]
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]    0   20   10   22   16   14   14   14    4    20
 [2,]   20    0   12   10    8   30   30   14   22     6
 [3,]   10   12    0   12    6   18   18    4   12    16
 [4,]   22   10   12    0    6   22   22   12   22    14
 [5,]   16    8    6    6    0   22   22    8   16    10
 [6,]   14   30   18   22   22    0   14   20   10    32
 [7,]   14   30   18   22   22   14    0   20   14    32
 [8,]   14   14    4   12    8   20   20    0   16    18
 [9,]    4   22   12   22   16   10   14   16    0    22
[10,]   20    6   16   14   10   32   32   18   22     0
```

9.3.3 LCS distances with internal gaps

To compute LCS distances between sequences containing gaps (see section 6.5), one can use the `with.miss=TRUE` option. In that case, missing states are considered as an additional valid state. Let us illustrate this with the following example sequence object

```
R> ex2.seq
```

```
   Sequence
s1 A-B-C-D
s2 A-B-*-C-D
s3 A-B-*-C-D-A
```

Computing LCS distances with the `with.miss=TRUE` option yields the following result

```
R> seqdist(ex2.seq, method = "LCS", with.miss = TRUE)
```

```
     [,1] [,2] [,3]
[1,]    0    1    2
[2,]    1    0    1
[3,]    2    1    0
```

[3] Recall that you can get normalized distances with the `norm=TRUE` option.

According to the formula above, the LCS distance between $s2$ and $s3$ is

$$\begin{aligned} d_{\mathcal{L}}(s2, s3) &= |s2| + |s3| - 2A_{\mathcal{L}}(s2, s3) \\ &= 5 + 6 - 2 * 5 \end{aligned}$$

with $A_{\mathcal{L}}(s2, s3)$, the longest common subsequence of $(s2, s3)$ being of length 5, i.e the length of `A-B-*-C-D`.

9.4 Optimal matching (OM) distances

Optimal matching generates edit distances that are the minimal cost, in terms of insertions, deletions and substitutions, for transforming one sequence into another. This edit distance has first been proposed by Levenshtein (1966) and has been popularized in the social sciences by Abbott (Abbott and Forrest, 1986; Abbott, 2001). The algorithm implemented in TraMineR is that of Needleman and Wunsch (1970).

The `seqdist()` function with `method="OM"` generates the optimal matching distances. In that case additional required arguments are:

- an insertion/deletion (indel) cost
- a substitution-cost matrix, giving the cost for substituting each state/event with another.

9.4.1 The insertion/deletion cost

The indel cost is a single value specified by the user. Its default value is 1.

9.4.2 The substitution-cost matrix

The substitution-cost matrix is a squared matrix of dimension $ns \times ns$, where ns is the number of distinct states in the data (the alphabet). The element (i, j) in the matrix is the cost of substituting state i with state j. Several methods exist to generate the substitution-cost matrix:

- Assigning a constant value, in which case all substitution costs are set equal to this constant (`method="CONSTANT"` option). The default constant value is 2.
- Using the transition rates between states observed in the sequence data (`method="TRATE"` option).

The transition rate between state i and state j is the probability of observing state j at time $t + 1$ given that the state i has been observed at time t. For $i \neq j$, the substitution cost is equal to

$$2 - p(i \mid j) - p(j \mid i)$$

where $p(i \mid j)$ is the transition rate between state i and state j. The transition rates can be obtained by the function `seqtrate()`.

The `seqsubm()` function returns a substitution-cost matrix generated with one of the above two methods. With the `method="CONSTANT"` option you provide the constant as `cval` argument while this argument is ignored with the `method="TRATE"` option. An example with a constant substitution cost is given on page 99. In the example below, the substitution-cost matrix is generated using the transition rates in the data.

```
R> couts <- seqsubm(biofam.seq, method = "TRATE")
R> round(couts, 2)
```

```
      0->  1->  2->  3->  4->  5->  6->  7->
 0-> 0.00 1.95 1.98 1.97 2.00 2.00 1.99 2.00
 1-> 1.95 0.00 2.00 1.92 2.00 2.00 1.98 2.00
 2-> 1.98 2.00 0.00 1.99 1.88 2.00 1.99 1.99
 3-> 1.97 1.92 1.99 0.00 2.00 2.00 1.80 1.99
 4-> 2.00 2.00 1.88 2.00 0.00 1.94 2.00 2.00
 5-> 2.00 2.00 2.00 2.00 1.94 0.00 1.88 2.00
 6-> 1.99 1.98 1.99 1.80 2.00 1.88 0.00 1.99
 7-> 2.00 2.00 1.99 1.99 2.00 2.00 1.99 0.00
```

The alphabet is composed of 8 distinct states, so the substitution-cost matrix has dimension 8×8. We can check with the `range()` function that the minimum cost is 0, for a substitution of one state by itself, and the maximum is 2, meaning that the transition never occurs in the data set.

```
R> range(couts)
```

```
[1] 0 2
```

9.4.3 Generating optimal matching distances

Optimal matching distances are generated with the `seqdist()` function by specifying the '`method="OM"`' option, an insertion/deletion cost and a substitution cost matrix. We begin with a simple example to understand OM distances.

```
R> ex3.seq
```

```
    Sequence
[1] A-B-C-D
[2] A-B-B-D
[3] A-B-C-D-D
[4] A-B-C-D
```

We generate a substitution cost matrix with constant value of 2

```
R> ccost <- seqsubm(ex3.seq, method = "CONSTANT", cval = 2)
R> ccost
```

```
    A-> B-> C-> D->
A->   0   2   2   2
B->   2   0   2   2
C->   2   2   0   2
D->   2   2   2   0
```

and compute the distances using the matrix and the default indel cost of 1

```
R> ex3.OM <- seqdist(ex3.seq, method = "OM", sm = ccost)
R> ex3.OM
```

```
     [,1] [,2] [,3] [,4]
[1,]    0    2    1    0
[2,]    2    0    3    2
[3,]    1    3    0    1
[4,]    0    2    1    0
```

The generated distance matrix contains the minimal editing costs for transforming the sequences into each other. This matrix is symmetrical, the minimal cost of transforming sequence x into sequence y being the same as the one of transforming sequence y into sequence x

- Since a single substitution of the third state is necessary to transform sequence 1 into sequence 2 (and vice versa), the OM distance between them is 2

  ```
  R> ex3.OM[1, 2]
  ```

  ```
  [1] 2
  ```

- One deletion (insertion) allow to turn sequence 4 into sequence 3 (and vice versa), hence the OM distance is 1

  ```
  R> ex3.OM[4, 3]
  ```

  ```
  [1] 1
  ```

- The cheapest way of turning sequence 2 into sequence 3 (or sequence 3 into sequence 2) involves two operations, one insertion/deletion and one substitution, yielding a cost of 3 (1+2)

  ```
  R> ex3.OM[2, 3]
  ```

  ```
  [1] 3
  ```

- Since sequences 1 and 4 are identical, no edit is needed and the OM distance is 0

  ```
  R> ex3.OM[1, 4]
  ```

  ```
  [1] 0
  ```

In the next example, we use the substitution cost matrix previously computed for the *biofam.seq* sequence object using the `seqsubm()` command

```
R> biofam.om <- seqdist(biofam.seq, method = "OM", indel = 3, sm = couts)
```

The computer needed 0.13 minutes, i.e. 8 seconds to create the distance matrix of size 2000 × 2000. The necessary size to store the matrix is roughly 30 Mb[4].

```
R> object.size(biofam.om)/1024^2
```

```
30.5176849365234 bytes
```

Here is the extract of the distance matrix for the 10 first sequences in the data set. We use the `round()` function to get a more readable output

```
R> round(biofam.om[1:10, 1:10], 1)
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
 [1,]  0.0 21.3 11.6 21.6 15.6 13.9 13.9 15.1  4.0  19.3
 [2,] 21.3  0.0 15.4 17.6 11.7 29.4 29.5 13.3 21.3   7.7
 [3,] 11.6 15.4  0.0 11.7  5.8 17.7 17.8  5.7 11.6  21.4
 [4,] 21.6 17.6 11.7  0.0  5.9 21.4 21.8 11.6 21.6  23.6
 [5,] 15.6 11.7  5.8  5.9  0.0 21.5 21.7  7.6 15.6  17.6
 [6,] 13.9 29.4 17.7 21.4 21.5  0.0 13.9 19.6  9.9  31.4
 [7,] 13.9 29.5 17.8 21.8 21.7 13.9  0.0 19.8 13.9  31.4
 [8,] 15.1 13.3  5.7 11.6  7.6 19.6 19.8  0.0 15.1  21.0
 [9,]  4.0 21.3 11.6 21.6 15.6  9.9 13.9 15.1  0.0  21.5
[10,] 19.3  7.7 21.4 23.6 17.6 31.4 31.4 21.0 21.5   0.0
```

[4] The result of the `object.size()` function is in bytes, it is translated into megabytes by dividing it by 1024^2

9.4.4 LCS distance as a special case of OM distance

The LCS distance is equal to the Optimal Matching distance computed with an indel cost of 1 and a constant substitution cost of 2. Let us verify it with the *ex3.seq* sequence used previously.

```
R> ex3.seq

    Sequence
[1] A-B-C-D
[2] A-B-B-D
[3] A-B-C-D-D
[4] A-B-C-D
```

Optimal matching distances were produced with a constant substitution cost of 2 (the default value) and an indel cost of 1 and stored in the *ex3.OM* matrix

```
R> ex3.OM

     [,1] [,2] [,3] [,4]
[1,]    0    2    1    0
[2,]    2    0    3    2
[3,]    1    3    0    1
[4,]    0    2    1    0
```

Now we produce the LCS distance matrix

```
R> ex3.LCS <- seqdist(ex3.seq, method = "LCS")
R> ex3.LCS

     [,1] [,2] [,3] [,4]
[1,]    0    2    1    0
[2,]    2    0    3    2
[3,]    1    3    0    1
[4,]    0    2    1    0
```

We can see that these LCS distances are the same as the OM ones. However, since we may not rely on human brain to compare the two matrices, we look for a way of checking this more rigorously. This is done with the `all.equal()` function

```
R> all.equal(ex3.OM, ex3.LCS)

[1] TRUE
```

9.4.5 Optimal matching with internal gaps

If missing values (internal gaps) are present in the sequences (see 6.5), one can nevertheless compute distances by setting the `with.miss` to `TRUE`. In that case, the substitution cost matrix must contain one additional entry for the 'missing state'. Let us use the following example data set

```
R> ex1

    [1]  [2]  [3]  [4] [5] [6]  [7] [8] [9] [10] [11] [12] [13]
s1 <NA> <NA> <NA>    A   A   A    A   A   A    A    A    A    A
s2    D    D    D    B   B   B    B   B   B    B <NA> <NA> <NA>
s3 <NA>    D    D    D   D   D    D   D   D    D    D <NA> <NA>
s4    A    A <NA> <NA>   B   B    B   B   D    D <NA> <NA> <NA>
s5    A <NA>    A    A   A   A <NA>   A   A    A <NA> <NA> <NA>
s6 <NA> <NA> <NA>    C   C   C    C   C   C    C <NA> <NA> <NA>
```

We define a sequence object with default options for handling missing values, that is, missing values appearing after the last valid state of a sequence are considered as void elements, and other missing values as unknown states (see section 6.5)

```
R> ex2.seq
```

```
   Sequence
s1 A-B-C-D
s2 A-B-*-C-D
s3 A-B-*-C-D-A
```

Now we compute a substitution cost matrix containing an entry for unknown states (the default substitution cost for unknown states is 2, the same as the default substitution cost for the other states) ...

```
R> subm <- seqsubm(ex2.seq, method = "CONSTANT", with.miss = TRUE)
R> subm
```

```
    A-> B-> C-> D-> *->
A->   0   2   2   2   2
B->   2   0   2   2   2
C->   2   2   0   2   2
D->   2   2   2   0   2
*->   2   2   2   2   0
```

... and compute the OM distances with the `with.miss=TRUE` option

```
R> seqdist(ex2.seq, method = "OM", sm = subm, with.miss = TRUE)
```

```
     [,1] [,2] [,3]
[1,]    0    1    2
[2,]    1    0    1
[3,]    2    1    0
```

One should be careful when computing distances between sequences containing unknown states. In the next example, we define a sequence object with two sequences $s3$ and $s4$ containing only missing values ...

```
R> s1 <- c(1, 1, 1, 1, 2, 2, 2, 2, 3, 3, 3, 3)
R> s2 <- rep(NA, 12)
R> ms <- rbind(s1, s1, s2, s2)
R> ms.seq <- seqdef(ms, left = NA, gaps = NA, right = NA)
R> ms.seq
```

```
   Sequence
s1 1-1-1-1-2-2-2-2-3-3-3-3
s1 1-1-1-1-2-2-2-2-3-3-3-3
s2 *-*-*-*-*-*-*-*-*-*-*-*
s2 *-*-*-*-*-*-*-*-*-*-*-*
```

... and compute the OM distances

```
R> subm <- seqsubm(ms.seq, method = "CONSTANT", with.miss = TRUE)
R> seqdist(ms.seq, method = "OM", sm = subm, with.miss = TRUE)
```

```
     [,1] [,2] [,3] [,4]
[1,]    0    0   24   24
[2,]    0    0   24   24
[3,]   24   24    0    0
[4,]   24   24    0    0
```

We can see that the distance between $s3$ and $s4$ is 0, hence they are considered as identical, as are $s1$ and $s2$.

9.5 Clustering distance matrices

A distance matrix does not say much by itself, and once it has been computed, a clustering method is usually applied to aggregate the sequences into a reduced number of groups. In the next example the `agnes()` function provided by the `cluster` library is called to create clusters from the previously computed optimal matching distance matrix (see 9.4.3). The chosen method for clustering is the Ward method.

```
R> library(cluster)
R> clusterward <- agnes(biofam.om, diss = TRUE, method = "ward")
```

Next we plot the dendrogram (Fig. 9.1)

```
R> plot(clusterward, which.plots = 2)
```

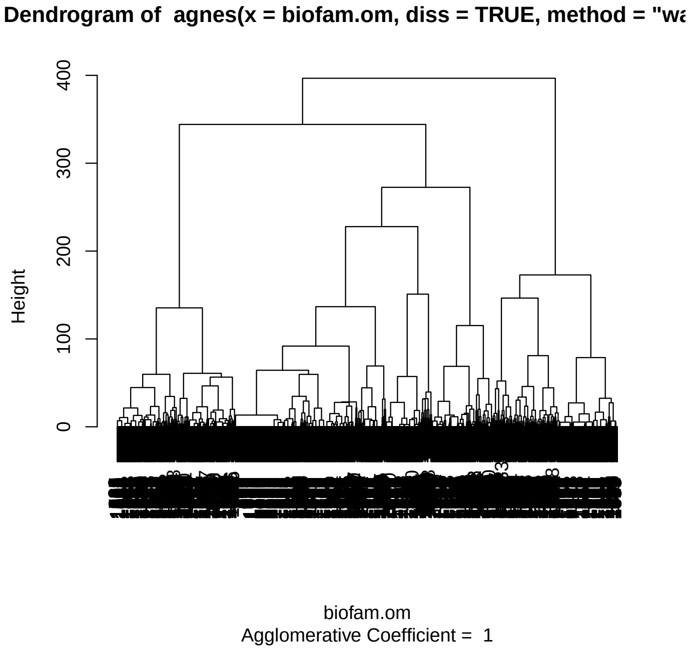


Figure 9.1: Hierarchical sequence clustering from the OM distances, Ward method

The cluster membership for each sequence is then retrieved. A three clusters solution is chosen here.

```
R> cluster3 <- cutree(clusterward, k = 3)
R> cluster3 <- factor(cluster3, labels = c("Type 1", "Type 2", "Type 3"))
R> table(cluster3)
```

```
cluster3
Type 1 Type 2 Type 3
   472    502   1026
```

The cluster3 object is a vector containing the cluster id number for each sequence. We use it to plot graphics helping to identify the typical longitudinal patterns that characterize the clusters. We begin with a frequency plot for each cluster (Fig. 9.2).

```
R> seqfplot(biofam.seq, group = cluster3, pbarw = T)
```

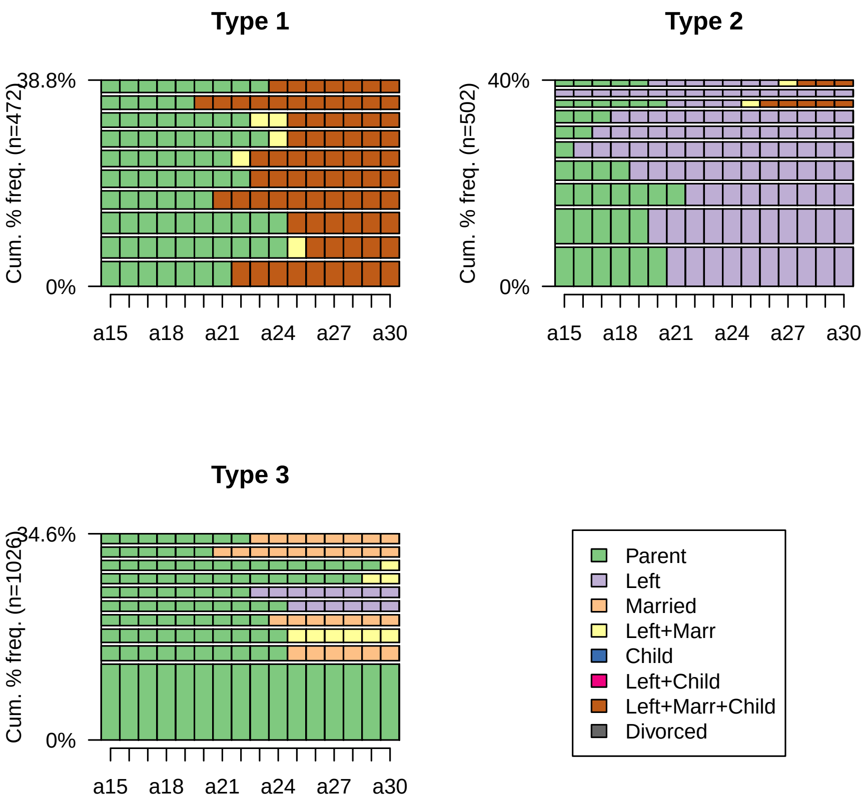


Figure 9.2: Sequence frequencies, by cluster - *biofam* data set

Another help to characterize the patterns within each cluster is to plot the mean times spent in each state

```
R> seqmtplot(biofam.seq, group = cluster3)
```

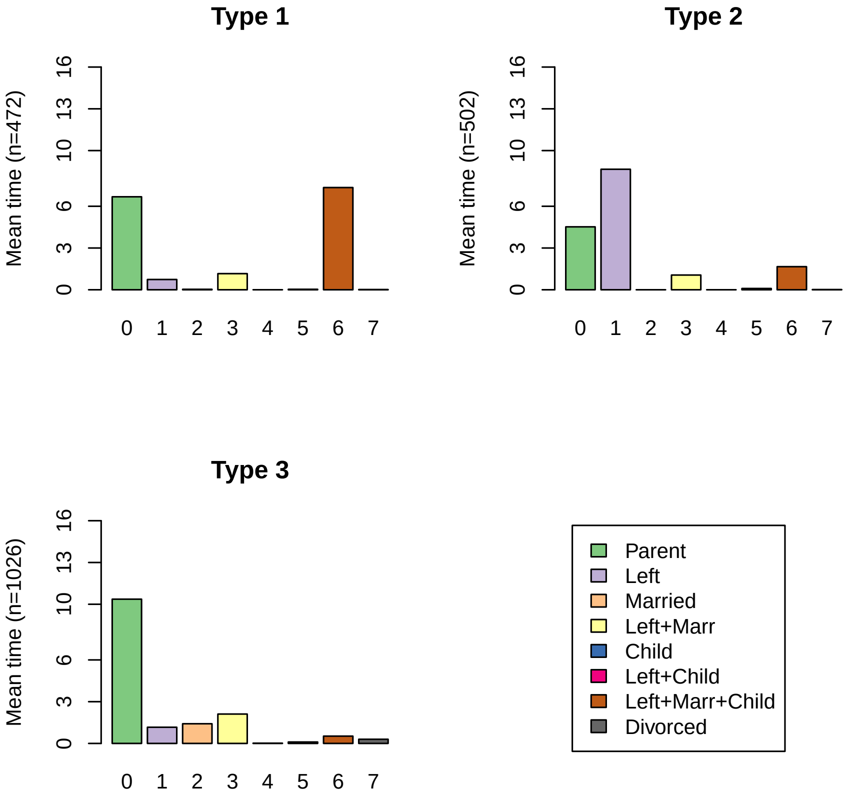


Figure 9.3: Mean time in each state, by cluster - *biofam* data set

Chapter 10

Analysing event sequences

Results shown in this chapter are obtained with:
TraMineR version 1.8
R version 2.9.2 (2009-08-24)-platform: i486-pc-linux-gnu.

The previous chapters dealt essentially with sequences of states. Here, the focus is on sequences of transitions or events. TraMineR offers specific tools for such kind of data that permit, among others, to mine frequent event subsequences (Studer et al., 2008; Agrawal and Srikant, 1995; Zaki, 2001). The TraMineR functions intended for sequences of events start with the "seqe" prefix, which stands for SEQuence of Events.

The concept of event sequence and its formalization were introduced by Agrawal and Srikant (1995) who were mainly interested in frequent buying sequences. We retain here the notation of Zaki (2001) but introduce a new terminology that we think is more appropriate for social sciences. In this chapter, the term "sequence" refers to a sequence of events rather than of states.

Hence, a *sequence* is considered to be an *ordered* list of *transitions*, each *transition* being characterized by the set of distinct *events* that must occur for the transition to take place (an event cannot appear more than once in a same transition). For instance, in the sequence (Leaving Home, Couple) → (Marriage) → (First child), "Leaving Home", "Couple", "Marriage", "First child" are events whereas (Leaving Home, Couple) is the *transition*, defined here by two events, between the state "Has not left home and no partner" to the state "Has left home and has a partner". The distinction between transition and event permits to account for the simultaneity of some events.

In this chapter, we are interested in finding frequent subsequences in our event sequence data set. We propose also tools for identifying among frequent subsequences those that discriminate the most between predefined groups such as between men and women for instance. A subsequence of x is an event sequence that is formed by a subset of the events of sequence x and that respects the order of the events in x. For instance, (Leaving Home, Couple) → (First child) is a subsequence of (Leaving Home, Couple) → (Marriage) → (First child) since the order of transitions and events are respected.

A subsequence is said *frequent* if it occurs in more than a given minimum number of sequences. This minimum required number of sequences to which the subsequence must belong is called *minimum support*. It should be set by the user. A subsequence is said to be *maximal* if it is not included in any other frequent subsequence.

In addition to the support requirement, TraMineR permits also to control the search of frequent subsequences with time constraints. For instance, we can specify a window size (the maximal time span during which a subsequence should occur) as well as maximum gaps (the maximum time between two transitions). Minimum and maximum ages can also be specified to study a particular period of the life course, such as the transition to adulthood for instance.

10.1 Creating event sequences

Let us introduce event sequence analysis with a small example. In order to perform an event sequence analysis, we first create an event sequence object with `seqecreate()`. This function accepts several formats.

Internally, TraMineR uses the TSE format, see Section 5.2.2 for more information. Thus, the natural way to define an event sequence object is from data in TSE form. The *actcal.tse* data set contains the information about the activity calendar in this format. In this case, we can use the following code to create an event sequence.

```
R> data(actcal.tse)
R> actcal.seqe <- seqecreate(id = actcal.tse$id, timestamp = actcal.tse$time,
+      event = actcal.tse$event)
```

We can also create an event sequence object from a state sequence object. To illustrate, assume we are interested in analysing frequent transitions occurring in the family life (*biofam* data set). We first create the state sequence object:

```
R> data(biofam)
R> bfstates <- c("Parent", "Left", "Married", "Left+Marr", "Child",
+      "Left+Child", "Left+Marr+Child", "Divorced")
R> biofam.seq <- seqdef(biofam, 10:25, states = bfstates, labels = bfstates)
```

and convert it into an event sequence object:

```
R> bf.seqe <- seqecreate(biofam.seq)
```

By default, `seqecreate()` creates a distinct (from-state > to-state) event for each found transition. This behavior can be modified through the `tevent` argument. When set to `state` a single to-state event (start event of the spell in the given state), is assigned to each transition (see below).

```
R> bf.seqestate <- seqecreate(biofam.seq, tevent = "state")
```

With the `tevent="period"` option, a pair of events (end-state event, start-state event) is assigned to each transition.

```
R> bf.seqeperiod <- seqecreate(biofam.seq, tevent = "period")
```

You can also provide a custom transition matrix specifying the set of events that define each transition, that is the set of events that are supposed to occur when a transition is observed. It may be useful to use one of the transition matrices automatically produced by the `seqetm()` function as starting point for designing a custom matrix.

To illustrate how resulting event sequences look out, we display the first event sequence from the event sequence objects created respectively with the “transition” (default), “state” and “period” method:

```
R> bf.seqe[1]
```

```
[1] (Parent)-9.00-(Parent>Left+Marr)-1.00-(Left+Marr>Left+Marr+Child)-6.00
```

```
R> bf.seqestate[1]
```

```
[1] (Parent)-9.00-(Left+Marr)-1.00-(Left+Marr+Child)-6.00
```

```
R> bf.seqeperiod[1]
```

```
[1] (Parent)-9.00-(endParent,Left+Marr)-1.00-(endLeft+Marr,Left+Marr+Child)-6.00
```

Event sequences are represented using the following form:

```
(e1,e2,...)-elapsedtime-(e2,...)-endtime
```

Where `elapsedtime` is the the time elapsed between two consecutive sets of events, `(e1,e2,...)` is a transition, that is a non empty list of simultaneous events and `endtime` is the time elapsed between the last transition and the end of observation. The string representing the first sequence (transition method) means that the trajectory described starts at time 0 with the "Parent" event (meaning that at the first observed age, the concerned person is living with her/his parents), which is followed nine years later by the event "Parent>LeftMarr" (leaving home and marrying) and one year later "LeftMarr>LeftMarrChild" (first child), which occurs 6 years before the end of the 16 years of observation.

10.2 Searching for frequent event subsequences

The function `seqefsub()` searches for frequent event subsequences. It takes at least two arguments: A event sequence object and a minimum support expressed in number of sequences (`minSupport`) or as a percentage by using the `pMinSupport` argument.

```
R> fsubseq <- seqefsub(bf.seqe, minSupport = 100)
R> fsubseq[1:5]
```

```
                             Subsequence Support Count
1                               (Parent)  0.9860  1972
2                 (Parent)-(Parent>Left)  0.4340   868
3                          (Parent>Left)  0.4340   868
4            (Left+Marr>Left+Marr+Child)  0.2860   572
5 (Parent)-(Left+Marr>Left+Marr+Child)  0.2825   565

Computed on 2000 event sequences
  Constraint           Value
 countMethod One by sequence
```

The function `seqefsub()` returns an object of type "subseqelist". This object can be printed, plotted and indexed to select specific subsequences. In our example, we printed only the first 5 subsequences.

Notice that the subsequences look as event sequences except that they do not hold time information. Hence, the sequence "(Parent)-(Parent>Left)" means staying with parents and then leaving home.

10.2.1 Plotting the results

The results of the `seqefsub()` function can be plotted with the `plot()` function. The graphic exhibits the frequency of each selected subsequence. The following example generates the plot shown in Figure 10.1.

```
R> plot(fsubseq[1:15], col = "cyan")
```

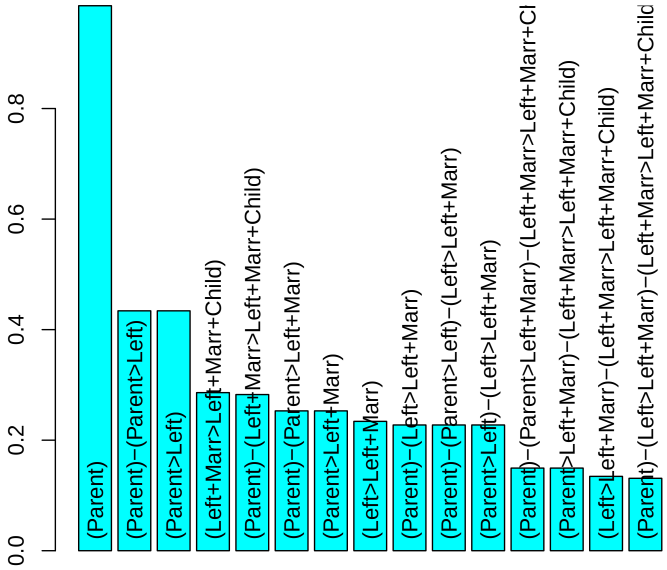


Figure 10.1: Frequencies of 15 most frequent event subsequences

10.3 Time constraints

The functions `seqefsub()` (that searches frequent subsequences) and several others (see below) accept time constraints through the `constraint` parameter. This constraint parameter should be the result of the `seqeconstraint()` function, which has the following parameters:

`maxGap`: The maximum time gap between two transitions.

`windowSize`: The maximum window size, that is the maximum time taken by a subsequence.

`ageMin`: Minimum age at beginning of subsequences.

`ageMax`: Maximum age at beginning of subsequences.

`ageMaxEnd`: Maximum age at end of subsequences.

Each of these parameters is ignored when set equal to -1, which is their default value. The following examples show how to set time constraints. First, we search for subsequences enclosed in a five year interval with no more than two years between two transitions:

```
R> time.constraint <- seqeconstraint(maxGap = 2, windowSize = 5)
R> fsubseq <- seqefsub(bf.seqe, pMinSupport = 0.01, constraint = time.constraint)
R> fsubseq[1:4]
```

```
                  Subsequence Support Count
1                    (Parent)   0.986  1972
2               (Parent>Left)   0.434   868
3 (Left+Marr>Left+Marr+Child)   0.286   572
4          (Parent>Left+Marr)   0.253   506

Computed on 2000 event sequences
  Constraint           Value
      maxGap               2
  windowSize               5
 countMethod One by sequence
```

The ageMin, ageMax and ageMaxEnd are relative values. In our case sequences start at 15 years old. Thus, if we want to search among subsequences beginning between ages 15 and 20, we should set ageMin to 0 (i.e. 15 − 15) and ageMax to 5 (i.e. 20 − 15).

```
R> time.constraint <- seqeconstraint(ageMin = 0, ageMax = 5)
R> fsubseq <- seqefsub(bf.seqe, pMinSupport = 0.01, constraint = time.constraint)
R> fsubseq[1:4]
```

```
                           Subsequence Support Count
1                             (Parent)  0.9860  1972
2                (Parent)-(Parent>Left)  0.4340   868
3 (Parent)-(Left+Marr>Left+Marr+Child)  0.2825   565
4           (Parent)-(Parent>Left+Marr)  0.2530   506

Computed on 2000 event sequences
  Constraint           Value
      ageMin               0
      ageMax               5
 countMethod One by sequence
```

If in addition we are interested only in subsequences that end before 20 years old, we set ageMaxEnd to 20.

```
R> time.constraint <- seqeconstraint(ageMin = 0, ageMax = 5,
+      ageMaxEnd = 5)
R> fsubseq <- seqefsub(bf.seqe, pMinSupport = 0.01, constraint = time.constraint)
R> fsubseq[1:4]
```

```
                  Subsequence Support Count
1                    (Parent)  0.9860  1972
2      (Parent)-(Parent>Left)  0.2205   441
3               (Parent>Left)  0.2205   441
4 (Parent)-(Parent>Left+Marr)  0.0250    50

Computed on 2000 event sequences
  Constraint           Value
      ageMin               0
      ageMax               5
   ageMaxEnd               5
 countMethod One by sequence
```

10.4 Identifying discriminant event subsequences

The function seqecmpgroup() identifies subsequences that discriminate significantly a group using the chi-square test. The subsequences are then ordered by decreasing discriminant power. The function takes at least the first two of the following arguments:

subseq: A "subseqelist" object containing the subsequences considered for discriminating the groups.

group: The variable that defines the groups.

method (optional): By default "chisq"; can be set to "bonferroni" to apply a Bonferroni correction to the p-value.

Function seqecmpgroup() returns a specific 'subseqelist' object that can be indexed, printed and plotted as any subsequence list. As an example, we look after the subsequences that discriminate the most birth cohorts. We first compute the list of frequent subsequences and create a cohort factor that distinguishes births before and after 1945. We then look for the most discriminating subsequences.

```
R> fsubseq <- seqefsub(bf.seqe, pMinSupport = 0.01)
R> cohort <- factor(biofam$birthyr > 1945, labels = c("<=1945",
+       ">1945"))
R> discrcohort <- seqecmpgroup(fsubseq, group = cohort, method = "bonferroni")
R> discrcohort[1:5]
```

```
                 Subsequence Support      p.value statistic index
1    (Parent)-(Parent>Left)   0.434 0.000000e+00 119.52974     2
2             (Parent>Left)   0.434 0.000000e+00 119.52974     3
3 (Parent)-(Parent>Married)   0.122 2.957312e-08  37.81393    20
4          (Parent>Married)   0.122 2.957312e-08  37.81393    21
5          (Left>Left+Marr)   0.234 4.036235e-07  32.72402     8
  Freq.<=1945 Freq.>1945 Resid.<=1945 Resid.>1945
1   0.3215941 0.56568947    -5.604735    6.066469
2   0.3215941 0.56568947    -5.604735    6.066469
3   0.1640408 0.07274701     3.953680   -4.279396
4   0.1640408 0.07274701     3.953680   -4.279396
5   0.1835032 0.29315961    -3.428990    3.711480

Computed on 2000 event sequences
  Constraint           Value
 countMethod One by sequence
```

10.4.1 Plotting the results

The results of the previous analysis can then be plotted (Figure 10.2) with the plot() function. In the resulting plot, the color of each bar is defined by the associated Pearson residual of the Chi-square test. For residuals ≤ -2 (dark red), the subsequence is significantly less frequent than expected under independence, while for residuals greater than 2 (dark blue), the subsequence is significantly more frequent.

```
R> plot(discrcohort[1:5])
```

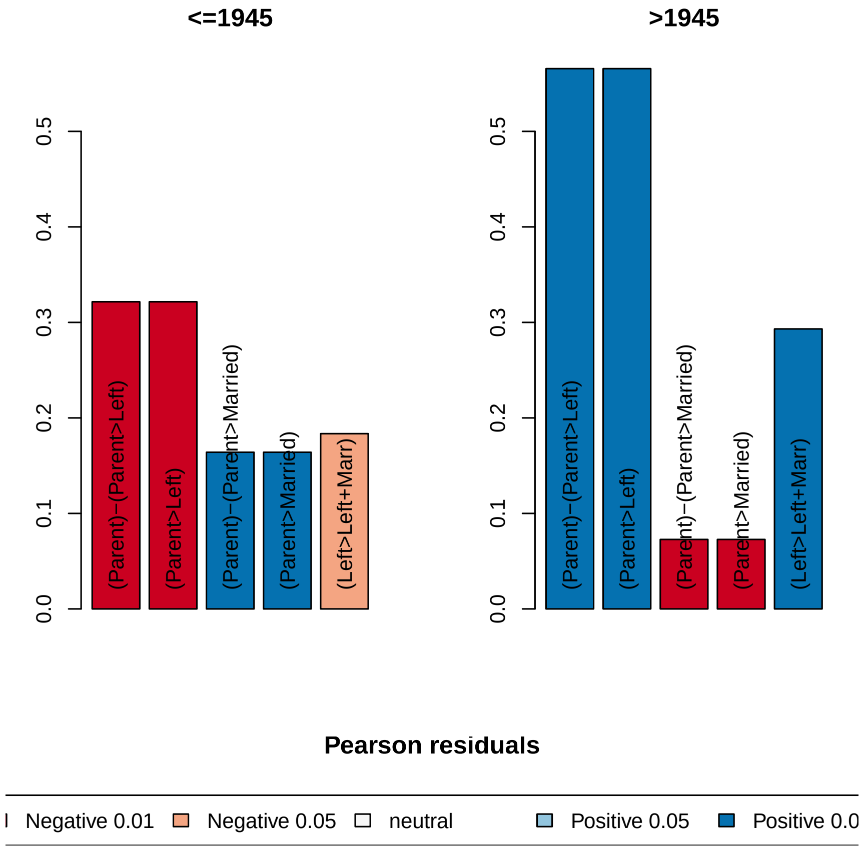


Figure 10.2: Five most discriminating event subsequences between those born before and after 1945.

10.5 More advanced topics and utilities

10.5.1 Looking after specific subsequences

We may want to search only for specific subsequences. For instance, we may be interested in individuals that experienced the following subsequences :

- (Parent) → (Left) → (LeftMarr): People leaving home, staying alone and getting married after that.

- (Parent) → (LeftMarr): People leaving home and getting married after that. This is a subsequence of the previous one.

The `seqefsub()` function can determine the frequency of specific subsequences. In order to do that, the subsequences must be encoded in the text format used for displaying the subsequences (see above). The previous subsequences would thus be encoded as follows:

```
R> mysubseqstr <- character(2)
R> mysubseqstr[1] <- "(Parent)-(Left)-(Left+Marr)"
R> mysubseqstr[2] <- "(Parent)-(Left+Marr)"
```

and here is how we get the frequency of these subsequences

```
R> mysubseq <- seqefsub(bf.seqestate, strsubseq = mysubseqstr)
R> print(mysubseq)
```

```
                 Subsequence Support Count
1         (Parent)-(Left+Marr)  0.4870   974
2 (Parent)-(Left)-(Left+Marr)  0.2275   455

Computed on 2000 event sequences
  Constraint           Value
 countMethod One by sequence
```

The result can be used as an argument of functions such as seqecmpgroup().

10.5.2 Counting the number of occurrence in each event sequence

We now use the preceding outcome to compute with the seqeapplysub() function the number of occurrences of each subsequence. The seqeapplysub() function takes three arguments: a list of subsequences (which includes the reference to the event sequences they were extracted from), a method and optionally a list of constraints. The method specifies the information we want. Possibilities are count (default), age, the age at first occurrence of a subsequence and presence which returns a matrix with ones indicating the presence of the subsequence and zero otherwise.

In the example below we count for each sequence how many time it contains each subsequence. The result is a matrix with rows corresponding to the sequences, columns to the specified subsequences and cell values equal to the requested counts.

```
R> msubcount <- seqeapplysub(mysubseq, method = "count")
R> msubcount[1:3, ]
```

```
                                                                  (Parent)-(Left+Marr)
(Parent)-9.00-(Left+Marr)-1.00-(Left+Marr+Child)-6.00                                 1
(Parent)-1.00-(Left)-10.00-(Left+Marr)-1.00-(Left+Marr+Child)-4.00                    1
(Parent)-7.00-(Left)-5.00-(Left+Marr)-1.00-(Left+Marr+Child)-3.00                     1
                                                                  (Parent)-(Left)-(Left+Marr)
(Parent)-9.00-(Left+Marr)-1.00-(Left+Marr+Child)-6.00                                        0
(Parent)-1.00-(Left)-10.00-(Left+Marr)-1.00-(Left+Marr+Child)-4.00                           1
(Parent)-7.00-(Left)-5.00-(Left+Marr)-1.00-(Left+Marr+Child)-3.00                            1
```

10.5.3 Selecting event subsequences

The function seqecontain() permits to select a set of event (sub)sequences containing specific events. It checks whether a given subsequence contains given events. For instance, we may want to select frequent subsequences containing one of the events “Parent>Married” or “Parent>Left+Marr”. The function returns a logical vector with TRUE/FALSE answer for each subsequence.

```
R> condition <- seqecontain(fsubseq, eventList = c("Parent>Married",
+       "Parent>Left+Marr"))
R> fsubseq[condition]
```

```
                                              Subsequence Support Count
1                            (Parent)-(Parent>Left+Marr)  0.2530   506
2                                     (Parent>Left+Marr)  0.2530   506
3 (Parent)-(Parent>Left+Marr)-(Left+Marr>Left+Marr+Child)  0.1495   299
4          (Parent>Left+Marr)-(Left+Marr>Left+Marr+Child)  0.1495   299
5                              (Parent)-(Parent>Married)  0.1220   244
6                                       (Parent>Married)  0.1220   244
7        (Parent)-(Parent>Left+Marr)-(Left+Marr>Divorced)  0.0105    21
8                 (Parent>Left+Marr)-(Left+Marr>Divorced)  0.0105    21

Computed on 2000 event sequences
  Constraint           Value
 countMethod One by sequence
```

To restrict the search to a subset of events, we may add the option `exclude=TRUE`. In this case, the function returns FALSE for any sequence that contains an event not specified in the `eventList` argument.

10.5.4 Duration of event sequences

It may be useful to set and retrieve the time span covered by an event sequence. We get the time span of an event sequence with the `seqelength()` function. There are two ways to set the duration. We can define an end-of-sequence event in which case the time span is the time until this event occurs. The end-of-sequence event is specified in `seqecreate()` with the `endEvent` option. Alternatively, we can set the total sequence duration explicitly with the `seqesetlength()` function.

```
R> bf.seqe[1:3]
```

```
[1] (Parent)-9.00-(Parent>Left+Marr)-1.00-(Left+Marr>Left+Marr+Child)-6.00
[2] (Parent)-1.00-(Parent>Left)-10.00-(Left>Left+Marr)-1.00-(Left+Marr>Left+Marr+Child)-4.00
[3] (Parent)-7.00-(Parent>Left)-5.00-(Left>Left+Marr)-1.00-(Left+Marr>Left+Marr+Child)-3.00
```

```
R> seqelength(bf.seqe[1:3])
```

```
[1] 16 16 16
```

```
R> sl <- numeric()
R> sl[1:2000] <- 14
R> seqesetlength(bf.seqe, sl)
R> seqelength(bf.seqe[1:3])
```

```
[1] 14 14 14
```

```
R> bf.seqe[1:3]
```

```
[1] (Parent)-9.00-(Parent>Left+Marr)-1.00-(Left+Marr>Left+Marr+Child)-4.00
[2] (Parent)-1.00-(Parent>Left)-10.00-(Left>Left+Marr)-1.00-(Left+Marr>Left+Marr+Child)-2.00
[3] (Parent)-7.00-(Parent>Left)-5.00-(Left>Left+Marr)-1.00-(Left+Marr>Left+Marr+Child)-1.00
```

Appendix A

Installing and using R

This appendix gives a short introduction to R. It explains where and how R can be obtained and describes its basic principles and operations. More detailed information can be found on the Comprehensive R-project Archive Network (CRAN) `http://www.r-project.org`. You may for instance download one the following introduction manual in pdf format `http://cran.r-project.org/doc/manuals/R-intro.pdf`. We also strongly recommend the introduction to R by Paradis (2005) available at `http://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf`.

A.1 Obtaining and installing R

R is a free integrated suite of software facilities for data manipulation, calculation and graphical display. It is available in precompiled binary form for Linux, MacOS X and Windows, and more generally in source form that can be compiled under many other operating systems. You can download R from the CRAN `http://cran.r-project.org/` (select a mirror close to you) where you find also installation instructions.

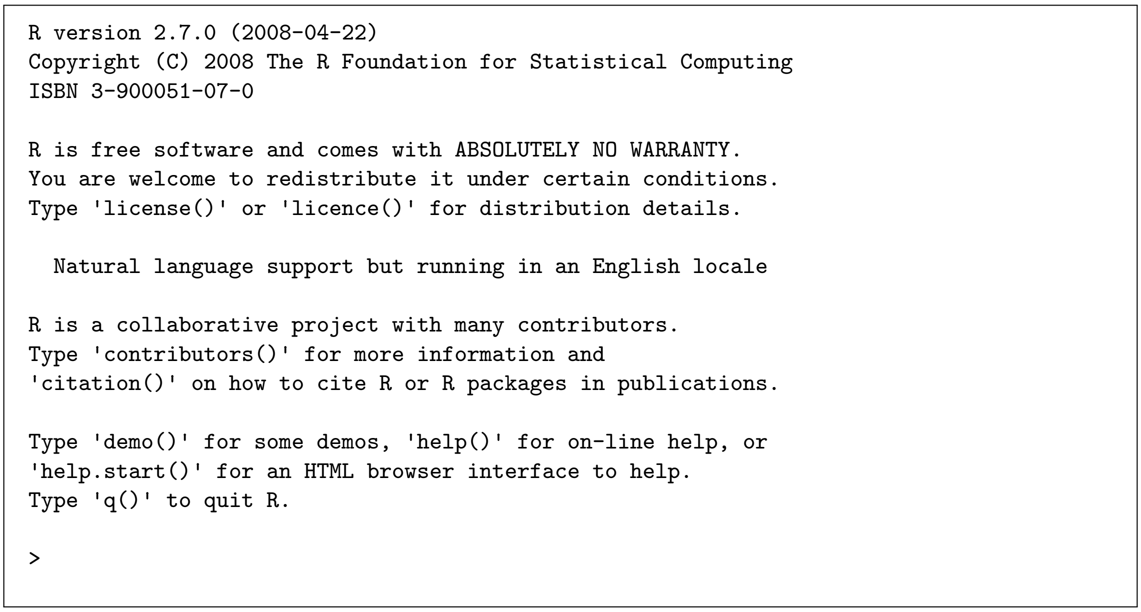
A.2 R basics

Starting R. Although there exist menu driven graphical user interfaces for R, R is originally a 'command line' environment. When starting R, you get a command line prompt (showed in a R console under Windows) at which you can enter commands.

If you are using Linux, just launch a terminal and enter 'R' at the command prompt. In Figure A.1 shows the screen display and command prompt as it appears after launching R in a Linux console. Here, the greeting message is in French because the authors of this manual run a French version of R.

To quit R, enter the command `q()`. You will be prompted for saving your workspace. Answer 'y' if you want to save all your data and objects. Your workspace will then be restored the next time you use R.

Writing and saving R program files. The best way of using R is to write command files. R command files usually have a '.R' extension. You can add comments in your program files. Starting with a double hashmark ('`##`'), everything to the end of the line is a comment. Under MacOS X and Windows, the R environment comes with a command editor that you can use to write, save and execute your programs. Under Linux, you have to resort to a separate editor such as `gedit` to `write` and save your programs. You may then copy/paste programs into the R console to run them or alternatively use the `source()` command.



```
R version 2.7.0 (2008-04-22)
Copyright (C) 2008 The R Foundation for Statistical Computing
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

Figure A.1: R starting welcome message and command prompt

Objects and functions Functions in R take one or more arguments.

Getting help Within R, you can get help about a function with the `help(function name)` command, including for all the functions provided by the TraMineR package. Try for instance the following

```
> help(seqdist)
```

A.3 Data manipulation in R

A.3.1 Creating and printing objects

The operator '`<-`' is used to assign a value to an R object and entering solely the name of the object prints its value (on the output screen). In the next example, we first create (or replace) the object '`x`' by assigning it the value 2 and then display its content

```
> x <- 2
> x
[1] 2
```

When printing the '`x`' object, the output contains '[1]' in front of the values of `x` indicating that the line begins with the first element of the object. In this case, it hasn't much interest because `x` has only one element. It may be useful, however, for objects containing more than one element, such as vectors, matrices or data frames that we describe hereafter.

A.3.2 Vectors

In R, vectors are very important. Even objects containing one single value are vectors

```
> z <- 4
> is.vector(z)
[1] TRUE
```

Creating vectors with `cbind()`. The widely used `c()` (or `cbind()`) function combines its arguments into a vector. In the following example we use this function to create a vector with the previously created ‘x’ and ‘z’ objects

```
> c(x,z)
[1] 2 4
```

Filling vectors with number sequences. It is often useful to generate a vector of consecutive numbers. This is easily done by using the sequence generating operator as shown in the following example.

```
> seq <- 1:50
> seq
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25
[26] 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
```

The content of the `seq` vector is printed in two lines, and the ‘`[26]`’ appearing in front of the second line indicates that the first element of this second line is 26th element the vector (here the value of the 26th element is 26).

A.3.3 Data frames, matrices and lists

In R, several object types are available apart from vectors. The object types we will have to deal with most of the time are data frames, matrices and lists. We briefly describe those objects and some hints for manipulating them.

Data frames. Since we haven’t yet introduced sequential data, we consider for illustrating purposes the classical *iris* data set that is distributed with R. We first load it into memory with the `data()` command and display its content by typing its name

```
> data(iris)
> iris
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1            5.1         3.5          1.4         0.2     setosa
2            4.9         3.0          1.4         0.2     setosa
3            4.7         3.2          1.3         0.2     setosa
4            4.6         3.1          1.5         0.2     setosa
5            5.0         3.6          1.4         0.2     setosa
6            5.4         3.9          1.7         0.4     setosa
7            4.6         3.4          1.4         0.3     setosa
8            5.0         3.4          1.5         0.2     setosa
9            4.4         2.9          1.4         0.2     setosa
10           4.9         3.1          1.5         0.1     setosa
.
.
.
141          6.7         3.1          5.6         2.4  virginica
142          6.9         3.1          5.1         2.3  virginica
143          5.8         2.7          5.1         1.9  virginica
144          6.8         3.2          5.9         2.3  virginica
145          6.7         3.3          5.7         2.5  virginica
146          6.7         3.0          5.2         2.3  virginica
147          6.3         2.5          5.0         1.9  virginica
148          6.5         3.0          5.2         2.0  virginica
149          6.2         3.4          5.4         2.3  virginica
150          5.9         3.0          5.1         1.8  virginica
```

This data set contains measurements about 150 iris flowers from 3 species, as we learn it by issuing the `help(iris)` command

```
> help(iris)
```

which produces in a separate window

```
iris                package:datasets                R Documentation

Edgar Anderson's Iris Data

Description:
     This famous (Fisher's or Anderson's) iris data set gives the
     measurements in centimeters of the variables sepal length and
     width and petal length and width, respectively, for 50 flowers
     from each of 3 species of iris.  The species are _Iris setosa_,
     _versicolor_, and _virginica_.
...
```

The `summary()` function returns basic statistics for all the variables in the data set

```
> summary(iris)
  Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
 Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
 1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
 Median :5.800   Median :3.000   Median :4.350   Median :1.300
 Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
 3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
 Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
         Species
   setosa    :50
   versicolor:50
   virginica :50
```

In R data frames, columns (variables) can be of mixed types. In the *iris* data set, the variables `Sepal.Length`, `Sepal.Width`, `Petal.Length` and `Petal.Width` are all numerical. The `summary()` function computes distribution indicators for them. On the other hand, 'Species' is a categorical variable. In R this variable type is called a *factor*, and the values a factor may take are called levels. The `Species` factor has three levels

```
> levels(iris$Species)
[1] "setosa"     "versicolor" "virginica"
```

Matrices. Matrices are multidimensional objects like data frames, however, they do not allow mixing data types. For example, if we try to transform the `iris` data frame into a matrix, all the elements, including numbers, will be converted to character strings, since one column of the data is of the character type. The function `as.matrix()` is used to convert the iris data frame into a matrix. There are a lot of similar functions in R for converting from one object type to another

```
> as.matrix(iris)
      Sepal.Length Sepal.Width Petal.Length Petal.Width Species
 [1,] "5.1"        "3.5"       "1.4"        "0.2"       "setosa"
 [2,] "4.9"        "3.0"       "1.4"        "0.2"       "setosa"
 [3,] "4.7"        "3.2"       "1.3"        "0.2"       "setosa"
 [4,] "4.6"        "3.1"       "1.5"        "0.2"       "setosa"
 [5,] "5.0"        "3.6"       "1.4"        "0.2"       "setosa"
 [6,] "5.4"        "3.9"       "1.7"        "0.4"       "setosa"
 [7,] "4.6"        "3.4"       "1.4"        "0.3"       "setosa"
 [8,] "5.0"        "3.4"       "1.5"        "0.2"       "setosa"
 [9,] "4.4"        "2.9"       "1.4"        "0.2"       "setosa"
[10,] "4.9"        "3.1"       "1.5"        "0.1"       "setosa"
...
```

Lists. A list is an object consisting of an ordered collection of objects. It is created with the `list()` command. The list below contains for instance three components.

```
> list.ex <- list(name="Alice", age=40, children.at=c(22,24,25))
> list.ex
$name
[1] "Alice"

$age
[1] 40

$children.at
[1] 22 24 25
```

We access a component by issuing for instance `list.ex$children.at` or, since we want here the 3rd component `list.ex[[3]]`.

A.3.4 Accessing and extracting data

Row and column names. Data frames and matrices have rows and column names (lists have elements names). The `rownames()` and `colnames()` functions can be used to access, modify or print these labels. The column names are correspond to what is known variable names in other statistical packages like Stata, SPSS or SAS.

```
> colnames(iris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

Row names are names assigned to the rows of the data object.

```
> rownames(iris)
  [1] "1"   "2"   "3"   "4"   "5"   "6"   "7"   "8"   "9"   "10"  "11"  "12"
 [13] "13"  "14"  "15"  "16"  "17"  "18"  "19"  "20"  "21"  "22"  "23"  "24"
.
.
.
[133] "133" "134" "135" "136" "137" "138" "139" "140" "141" "142" "143" "144"
[145] "145" "146" "147" "148" "149" "150"
```

Default row names are the row numbers, as illustrated above for the *iris* data set. Any character string can be used as row name. With the `paste()` command that concatenates its arguments into a character string, we may for instance create a vector of 150 names composed with the (French) string “fleur d’iris n.” and a number from 1 to 150, and assign this vector as row names

```
> row.names(iris) <- paste("fleur d'iris n.",1:150)
> iris
                    Sepal.Length Sepal.Width Petal.Length Petal.Width
fleur d'iris n. 1            5.1         3.5          1.4         0.2
fleur d'iris n. 2            4.9         3.0          1.4         0.2
fleur d'iris n. 3            4.7         3.2          1.3         0.2
fleur d'iris n. 4            4.6         3.1          1.5         0.2
fleur d'iris n. 5            5.0         3.6          1.4         0.2
fleur d'iris n. 6            5.4         3.9          1.7         0.4
fleur d'iris n. 7            4.6         3.4         1.4         0.3
fleur d'iris n. 8            5.0         3.4         1.5         0.2
fleur d'iris n. 9            4.4         2.9         1.4         0.2
fleur d'iris n. 10           4.9         3.1         1.5         0.1
...
```

Indexing rows and columns. Elements of an R data frame or matrix is accessed by specifying the row and/or column index. One solution is to give the row and column numbers as indexes. The following command accesses the sepal length (first column) of the first iris flower (first row) from the iris data set

```
> iris[1,1]
[1] 5.1
```

Alternatively, we may use the row and column names. The following example is equivalent to the previous command

```
> iris[1,"Sepal.Length"]
[1] 5.1
```

It is also possible to use previously created row names

```
> iris["fleur d'iris n. 1","Sepal.Length"]
[1] 5.1
```

In R, there are almost as many ways of doing a same thing as there are stars in the universe. An additional possibility is for instance to extract the first column with the $ operator and to specify the first element of the resulting vector

```
> iris$Sepal.Length[1]
[1] 5.1
```

For accessing more than one element, we can use the number sequence generating mechanism. For example, we display the first 10 rows of the `iris` data set by issuing the following command in which the missing second argument means that all columns should be selected.

```
> iris[1:10,]
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1           5.1         3.5          1.4         0.2  setosa
2           4.9         3.0          1.4         0.2  setosa
3           4.7         3.2          1.3         0.2  setosa
4           4.6         3.1          1.5         0.2  setosa
5           5.0         3.6          1.4         0.2  setosa
6           5.4         3.9          1.7         0.4  setosa
7           4.6         3.4          1.4         0.3  setosa
8           5.0         3.4          1.5         0.2  setosa
9           4.4         2.9          1.4         0.2  setosa
10          4.9         3.1          1.5         0.1  setosa
```

A.4 R libraries

When launching R, you have access to a set of basic functions. You may access additional and more sophisticated functions by explicitly loading add-on packages with the `library()` function. Some of these add-on packages (libraries) may be installed by default on your system, which is for example the case of the `foreign` library for importing data sets stored in various formats such as Stata, SAS or SPSS. In this case, you just have to issue

```
> library(foreign)
```

to access the functions provided by the package.

In order to use add-on packages (like TraMineR) that are not installed, you need indeed to first install them on your system. A large number of official and contributed add-on packages are available on the CRAN http://cran.r-project.org/src/contrib/PACKAGES.html. For installing any of these packages, you can just issue an `install.packages()` command

```
> install.packages("package_name")
```

within an R console and choose a mirror close to you in the menu.

For installing other packages that are not distributed through the CRAN (like TraMineR for the moment), you have to get the package source or binary file and install it manually as described in Chapter 3. Once the package is installed, you will be able to access its functions after issuing the suited `library()` command, e.g.

```
> library(TraMineR)
```

A.5 Some other useful functions

A.5.1 The apply function

The `apply` function permits to apply a function to every row (or every column) of a matrix or data frame. This is a very useful function.

In the example below we create a 3 × 4 table by combining the three rows of length 4. We then compute the mean value of each column (the 2nd dimension) and then of each row (the 1st dimension).

```
> mat <- rbind(c(1,3,5,4),c(2,3,1,5),c(2,6,3,1))
> mat
     [,1] [,2] [,3] [,4]
[1,]    1    3    5    4
[2,]    2    3    1    5
[3,]    2    6    3    1
> apply(mat,2,mean)
[1] 1.666667 4.000000 3.000000 3.333333
> apply(mat,1,mean)
[1] 3.25 2.75 3.00
```

A.5.2 The table function

For factor variables, i.e. categorical variables, the `table()` command gives the count of each of its value. As seen before, the `$` operator followed by the column name permits to extract the corresponding column from a data frame or matrix. In the next example we tabulate the `Species` variable with the `table()` function

```
> table(iris$Species)
    setosa versicolor  virginica
        50         50         50
```

A.6 Creating and saving graphics

The `pdf()` and `ps()` commands open a '.pdf' or '.postscript' file that will contain all the graphics plotted with plots commands (eg. `plot()`. The `dev.off()` must be used to close the file. The next example shows how to store an histogram of 1000 random generated numbers drawn from the normal distribution in the *myplot.pdf* file.

```
> pdf(file="**location**/myplot.pdf")
> hist(rnorm(1000))
> dev.off()
```

There are a lot fine tuning parameters that can be used to set the output page size, font sizes, etc. Check the available options with `?pdf` or `?ps`. Note that there are similarly `png()`, `jpeg()`, `tiff()` and some other functions for producing graphics in other formats.

A.7 Performance and memory usage

In R, objects are stored in memory. The size and number of objects you can handle is limited by the memory size. If you don't further need an object, you can free memory by deleting it with the command `rm(objectname)`. For example, a sequence data containing 4318 rows of 16 states needs 0.52 Mb (541kb).

Appendix B

Information about TraMineR content

Below we show the content of the information window obtained with `library(help=TraMineR)`. This information indicates among others the version number of the installed TraMineR package and the list of available functions and data sets. Indeed, since further versions of TraMineR will most probably offer new features we strongly recommend that you check the updated information window on your system after installing a new version.

```
Description:

Package:            TraMineR
Version:            1.4
Date:               2009-08-06
Title:              Sequences and trajectories mining for social
                    scientists
Author:             Alexis Gabadinho <alexis.gabadinho@unige.ch>,
                    Matthias Studer <matthias.studer@unige.ch>, Nicolas
                    S. Muller <nicolas.muller@unige.ch>, Gilbert
                    Ritschard <gilbert.ritschard@unige.ch>.
Maintainer:         Alexis Gabadinho <alexis.gabadinho@unige.ch>
Depends:            R (>= 2.7.1), RColorBrewer, boot
Suggests:           cluster
Description:        This package is a toolbox for sequence
                    manipulation, description, rendering and more
                    generally sequence data mining in the field of
                    social sciences. Though it is primarily intended
                    for analyzing state or event sequences that
                    describe life courses such as family formation
                    histories or professional careers its features
                    apply indeed also to many other kinds of
                    categorical sequence data. It accepts as input many
                    different sequence representations and provides
                    tools for translating sequences from one format to
                    another. It offers several statistical functions
                    for describing and rendering sequences, for
                    computing distances between sequences with
                    different metrics among which optimal matching, the
                    longest common prefix and the longest common
                    subsequence, and simple functions for extracting
                    the most frequent subsequences and identifying the
                    most discriminating ones among them. A user's guide
                    can be found on TraMineR's web page.
License:            GPL (>= 2)
URL:                http://mephisto.unige.ch/traminer
Packaged:           2009-08-06 10:21:30 UTC; hornik
Repository:         CRAN
Date/Publication:   2009-08-06 10:54:02
Built:              R 2.9.1; i386-pc-mingw32; 2009-08-06 19:40:20 UTC;
                    windows


Index:

TraMineR.checkupdates   Check for updates
actcal                  Example data set: Activity calendar from the
                        Swiss Household Panel
actcal.tse              Example data set: Activity calendar from the
                        Swiss Household Panel (time stamped event
                        format)
alphabet                Get or set the alphabet of a sequence object
biofam                  Example data set: Family life states from the
                        Swiss Household Panel biographical survey
cpal                    Get or set the color palette of a sequence
                        object
dissassoc               Analysis of discrepancy based on dissimilarity
                        measure
disscenter              Compute distance to the center of a group
dissmfac                Multi-factor ANOVA from a dissimilarity matrix
disstree                Dissimilarity Tree
disstree2dot            Graphical representation of a dissimilarity
                        tree
disstreeleaf            Terminal node appartenance
dissvar                 Dissimilarity based discrepancy
ex1                     Example data set with missing values and
                        weights
famform                 Example data set: sequences of family formation
mvad                    Example data set: Transition from school to
                        work
plot.stslist            Plot method for state sequence objects
plot.stslist.freq       Plot method for sequence frequency tables
plot.stslist.meant      Plot method for objects produced by the
                        seqmeant function
plot.stslist.modst      Plot method for modal state sequences
plot.stslist.rep        Plot method for representative sequence sets
plot.stslist.statd      Plot method for objects produced by the
                        seqstatd function
plot.subseqelist        Plot frequencies of subsequences
plot.subseqelistchisq   Plotting discriminant subsequences
read.tda.mdist          Read a distance matrix produced by TDA.
seqLLCP                 Compute the length of the longest common prefix
                        of two sequences
seqLLCS                 Compute the length of the longest common
                        subsequence of two sequences
seqST                   Sequences turbulence
seqcomp                 Compare two sequences
seqconc                 Concatenate vectors of states or events into a
                        character string
```

seqdecomp	Convert a character string into a vector of states or events
seqdef	Create a state sequence object
seqdiff	Decompose the difference between groups of sequences
seqdim	Returns the dimension of a set of sequences
seqdist	Distances between sequences
seqdistmc	Multichannel distances between sequences
seqdss	Extract distinct states sequence from a sequence object
seqdur	Extracts states durations from a sequence object.
seqeapplysub	Checking if event sequences contain given subsequences
seqecmpgroup	Identifying discriminating subsequences
seqeconstraint	Setting time constraint
seqecontain	Check if sequence contains events
seqecreate	Create event sequence objects.
seqefsub	Searching for frequent subsequences
seqeid	Retrieve id of an event sequence object.
seqelength	Length of event sequences
seqetm	Create a transition-definition matrix
seqfind	Find the occurrences of sequence(s) x in the set of sequences y
seqformat	Translation between sequence formats
seqfpos	Search for the first occurrence of a given element in a sequence
seqgen	Random sequences generation
seqient	Within sequences entropy
seqistatd	States frequency for each individual sequence
seqlegend	Plot a legend for the states in a sequence object
seqlength	Sequence length
seqlogp	Computing the logarithm of sequences probabilities
seqmeant	Mean durations in each state
seqmodst	Sequence of modal states
seqmpos	Number of matching positions between two sequences.
seqnum	Translate a sequence object's alphabet into numerical alphabet, ranging 0-(nbstates-1).
seqplot	Plot functions for state sequence objects
seqpm	Find patterns in sequences
seqrep	Extracting sets of representative sequences
seqsep	Adds separators to sequences stored as character string
seqstatd	Sequence of transversal state distributions and their entropies
seqstatf	State frequencies in the all sequence data set
seqstatl	List of distinct states or events (alphabet) in a sequence data set.
seqsubm	Create a substitution-cost matrix
seqsubsn	Number of distinct subsequences in a sequence.
seqtab	Frequency table of the sequences
seqtrate	Compute transition rates between states
seqtree2dot	Graphical representation of a dissimilarity

```
tree
```

Bibliography

Aassve, A., F. Billari, and R. Piccarreta (2007). Strings of adulthood: A sequence analysis of young british women's work-family trajectories. *European Journal of Population 23*(3), 369–388.

Abbott (2001). *Time Matters. On Theory and Methods.* Chicago: Chicago Press.

Abbott, A. and J. Forrest (1986). Optimal matching methods for historical sequences. *Journal of Interdisciplinary History 16*, 471–494.

Agrawal, R. and R. Srikant (1995). Mining sequential patterns. In P. S. Yu and A. L. P. Chen (Eds.), *Proceedings of the International Conference on Data Engeneering (ICDE), Taipei, Taiwan*, pp. 487–499. IEEE Computer Society.

Billari, F. C. (2001). The analysis of early life courses: complex descriptions of the transition to adulthood. *Journal of Population Research 18*(2), 119(24)–.

Brzinsky-Fay, C., U. Kohler, and M. Luniak (2006). Sequence analysis with Stata. *The Stata Journal 6*(4), 435–460.

Elzinga, C. and A. Liefbroer (2007). De-standardization of family-life trajectories of young adults: A cross-national comparison using sequence analysis. *European Journal of Population/Revue européenne de Démographie 23*(3), 225–250.

Elzinga, C. H. (2006). Turbulence in categorical time series. *Mathematical Population Studies (submitted).*

Elzinga, C. H. (2007). *CHESA 2.1 User Manual.* Amsterdam: Vrije Universiteit.

Elzinga, C. H. (2008). Sequence analysis: Metric representations of categorical time series. *Sociological Methods and Research.* forthcoming.

Fussell, E. (2005). Measuring the early adult life course in Mexico: An application of the entropy index. In R. Macmillan (Ed.), *The Structure of the Life Course: Standardized? Individualized? Differentiated?*, Advances in Life Course Research, Vol. 9, pp. 91–122. Amsterdam: Elsevier.

Gauthier, J.-A. (2007). *Empirical categorizations of social trajectories: A sequential view on the life course.* thèse, Université de Lausanne, Faculté des sciences sociales et politique (SSP), Lausanne.

Giele, J. and G. Elder (Eds.) (1998). *Methods of Life Course Research. Qualitative and Quantitative Approaches.* CA, Sage: Thousand Oaks.

Haubold, B. and T. Wiehe (2006). *Introduction to computational biology An evolutionary approach.* Birkhäuser Verlag.

Levenshtein, V. (1966). Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady 10*, 707–710.

McVicar, D. and M. Anyadike-Danes (2002). Predicting successful and unsuccessful transitions from school to work by using sequence methods. *Journal of the Royal Statistical Society. Series A (Statistics in Society) 165*(2), 317–334.

Müller, N. S., M. Studer, and G. Ritschard (2007). Classification de parcours de vie à l'aide de l'optimal matching. In *XIVe Rencontre de la Société francophone de classification (SFC 2007), Paris, 5 - 7 septembre 2007*, pp. 157–160.

Needleman, S. and C. Wunsch (1970). A general method applicable to the search for similarities in the amino acid sequence of two proteins. *Journal of Molecular Biology 48*, 443–453.

Notredame, C., P. Bucher, J.-A. Gauthier, and E. D. Widmer (2006). T-COFFEE/SALTT: User guide and reference manual. Technical report, CNRS Marseille and PAVIE University of Lausanne. (available at http://www.tcoffee.org/saltt/).

Paradis, E. (2005). *R pour les débutants.* F-34095 Montpellier: Institut des Sciences de l'Evolution Université Montpellier II.

Rohwer, G. and U. Pötter (2002). TDA user's manual. Software, Ruhr-Universität Bochum, Fakultät für Sozialwissenschaften, Bochum.

Scherer, S. (2001). Early career patterns: A comparison of Great Britain and West Germany. *European Sociological Review 17*(2), 119–144.

Studer, M., A. Gabadinho, N. S. Müller, and G. Ritschard (2008). Approches de type n-grammes pour l'analyse de parcours de vie familiaux. *Revue des nouvelles technologies de l'information RNTI E-11, II*, 511–522.

Zaki, M. J. (2001). SPADE: An efficient algorithm for mining frequent sequences. *Machine Learning 42*(1/2), 31–60.

Index