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

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/

July 10, 2008
(For version 1.0)

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

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 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, 2008. (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 expect, it was also inspired by the authors' taste for Gewurztraminer wine. This manual 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 datasets and R scripts (sequence of R-commands) [1] are included in the TraMineR distribution package. 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. Appendix B explains how to install TraMineR in R, while Chapter 2 shortly explains how to use the package and describes the illustrative datasets provided with it.

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.
- 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

[1] R demo scripts named *Describing_visualizing*, *Similarities* and *Event_sequences* 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.

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

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.

1.1 Preliminary remarks about sequences

In the social sciences, sequence data represent typically longitudinal biographical data such as employment histories or family life courses. Brzinsky-Fay et al. (2006) define for instance a *sequence* as an ordered list of states (employed/unemployed) or events (leaving parental home, marriage, having a child).

For a more 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 $x = x_1 x_2 \dots x_k$ 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*.

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 $x = S, U, M, MC$.

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.

All these various representations will be discussed in more details in Chapter 3. For now let us just retain that there are multiple ways of presenting time stamped sequence data and that TraMineR will prove useful for converting from one form to the other.

1.2 A short example to begin with

Nothing is better than an example to present the features of TraMineR . We will use for this purpose a data set from McVicar and Anyadike-Danes (2002) which is freely downloadable from the internet (see Section 2.2). The data contains 72 monthly activity state variables from July 1993 to June 1999 for 712 individuals. We suppose that the file has been downloaded and imported into R (see Section 2.2), that the name of the data frame is *mvad*, and that the TraMineR library has been loaded with the `library(TraMineR)` command. Some results of the following commands are presented in Figure 1.1.

1. Define a vector containing the legends for the states to appear in the graphics and create a sequence object which will be used as argument to the next functions

```
mvad.lab <- c("school", "FE", "employment","training",
              "joblessness", "HE")
mvad.seq <- seqdef(mvad, 15:86, labels=mvad.lab)
```

2. Plot of the 10 first sequences. Result in Fig. 1.1(a).

```
seqiplot(mvad.seq, withlegend=F)
```

3. Plot all sequences sorted by variable 'gcse5eq', a binary dummy indicating qualifications gained by the end of compulsory education: 1 if 5 or GCSEs at grades A-C or equivalent, 0 otherwise. Result in Fig. 1.1(b)).

```
seqiplot(mvad.seq, tlim=0, sortv=mvad$gcse5eq, withborder=F, space=0,
         withlegend=F)
```

4. Draw the sequence frequency plot of the 10 most frequent sequences with bar width proportional to the frequencies. Result in Fig. 1.1(c).

```
seqfplot(mvad.seq, pbarw=T, withlegend=F)
```

5. Plot the state distribution by time points. Result in Fig. 1.1(d).

```
seqdplot(mvad.seq, withlegend=F)
```

6. Plot a single legend as a separate graphic since several plots use the same color codes for the states. Result in Fig. 1.1(e).

```
seqlegend(mvad.seq)
```

7. Compute, summarize and plot the histogram of the sequence turbulences. Result in Fig. 1.1(f).

```
mvad.turb <- seqST(mvad.seq)
summary(mvad.turb)
hist(mvad.turb, col="cyan")
```

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

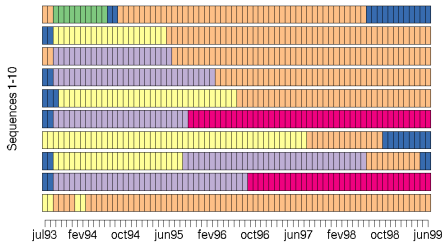
```
submat   <- seqsubm(mvad.seq, method= "TRATE")
dist.om1 <- seqdist(mvad.seq, method="OM", indel=1, sm=submat)
```

9. 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 3 class solution. We do not show here the dendrogram produced by `plot(clusterward1)` which, indeed, is not a TraMineR feature.

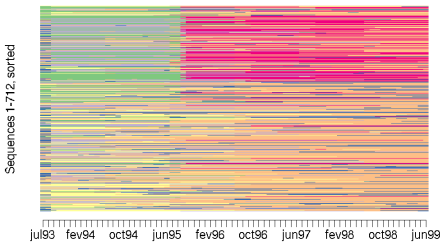
```
library(cluster)
clusterward1 <- agnes(dist.om1, diss=TRUE, method="ward")
plot(clusterward1)
cl1.3 <- cutree(clusterward1, k=3)
```

10. Plot the state distribution within each cluster. Result in Fig. 1.2.

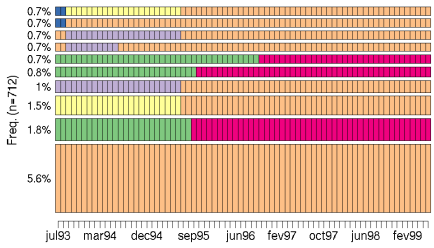
```
par(mfrow=c(2,2))
seqdplot(mvad.seq[cl1.3 == 1,], withlegend=F, title="Type 1")
seqdplot(mvad.seq[cl1.3 == 2,], withlegend=F, title="Type 2")
```



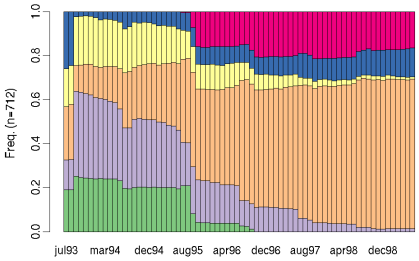
(a) Index plot of first 10 sequences



(b) Index plot of all sequences sorted by 'gcse5eq'

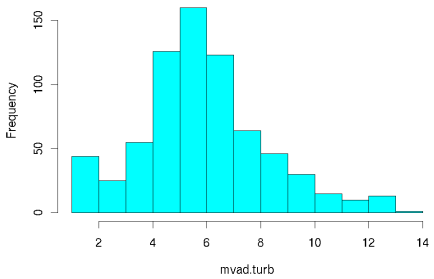


(c) Frequency plot of 10 most frequent sequences



(d) Distribution plot by time point

(e) Legend



(f) Histogram of sequence turbulence

Figure 1.1: A short example - Data from McVicar and Anyadike-Danes (2002)

```
seqdplot(mvad.seq[cl1.3 == 3,], withlegend=F, title="Type 3")
seqlegend(mvad.seq)
```

11. Plot the 10 most frequent sequences of each cluster. Result in Fig. 1.3.

```
par(mfrow=c(2,2))
seqfplot(mvad.seq[cl1.3 == 1,], pbarw=T, withlegend=F, title="Type 1")
seqfplot(mvad.seq[cl1.3 == 2,], pbarw=T, withlegend=F, title="Type 2")
seqfplot(mvad.seq[cl1.3 == 3,], pbarw=T, withlegend=F, title="Type 3")
seqlegend(mvad.seq)
```

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:

12. Define the sequences of transitions, put them in vertical time stamped event form and then turn them into an event sequence object.

```
transition <- seqetm(mvad.seq, method="transition")
mvad.tse <- seqformat(mvad, var=15:86, from='STS', to='TSE',
  tevent=transition)
mvad.seqe <- seqecreate(id=mvad.tse$id, time=mvad.tse$time,
  event=mvad.tse$event)
```

13. Look for frequent event subsequences and plot the 15 most frequent ones. Result in Fig. 1.4.

```
fsubseq <- seqefsub(mvad.seqe,pMinSupport=0.05)
seqefplot(fsubseq$subseq[1:15], mvad.seqe, col="cyan")
```

14. Determine the most discriminating transitions between clusters and plot the frequencies by cluster of the 6 first ones. Result in Fig. 1.5.

```
discr <- seqecmpgroup(fsubseq$subseq, mvad.seqe,cl1.3)
seqefplot(fsubseq$subseq[discr$index[1:6]], mvad.seqe,
    group=cl1.3, mfrow=c(2,3), col="cyan")
```

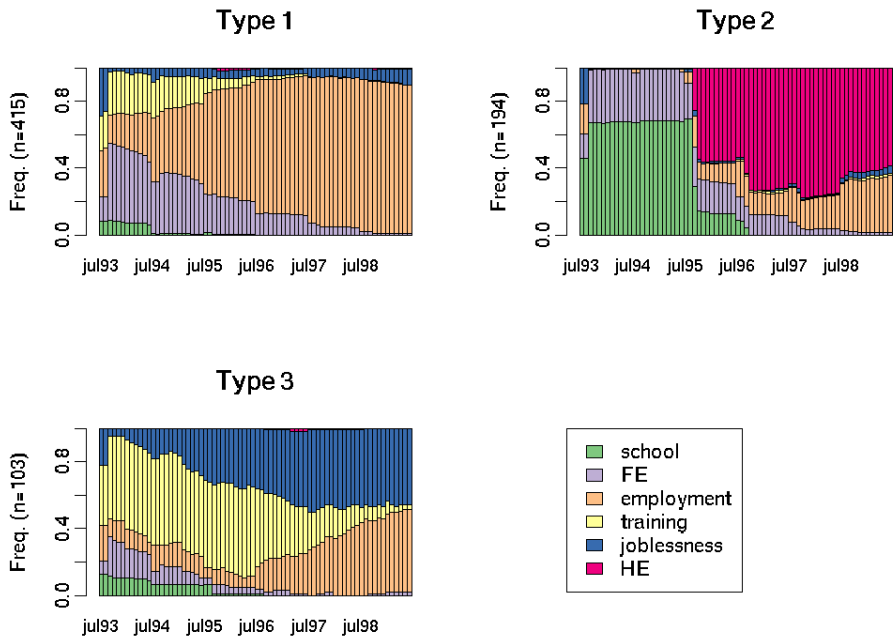


Figure 1.2: A short example - State distribution within each cluster - Data from McVicar and Anyadike-Danes (2002)

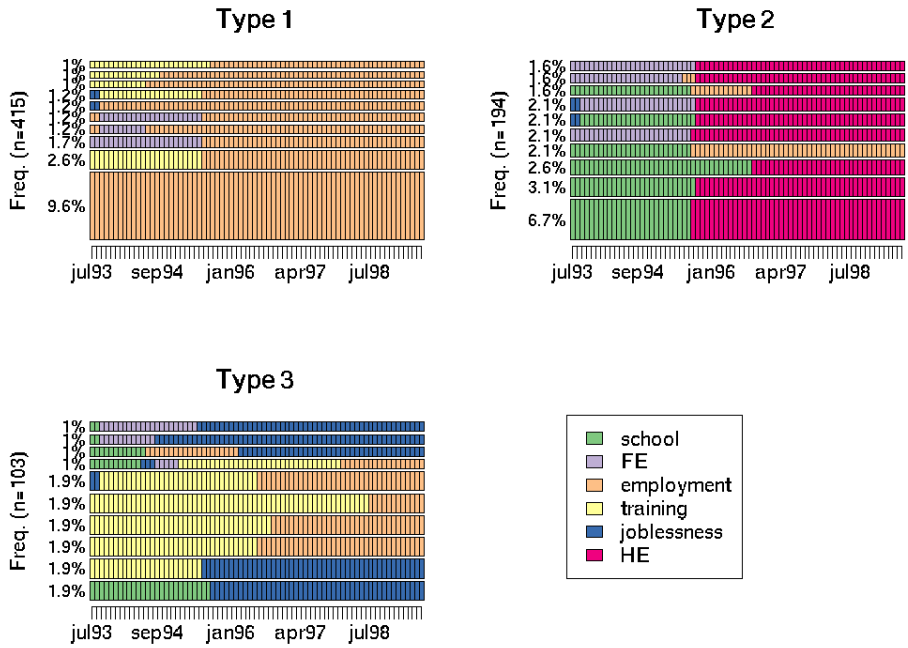


Figure 1.3: A short example - State frequencies within each cluster - Data from McVicar and Anyadike-Danes (2002)

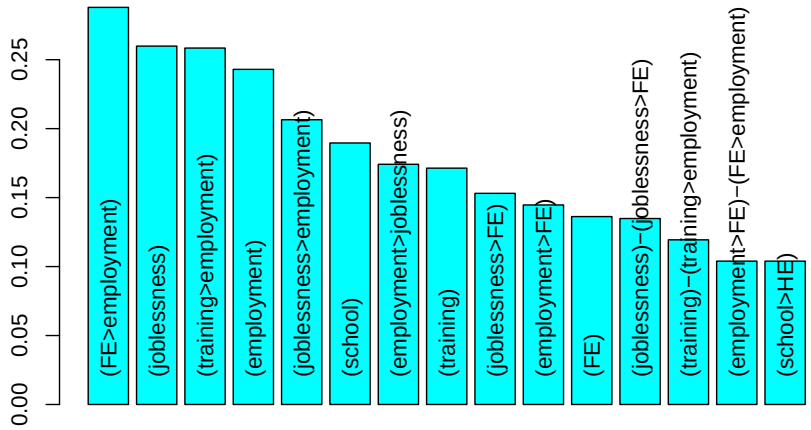


Figure 1.4: A short example - Frequencies of most frequent transitions - Data from McVicar and Anyadike-Danes (2002)

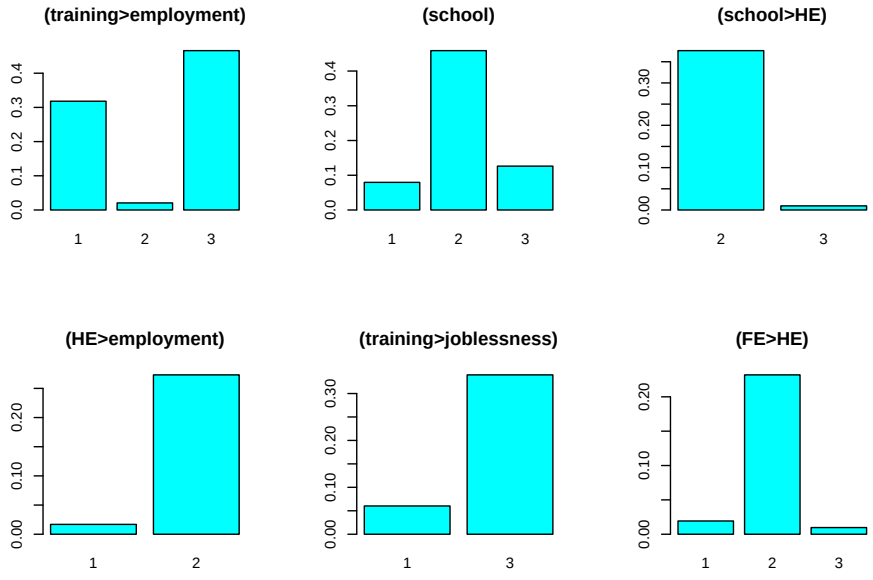


Figure 1.5: A short example - Most discriminating transitions between clusters - Data from McVicar and Anyadike-Danes (2002)

Chapter 2

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.

TraMineR can be installed either from a precompiled binary package or from source files. The latest versions for Linux (32 and 64 bits), Mac OS/X and Windows are available at http://mephisto.unige.ch/pub/traminer/ or directly from the CRAN http://cran.r-project.org/. For more detail on how to install TraMineR, see Appendix B p. 91.

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.

2.1 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

```
> 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

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

The above command opens a help window. The content of the obtained help window is shown in Appendix C. Since this appendix will probably be outdated for further releases of the package, we recommend that you preferably check the help about the TraMiner library on your system.

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 4 and 7) 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 4). 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:

```
> data(actcal)
> actcal.seq <- seqdef(var=13:24,data=actcal)
 [>] distinct states appearing in the data: A/B/C/D
 [>] alphabet: 1=A 2=B 3=C 4=D
 [>] 2000 sequences in the data set
 [>] min/max sequence length: 12/12
```

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

```
> seqdef(13:24, actcal)
Error in subset.default(data, , var) :
  argument "subset" is missing, with no default
```

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

```
> ?seqdef
```

or

```
> help(seqtab)
```

2.2 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*

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

data sets were created from the Swiss Household Panel[2], SHP, data (http://www.swisspanel.ch/.)

2.2.1 The *actcal* data set

Figure 2.1 shows how to load the *actcal* data set, listing the names of its columns and displaying the content of columns 13 to 24 (that contain the sequence data) in the 10 first records. You may get an overview and summary statistics of the whole *actcal* data set by issuing the `summary(actcal)` command (output not shown). This

```
> data(actcal)
> 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"
> actcal[1:10,13:24]
     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
2905     D     D     D     D     D     D     D     D     D     D     D     D
106      A     A     A     A     A     A     A     A     A     A     A     A
5113     A     A     A     A     A     A     A     A     A     A     A     A
4503     A     A     A     A     A     A     A     A     A     A     A     A
```

Figure 2.1: The sequences in the first 10 rows of the *actcal* data set

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 2.1. 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 2.2. 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

[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.

Table 2.1: 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 2.2: 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

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. 87).

2.2.2 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 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 2.3. The list of variables is shortly described in Table 2.4.

Table 2.3: 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 2.4: 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

2.2.3 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.

The *MVAD* data set is the data set used and described by McVicar and Anyadike-Danes (2002). It is used by Elzinga (2007) for illustrating his CHESA software (http://home.fsw.vu.nl/ch.elzinga/). It can be freely downloaded from http://www.blackwellpublishing.com/rss/Volumes/Av165p2.htm and converted into an R data set with the following steps:

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

   ```
   > mvad <- read.csv(file="/usr/local/temp/McVicar.csv",header=TRUE)
   ```

   where you should indeed adapt the path "/usr/local/temp/" to the '.csv' file.

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 2.5.

2.3 Performance and memory usage

Depending on your system and the size of your data, some functions for sequence data analysis may take some time, especially the computation of distances between

Table 2.5: 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	"

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. To give an idea, computing optimal matching distances for the 4318 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.

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.

Chapter 3

Definition and representation of sequence data

Broadly, sequences are ordered lists of states or events. However, sequences representation in data files can vary a lot, depending on the way data were collected and the way information is organized. In some cases, sequences are not explicitly defined but can be constructed from data originally collected as spells or time stamped events.

3.1 Ontology

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

3.1.1 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 2.3 page 19. Conversion between state sequences and event sequences is therefore not straightforward.

Figure 3.1 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, no event occurred during the observation period since the respondent stays in the same state during the whole sequence. In sequence 9, two events occurred:

- 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

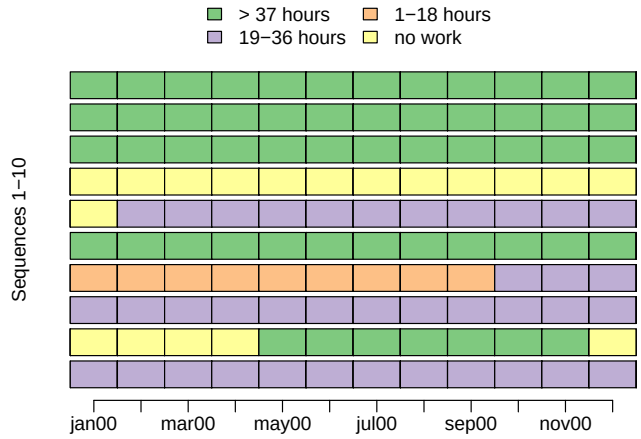


Figure 3.1: First 10 sequences of the *actcal* data

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.

```
> sps
    STS formatted sequence
[1] "000-000-000-000-000-000-000-000-000-000-000-000-0W0-0W0-0W0-0W0-0W0
-0W0-0W0-0W0-0W0-0WU-0WU-0WU-0WU-0WU-1WU-1WU"
[2] "000-000-000-000-000-000-000-000-000-000-000-000-0W0-0W0-0W0-0W0-0W0
-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-1WU-1WU"
```

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, 1=working) and the third character for the union status (0=not in union, 1=in union). The alphabet contains 16 distinct states (see Table 1, page 376 in Aassve et al., 2007).

3.1.2 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*.

3.1.3 Time reference: Internal and external clocks

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).

3.1.4 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.

3.1.5 Ontology

An ontology of sequence data formats can be defined by a nested suite of 'yes/no' questions about properties of the format. Figure 3.2 shows an ontology of types of longitudinal data, i.e. data organized according to time.

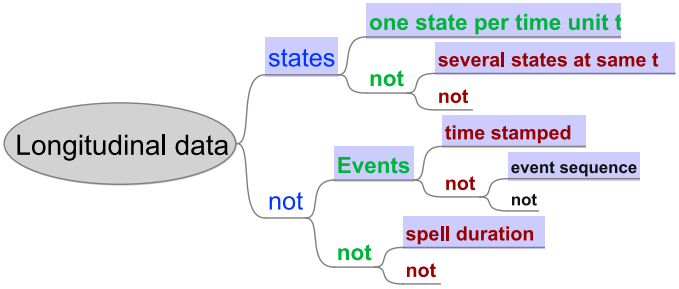


Figure 3.2: Ontology of types of longitudinal data

3.2 Identifying and defining some (common) data formats

Using some elements of the ontology, Table 3.1 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. These formats are further described in this section.

3.2.1 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. 2.2.1), sequences are in columns 13 to 24 representing the monthly activity statuses from January to December 2000.

```
> actcal[1:10,13:24]
     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
2905     D     D     D     D     D     D     D     D     D     D     D     D
106      A     A     A     A     A     A     A     A     A     A     A     A
5113     A     A     A     A     A     A     A     A     A     A     A     A
4503     A     A     A     A     A     A     A     A     A     A     A     A
```

3.2.2 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.

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

This format is an alternative way of representing 'STS' sequences. Here are the same sequences in compressed 'STS' format

```
000-000-000-000-000-000-000-000-000-000-000-000-0W0-0W0-0W0-0W0-0W0-0W0-
0W0-0W0-0W0-0WU-0WU-0WU-0WU-0WU-1WU-1WU
000-000-000-000-000-000-000-000-000-000-000-000-0W0-0W0-0W0-0W0-0W0-0W0-
0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-1WU-1WU
```

3.2.3 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 4.3.3 page 41). Each record of the TSE representation usually contains a case

Table 3.1: Sequence data representations

Data type	Code	(S)tates or (E)vents	One (1) or several (M) rows for each individual	Import into a sequence object	Example
State-sequence	STS	S	1	Yes	**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
State-permanence (1)	SPS1	S	1	Yes	**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)
State-permanence (2)	SPS2	S	1	Yes	**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
Distinct-State-Sequence	DSS	S	1	Yes (use STS)	**Id State 1 State 2 State 3 State 4 State 5** 101 S M MC D 102 S MC
Time-stamped event	TSE	E	M	Yes (event sequence)	**id time event** 101 21 Marriage 101 23 Child 101 27 Divorce 102 21 Marriage 102 21 Child
Spell	SPELL	S	M	Yes	**id index from to status** 101 1 18 20 Single 101 2 21 22 Married 101 2 23 26 Married w Children 101 3 27 .. Divorced 102 1 18 20 Single 102 2 21 27 Married w Children
Person-period			M		

Table 3.2: 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)

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.

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

3.2.4 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 3.2. 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).

```
idpers index from until status
  2713     1 1965  1989      1
  2713     2 1989  1990      5
  2713     3 1990  1991      5
  2713     4 1991  2002      5
  2714     1 1968  1985      1
  2714     2 1985  1988      7
  2714     3 1989  1990      5
  2714     4 1990  1991      5
```

[1] Original personal identification numbers have been modified.

```
 2714     5 1991  2002      5
 3713     1 1961  1978      1
 3713     2 1978  1983      3
 3713     3 1983  1984      4
 3713     4 1984  1985      3
 3713     5 1985  1999      4
 3713     6 1999  2001      7
11714     1 1973  1993      1
11714     2 1993  2002      5
```

3.2.5 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 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.

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

This data presentation is intended for mobility analysis where the concern is the transition from the state 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:

```
> seqs <- data.frame(y2000="A", y2001="A", y2002="C", y2003="D", y2004="D")
> seqformat(seqs,from="STS", to="SRS")
 => converting sequence data from STS to SRS
 => 1 rows/sequences converted to internal (STS) format
 => SRS formatted output has 5 rows
     id idx  T-4  T-3  T-2  T-1 T
[1]   1   1 <NA> <NA> <NA> <NA> A
[1]1  1   2 <NA> <NA> <NA>    A A
[1]2  1   3 <NA> <NA>    A    A C
[1]3  1   4 <NA>    A    A    C D
[1]4  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.

Chapter 4

Importing and handling sequence data in TraMineR

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 Section 4.2, or an event sequence object as explained in Section 7.1).

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. We describe then how TraMineR can create a sequence object by converting data from several different input formats. 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. We also describe functions provided by the TraMineR package to work with sequence data and to convert from one format into another one.

4.1 Importing data sets into R

Data files generated by statistical programs such as SPSS and Stata can be directly imported into R by using the `foreign`[1] library and saved later as R data sets. For SPSS files the import command reads `read.spss()` and it is `read.dta()` for Stata files. For more details, see the R-data manual http://cran.r-project.org/doc/manuals/R-data.pdf.

4.1.1 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], whose values will be 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 numerical coding is lost. If you prefer preserving the numerical coding and losing the labels, use the `convert.factors = FALSE` option.

[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.

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

Stata ('.dta') format. Here is an example on how to import the living arrangement history data from the biographic questionnaire of the Swiss Household Panel (SHP). The file *shp0_bvla_user.dta* containing the data is provided on the SHP CD that the users receive once they have signed a data protection contract. The R function to import data sets saved in the Stata ('.dta') format is provided by the *foreign* library and reads `read.dta()`. The `head()` function shows the first 6 rows of the imported data set.

```
> library(foreign)
> LA <- read.dta("*PATH*/SHP-Data/Biography/STATA/shp0_bvla_user.dta")
> head(LA)
  idpers q.source bvla.idx bvla013 bvla014                       bvla100
1   **01     2002        1    1965    1989     with both natural parents
2   **01     2002        2    1989    1990 with partner (married or not)
3   **01     2002        3    1990    1991 with partner (married or not)
4   **01     2002        4    1991    2002 with partner (married or not)
5   **02     2002        1    1968    1985     with both natural parents
6   **02     2002        2    1985    1988                        alone
```

SPSS ('.sav') format The *biofam* data set (see Section 2.2.2) included in the TraMineR package was created in SPSS. Here is how it was converted into an R data frame using the `read.spss()` function provided by the *foreign* library.

```
> library(foreign)
> biofam <- read.spss("*PATH*/biofam.sav", to.data.frame=TRUE)
```

4.1.2 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 was given in Section 2.2.3 p. 20 where we provide the command for reading the freely available *MVAD* data set.

4.1.3 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 4.1.1) are stored as data frames. The *actcal* and *biofam* data sets are each a data frame object.

4.1.4 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). The TraMineR package 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.

```
> 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. The sequences below are stored in the compressed format. They are sequences of the *actcal* data set compressed with the `seqconc()` function.

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

4.2 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[5] together with some of their attributes.

The `seqdef()` function accepts input data stored in several of the formats described in Chapter 3. Some examples showing how to create a sequence object from sequence data in several input formats are provided below.

[3] This is done by means of the `seqfcheck()` function that searches for any separator in the data.
[4] In TraMineR, the default separator is '-', but other user specified separators can be specified.
[5] The class of this object is 'stslist'.

4.2.1 Creating a 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.

```
> data(actcal)
> actcal.seq <- seqdef(actcal,13:24)
 [>] distinct states appearing in the data: A/B/C/D
 [>] alphabet: 1=A 2=B 3=C 4=D
 [>] 2000 sequences in the data set
 [>] min/max sequence length: 12/12
```

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.

```
> 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.

```
> actcal.seq <- seqdef(actcal, var=c("jan00","feb00","mar00","apr00",
+ "may00","jun00","jul00","aug00","sep00","oct00","nov00","dec00"))
 [>] distinct states appearing in the data: A/B/C/D
 [>] alphabet: 1=A 2=B 3=C 4=D
 [>] 2000 sequences in the data set
 [>] min/max sequence length: 12/12
```

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 3.2.1), 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).

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

```
> seq1 <- "000-000-000-000-000-000-000-000-000-000-000-000-0W0-0W0-0W0-
0W0-0W0-0W0-0W0-0W0-0W0-0W0-0WU-0WU-0WU-0WU-0WU-1WU-1WU"
> seq2 <- "000-000-000-000-000-000-000-000-000-000-000-000-0W0-0W0-0W0-
0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-1WU-1WU"
> 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

```
> seq.ex1 <- seqdef(seq.ex1)
 [>] distinct states appearing in the data: 000/0W0/0WU/1WU
 [>] alphabet: 1=000 2=0W0 3=0WU 4=1WU
 [>] 2 sequences in the data set
 [>] min/max sequence length: 28/28
> seq.ex1
    Sequence
[1] 000-000-000-000-000-000-000-000-000-000-000-000-0W0-0W0-0W0-0W0-0W0-
0W0-0W0-0W0-0W0-0WU-0WU-0WU-0WU-0WU-1WU-1WU
[2] 000-000-000-000-000-000-000-000-000-000-000-000-0W0-0W0-0W0-0W0-0W0-
0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-1WU-1WU
```

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

```
> print(seq.ex1,format="SPS")
 [>] STS sequences converted to 2 SPS seq./rows
    SPS sequence
[1] (000,12)-(0W0,9)-(0WU,5)-(1WU,2)
[2] (000,12)-(0W0,14)-(1WU,2)
```

As discussed in the previous chapter, sequences may not be stored in the STS format, depending on the way data were collected and the way information is organized. In some cases, sequences are not directly present but can be constructed 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 for further visualizing and analysis, using some 'real-life' example data sets. The `informat` option to the `seqdef()` function is used to specify the format of the input data. To identify the format of your data, see Section 3.2.

Creating a sequence object from SPS-formatted data In the SPS format (see Section 3.2.2), sequences are defined with state-duration couples. The next example shows the content of a text file with such data and some covariates

```
 1 95506 0.896 20 2 0 4 4 M/44 MC/9 SC/91
 2 95507 0.967 20 1 0 4 1 S/66 U/10 M/12 MC/56
 3 95508 0.967 20 1 0 4 4 S/72 U/5 M/67
 4 95510 0.896 20 2 0 4 1 S/10 U/1 UC/133
 5 95527 0.967 20 1 0 4 4 S/54 U/18 S/15 U/11 M/29 MC/17
 6 95530 0.896 20 2 0 4 2 S/10 U/14 M/8 MC/112
 7 95534 0.896 20 2 0 4 3 S/36 U/47 S/45 M/16
 8 95537 0.842 20 3 0 4 4 S/86 M/52 MC/6
 9 95538 0.967 20 1 0 4 4 S/134 M/10
10 95544 0.967 20 1 0 4 1 S/111 U/33
```

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 row names $(1, 2, 3, \ldots)$ are contained in the first column with the `row.names=1` option, and that empty strings should be treated as missing values with the `na.strings=""`.

```
> sweden <- read.table(file="**PATH**/sweden.txt",
+ header=FALSE, row.names=1, sep=" ", na.strings="")
```

The sequence data is contained in columns 8 to 19 (named V9 to V20 because in the text file the first column contained row names). Note that sequences are stored in an unequal number of variables, depending on the number of distinct states the individuals passed through.

```
> head(sweden)
     V2    V3 V4 V5 V6 V7 V8   V9  V10    V11    V12  V13   V14  V15  V16
```

```
1 95506 0.896 20  2  0  4  4 M/44 MC/9  SC/91   <NA> <NA>  <NA> <NA> <NA>
2 95507 0.967 20  1  0  4  1 S/66 U/10   M/12  MC/56 <NA>  <NA> <NA> <NA>
3 95508 0.967 20  1  0  4  4 S/72  U/5   M/67   <NA> <NA>  <NA> <NA> <NA>
4 95510 0.896 20  2  0  4  1 S/10  U/1 UC/133   <NA> <NA>  <NA> <NA> <NA>
5 95527 0.967 20  1  0  4  4 S/54 U/18   S/15   U/11 M/29 MC/17 <NA> <NA>
6 95530 0.896 20  2  0  4  2 S/10 U/14    M/8 MC/112 <NA>  <NA> <NA> <NA>
   V17  V18  V19  V20 V21
1 <NA> <NA> <NA> <NA>  NA
2 <NA> <NA> <NA> <NA>  NA
3 <NA> <NA> <NA> <NA>  NA
4 <NA> <NA> <NA> <NA>  NA
5 <NA> <NA> <NA> <NA>  NA
6 <NA> <NA> <NA> <NA>  NA
```

Now importing this data into a sequence object is very straightforward

```
> sweden.seq <- seqdef(data=sweden,var=8:19,informat='SPS2')
 [>] SPS2 data converted into 1989 STS sequences
 [>] distinct states appearing in the data: M/MC/S/SC/U/UC
 [>] alphabet: 1=M 2=MC 3=S 4=SC 5=U 6=UC
 [>] 1989 sequences in the data set
 [>] min/max sequence length: 144/144
```

where SPS2 is the SPS format with the '/' separator, i.e./ that the coding is of the form state/duration.

Creating a sequence object from SPELL-formatted data Data in the SPELL format can be converted and loaded in a sequence object with the `informat="SPELL"` option. The data provided as spell input should comply the structure described in Table 4.1 (variable order must be respected). An example is shown in 3.2.4 on page 27. The spells for each individual must be sorted by starting time. As an example let us create a sequence object with SPELL data extracted from

Table 4.1: Structure for the spell format

Position	Variable
1	Personal identification number
2	Spell index for the given individual
3	Start time
4	End time
5	Status

the Swiss Household Panel retrospective survey. The original data containing living arrangement history (see Table 3.2 on page 27 for the state description) has been imported into R. The living arrangement histories for three individuals (id = 2713, 2714 and 3713) are displayed below.

```
> LA[1:15,]
   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
```

Now we create a sequence object from this data

```
> LA.seq <- seqdef(LA, informat="SPELL")
 [>] SPELL data converted into 5560 STS sequences
 [>] STS sequences converted to 5560 STS seq./rows
 [>] distinct states appearing in the data: /1/10/11/2/3/4/5/6/7/8/9
 [>] alphabet: 1= 2=1 3=10 4=11 5=2 6=3 7=4 8=5 9=6 10=7 11=8 12=9
 [>] 5560 sequences in the data set
 [>] min/max sequence length: 0/122
```

and display (in SPS format) the resulting sequences for the first three individuals

```
> print(LA.seq[1:3,], format="SPS")
 [>] converting from STS to SPS formats => 3 STS => 3 SPS seq./rows
    SPS sequence
[1] (1,24)-(5,13)
[2] (1,17)-(7,3)-(5,13)
[3] (1,17)-(3,5)-(4,1)-(3,1)-(4,14)-(7,2)
```

4.2.2 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's functions, for instance the alphabet, color palette and state labels associated to the object are used by the TraMineR sequence plotting functions[6]. If no values for the attributes are provided by the user, TraMineR sets them automatically. The default values chosen by TraMineR and the user-available options to override them are described below.

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. Suppose we have imported a data set where the value labels for the possible states are "Happy", "Unhappy", "Rich" and "Dead". Looking at the data, we get

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

```
> seq.ex2
       T1    T2      T3      T4      T5      T6   T7   T8   T9  T10
[1] Happy Happy Unhappy Unhappy Unhappy Unhappy Rich Rich Rich Rich
[2] Happy Happy    Rich    Rich    Rich    Rich Dead Dead Dead Dead
```

Now, if we create a sequence object with this data set, the sequences show out in compressed form

```
> seqdef(seq.ex2)
 [>] Alphabet is defined as the distinct states appearing in the data
 [>] 2 sequences in the data set
 [>] min/max sequence length: 10/10
    Sequence
[1] Happy-Happy-Unhappy-Unhappy-Unhappy-Unhappy-Rich-Rich-Rich-Rich
[2] Happy-Happy-Rich-Rich-Rich-Rich-Dead-Dead-Dead-Dead
```

Depending on the length of the state labels and the length of the sequences, 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. Suppose we want to use the state labels "D" for "Dead", "H" for "Happy","R" for "Rich" and "U" for "Unhappy" when creating the sequence object from the previous example. We have to provide the `seqdef` function with these new labels. The order in which we give these labels is very important: The labels must *match the order of the states* as sorted in the list outputted by the `seqstatl` function (most often it is the alphabetical order).

```
> seqstatl(seq.ex2)
[1] "Dead"    "Happy"   "Rich"    "Unhappy"
```

Now, we create the sequence object with the new state labels, yielding shorter output when displaying the sequences

```
> seqdef(seq.ex2,states=c("D","H","R","U"))
 [>] 2 sequences in the data set
 [>] min/max sequence length: 10/10
    Sequence
[1] H-H-U-U-U-U-R-R-R-R
[2] H-H-R-R-R-R-D-D-D-D
```

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

```
> actcal.s1 <- seqdef(actcal[1:3,13:24])
 [>] distinct states appearing in the data: A/B/D
 [>] alphabet: 1=A 2=B 3=D
 [>] 3 sequences in the data set
 [>] min/max sequence length: 12/12
```

and one with sequences 7 to 9.

```
> actcal.s2 <- seqdef(actcal[7:9,13:24])
 [>] distinct states appearing in the data: A/D
 [>] alphabet: 1=A 2=D
 [>] 3 sequences in the data set
 [>] min/max sequence length: 12/12
```

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 ...

```
> actcal.s1 <- seqdef(actcal[1:3,13:24], alphabet=c("A","B","C","D"))
 [>] distinct states appearing in the data: A/B/D
 [>] alphabet: 1=A 2=B 3=C 4=D
 [>] 3 sequences in the data set
 [>] min/max sequence length: 12/12
```

... and the second object

```
> actcal.s2 <- seqdef(actcal[7:9,13:24], alphabet=c("A","B","C","D"))
 [>] distinct states appearing in the data: A/D
 [>] alphabet: 1=A 2=B 3=C 4=D
 [>] 3 sequences in the data set
 [>] min/max sequence length: 12/12
```

which permits to directly compare plots and tabulations of each sequence object.

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.

```
> actcal.seq <- seqdef(actcal,13:24, cpal=c("red","blue","green","yellow"))
 [>] distinct states appearing in the data: A/B/C/D
 [>] alphabet: 1=A 2=B 3=C 4=D
 [>] 2000 sequences in the data set
 [>] min/max sequence length: 12/12
```

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

```
> 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.

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

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.

```
> actcal.seq <- seqdef(actcal,13:24,labels=c("> 37 hours", "19-36 hours",
+ "1-18 hours", "no work"))
 [>] distinct states appearing in the data: A/B/C/D
 [>] alphabet: 1=A 2=B 3=C 4=D
 [>] 2000 sequences in the data set
 [>] min/max sequence length: 12/12
```

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 4.2.3).

4.2.3 Indexing and printing sequences

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

```
> actcal.seq[1:5,3:8]
    Sequence
[1] B-B-B-B-B-B
[2] D-D-A-A-A-A
[3] B-B-B-B-B-B
[4] C-C-C-C-C-C
[5] 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

```
> print(actcal.seq[1:5,3:8], ext=TRUE)
    mar00 apr00 may00 jun00 jul00 aug00
[1]     B     B     B     B     B     B
[2]     D     D     A     A     A     A
[3]     B     B     B     B     B     B
[4]     C     C     C     C     C     C
[5]     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

```
> print(actcal.seq[1:5,3:8], format="SPS")
 [>] STS sequences converted to 5 SPS seq./rows
    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, a sequence object is first created from the now well known *actcal* data set, with user specified color palette and states labels

```
> actcal.seq <- seqdef(actcal,13:24,
+ labels=c("> 37 hours", "19-36 hours", "1-18 hours", "no work"),
+ cpal=c("red","blue","green","yellow"))
 [>] distinct states appearing in the data: A/B/C/D
 [>] alphabet: 1=A 2=B 3=C 4=D
 [>] 2000 sequences in the data set
 [>] min/max sequence length: 12/12
```

and then the sequences for the summer months only are selected. 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.

```
> actcal.summer <- actcal.seq[,6:9]
> attr(actcal.summer,"cpal")
[1] "red"    "blue"   "green"  "yellow"
> attr(actcal.summer,"labels")
[1] "> 37 hours"  "19-36 hours" "1-18 hours"  "no work"
> attr(actcal.summer,"start")
[1] 6
```

The column names are retrieved with the `names` function

```
> names(actcal.summer)
[1] "jun00" "jul00" "aug00" "sep00"
```

4.2.4 Sequences of unequal length and missing values

For several reasons, sequences in a data set may have different lengths. For example:

- The length of the follow up is not the same for all individuals.
- In event sequences, the number of events experienced by each individual differs from one individual to the other.
- Sequences defined as the list of successive states without duration information are typically of varying length.

In the example below taken from Elzinga (2008), sequences have unequal lengths because they contain only the distinct states the individuals have passed through. The example is based on the *famform* data set distributed with the TraMineR package. In this data set, sequences are recorded in the compressed format, i.e. as character strings.

```
> data(famform)
> famform
  Sequence
v "S-U"
w "S-U-M"
x "S-U-M-MC"
y "S-U-M-MC-SC"
z "U-M-MC"
```

Suppose we loaded this example from a data file where the sequences are in the 'extended' format, i.e. each state is stored in a separate column (variable). It would look like this, with NA values at the end of the sequences containing less than the maximum number of states encountered

```
> 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
```

In TraMineR, the NA at the end of a sequence are not considered as true missing values. The sequence is supposed to end with the first NA value it contains. For example, when creating a sequence object from the extended version of the *famform* data set, we get a sequence of length 2 for the first record.

```
> seqdef(seqdecomp(famform))
 [>] Alphabet is defined as the distinct states appearing in the data
 [>] 5 sequences in the data set
 [>] min/max sequence length: 2/5
    Sequence
[1] S-U
[2] S-U-M
[3] S-U-M-MC
[4] S-U-M-MC-SC
[5] U-M-MC
```

4.3 Converting between formats

TraMineR also offers facilities to convert to and from several data forms. Such transformations may prove useful for applying other statistical methods to our data such as for instance survival analysis and regression trees). Data conversion is done with the `seqformat()`, `seqconc()` and `seqdecomp()` functions described in this section. The first of these functions is also used to create event sequences from state sequences, which can then be analysed with the TraMineR functions dedicated to event sequences (see Chapter7). 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 compressed format, in which the sequences are stored in 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.

4.3.1 Converting to and from the SPS format

The `seqformat()` function allows to convert from and to the state-permanence-sequence SPS format (see Section 4.3). 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.

```
> actcal.SPS <- seqformat(actcal, 13:24, from='STS', to='SPS')
 Converting sequence data from STS  to SPS
> actcal.SPS[1:10]
 [1] "(B,12)"            "(D,4)-(A,7)-(D,1)" "(B,12)"
 [4] "(C,9)-(B,3)"       "(A,12)"            "(D,1)-(B,11)"
 [7] "(D,12)"            "(A,12)"            "(A,12)"
[10] "(A,12)"
```

4.3.2 Converting between compressed and extended formats

The `seqconc()` and `seqdecomp()` functions convert between compressed and extended representations of sequence data. In the example below, we use the `seqconc()` function to translate the *actcal* data set into the compressed format and

store the result in the `actcal.comp` object. The `actcal.comp` object is now a vector, whose first 10 elements are shown here.

```
> actcal.comp <- seqconc(actcal,13:24)
> actcal.comp[1:10]
 [1] "B-B-B-B-B-B-B-B-B-B-B-B" "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" "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" "D-B-B-B-B-B-B-B-B-B-B-B"
 [7] "D-D-D-D-D-D-D-D-D-D-D-D" "A-A-A-A-A-A-A-A-A-A-A-A"
 [9] "A-A-A-A-A-A-A-A-A-A-A-A" "A-A-A-A-A-A-A-A-A-A-A-A"
```

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, the sequence is stored in the sole variable[7]

```
> actcal.ext <- seqdecomp(actcal.comp)
> actcal.ext[1:10,]
     [,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"
[7]  "D"  "D"  "D"  "D"  "D"  "D"  "D"  "D"  "D"  "D"   "D"   "D"
[8]  "A"  "A"  "A"  "A"  "A"  "A"  "A"  "A"  "A"  "A"   "A"   "A"
[9]  "A"  "A"  "A"  "A"  "A"  "A"  "A"  "A"  "A"  "A"   "A"   "A"
[10] "A"  "A"  "A"  "A"  "A"  "A"  "A"  "A"  "A"  "A"   "A"   "A"
```

In the next example taken from Aassve et al., 2007 and introduced in Section 3.1.1, states are coded with character strings of length 3 and separated with the '-' symbol. Each sequence is transformed into a (row) vector of 28 elements (states).

```
> seqdecomp(seq.ex1)
    [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9]  [,10] [,11] [,12]
[1] "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000"
[2] "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000" "000"
    [,13] [,14] [,15] [,16] [,17] [,18] [,19] [,20] [,21] [,22] [,23] [,24]
[1] "0W0" "0W0" "0W0" "0W0" "0W0" "0W0" "0W0" "0W0" "0W0" "0WU" "0WU" "0WU"
[2] "0W0" "0W0" "0W0" "0W0" "0W0" "0W0" "0W0" "0W0" "0W0" "0W0" "0W0" "0W0"
    [,25] [,26] [,27] [,28]
[1] "0WU" "0WU" "1WU" "1WU"
[2] "0W0" "0W0" "1WU" "1WU"
```

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.

```
> seqdecomp("aaaaaa", sep="")
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,] "a"  "a"  "a"  "a"  "a"  "a"
```

4.3.3 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 $ns \times ns$ where ns is the number

[7] 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.

of distinct states appearing in the sequences must be given. In this matrix, the cell (a, b), where a is the row index and b the column index, contains all events associated with a transition from state a to state b separated by a comma. The diagonal of this matrix has a special meaning. It just defines the initial event of the sequence. Hence the position (a, a) gives the event generated when the sequence starts with the 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 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 correspondence) 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 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.

```
> data(actcal)
> ## Creating STS format
> actcal.seq <- seqdef(actcal,13:24,
+ labels=c("FullTime", "PartTime", "LowPartTime", "NoWork"))
> transition <- seqetm(actcal.seq, method="transition")
> 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 method generates a "begin" event and an "end" event for each spell. The diagonal is considered as a "begin" event. The extra parameter "bp" can be set to "begin" to generate a distinct event on the diagonal.

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

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".

Table 4.2: 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 4.3: Events associated to each state transition

From state	To state: Full time A	Part time B	Low part time C	No work 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

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.

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

```
> actcal.tse <- seqformat(actcal, var=13:24, from='STS', to='TSE',
```

```
+ tevent=transition)
 => Converting 2000 sequences to TSE format, please wait
 [>] STS sequences converted to 2954 TSE seq./rows
> actcal.tse[1:10,]
   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
7   4    0 LowPartTime
8   4    9    Increase
9   4    9    PartTime
10  5    0    FullTime
```

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.

4.3.4 Converting from SPELL format

The following command translates the `LA` data set described above (page 34) to the STS sequence-status format. The `from` option of the `seqformat()` function is set to 'SPELL'.

```
> LA.sts <- seqformat(LA, from="SPELL", to="STS")
 [>] SPELL data converted into 5560 STS sequences
Il y a eu 50 avis ou plus (utilisez warnings() pour voir les 50 premiers)
> LA.sts[1:4,]
[1] "1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-5-5-5-5-5-5-5-5-5-
5-5-5-5"
[2] "1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-7-7-7-5-5-5-5-5-5-5-5-5-5-5-5-5"
[3] "1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-3-3-3-3-3-4-3-4-4-4-4-4-4-4-4-
4-4-4-4-4-7-7"
[4] "1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-1-5-5-5-5-5-5-5-5-5"
```

It is also possible to convert directly into the more concise 'state-permanence' format by setting the `to` option to 'SPS'

```
> seqformat(LA[1:17,],from="SPELL",to="SPS")
 [>] SPELL data converted into 4 STS sequences
 [>] STS sequences converted to 4 SPS seq./rows
    SPS sequence
[1] "(1,24)-(5,13)"
[2] "(1,17)-(7,3)-(5,13)"
[3] "(1,17)-(3,5)-(4,1)-(3,1)-(4,14)-(7,2)"
[4] "(1,20)-(5,9)"
```

Chapter 5

Describing and visualizing sequences

This chapter presents the main TraMineR tools for describing and visualizing sequences. We first briefly explain in Section 5.1 the general plotting philosophy adopted in TraMineR. Section 5.2 presents then tools for describing and visualizing set properties of the sequences from an aggregated standpoint and Section 5.3 focuses on the characterization of individual sequence properties and their summary.

5.1 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.

5.1.1 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 4.2.2) 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.

5.1.2 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 two plots and a legend side by side as shown in Figure 5.1.

```
par(1,3)
seqdplot(biofam.seq, withlegend=FALSE)
seqfplot(biofam.seq, withlegend=FALSE, pbarw=TRUE)
seqlegend(biofam.seq)
```

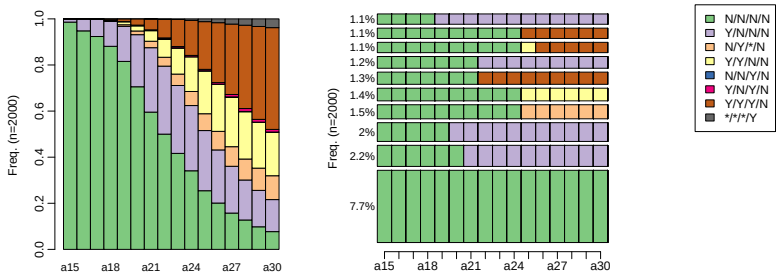


Figure 5.1: Legend plotted as an additional graphic

5.2 Describing and visualizing sequence data

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

5.2.1 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 2.1 page 18 for their description).

```
> data(actcal)
> stated <- c("> 37 hours", "19-36 hours", "1-18 hours", "no work")
> actcal.seq <- seqdef(actcal,13:24,labels=stated)
 [>] distinct states appearing in the data: A/B/C/D
 [>] alphabet: 1=A 2=B 3=C 4=D
 [>] 2000 sequences in the data set
 [>] min/max sequence length: 12/12
> 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.

[1] in that case, some states in the alphabet may not appear in the data. See Sec. 4.2 for more information on this topic

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

```
> sp.ex1 <- rbind("(000,12)-(0W0,9)-(0WU,5)-(1WU,2)",
+ "(000,12)-(0W0,14)-(1WU,2)")
> sp.ex1
     [,1]
[1,] "(000,12)-(0W0,9)-(0WU,5)-(1WU,2)"
[2,] "(000,12)-(0W0,14)-(1WU,2)"
> seqstatl(sp.ex1, format='SPS')
 [>] SPS data converted into 2 STS sequences
[1] "000" "0W0" "0WU" "1WU"
```

5.2.2 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 *biofam* data set. We first define a `bfstates` vector of state labels to be used for the legend of the colors used in the plot. This vector has eight elements since there are eight different states in the *biofam* data set. The states are defined according to Table 2.3 p. 19 and their labels are abbreviated 'Y' for Yes, 'N' for No and '*' for neither Yes nor No. The position of the Y, N, * symbols refers respectively to leaved parental home, married, children and divorce.

```
> bfstates <- c("N/N/N/N", "Y/N/N/N", "N/Y/*/N", "Y/Y/N/N",
+ "N/N/Y/N", "Y/N/Y/N", "Y/Y/Y/N", "*/*/*/Y")
> biofam.seq <- seqdef(biofam,10:25,labels=bfstates)
 [>] distinct states appearing in the data: 0/1/2/3/4/5/6/7
 [>] alphabet: 1=0 2=1 3=2 4=3 5=4 6=5 7=6 8=7
 [>] 2000 sequences in the data set
 [>] min/max sequence length: 16/16
> seqdplot(biofam.seq)
```

The `ltext` argument overrides the labels associated to the sequence object (See Section 4.2.2). The resulting graphic is shown in Figure 5.2. The proportion of individuals who have not leaved parental home diminishes over time, while the proportion of individuals in state Y/Y/Y/N (having left home, getting married and having a child) increases to become the most frequent state at age 30.

The state distribution plot for the *actcal* data, obtained with the command below, shows a different pattern (Figure 5.3). The distribution of the work statuses looks very stable over time.

```
> 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 2.3 page 19 for the description of the states). The first argument to the `seqstatd()` function is the previously created `biofam.seq` sequence object.

```
> seqstatd(biofam.seq)
$Frequencies
   a15  a16  a17  a18  a19  a20  a21  a22  a23  a24  a25  a26  a27  a28  a29  a30
0 0.99 0.95 0.92 0.88 0.82 0.71 0.60 0.50 0.42 0.34 0.25 0.20 0.16 0.13 0.10 0.08
1 0.01 0.05 0.08 0.11 0.15 0.23 0.28 0.30 0.29 0.28 0.26 0.23 0.20 0.17 0.16 0.14
2 0.00 0.00 0.00 0.00 0.01 0.02 0.03 0.04 0.05 0.06 0.07 0.08 0.08 0.09 0.10 0.10
3 0.00 0.00 0.00 0.00 0.01 0.02 0.05 0.08 0.11 0.15 0.19 0.20 0.22 0.20 0.20 0.19
4 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00
5 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.01 0.01 0.01 0.01 0.01 0.01 0.01 0.01
6 0.00 0.00 0.00 0.00 0.01 0.03 0.05 0.08 0.12 0.15 0.21 0.26 0.31 0.36 0.40 0.44
```

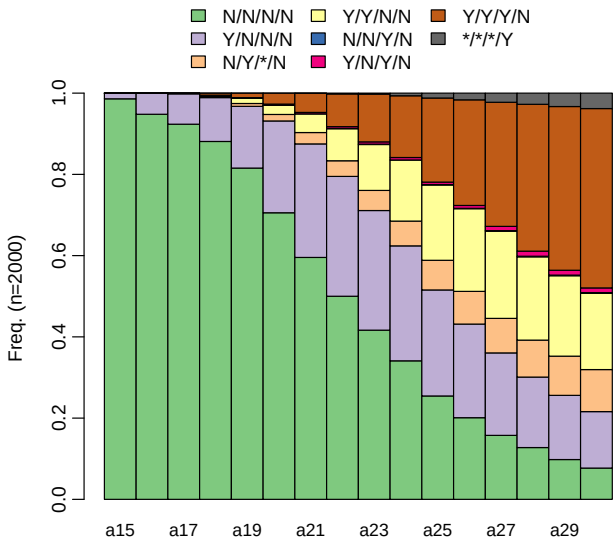


Figure 5.2: Distribution of the family statuses by age in the *biofam* data set (data from the Swiss Household Panel)

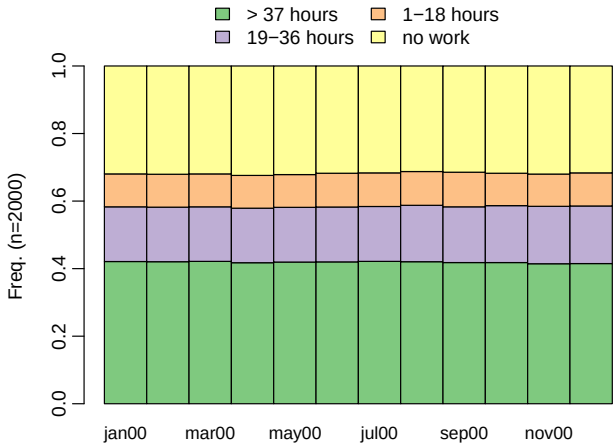


Figure 5.3: Distribution of the work statuses by month in the *actcal* data set (data from the Swiss Household Panel)

```
7 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.00 0.01 0.01 0.02 0.02 0.03 0.03 0.04

$ValidStates
 a15  a16  a17  a18  a19  a20  a21  a22  a23  a24  a25  a26  a27  a28  a29  a30
2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000 2000

$Entropy
 a15  a16  a17  a18  a19  a20  a21  a22  a23  a24  a25  a26  a27  a28  a29  a30
0.04 0.10 0.13 0.20 0.29 0.41 0.52 0.61 0.68 0.74 0.78 0.79 0.80 0.79 0.77 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

$$h(p_1, \ldots, p_s) = -\sum_{i=1}^{s} p_i \log_2(p_i)$$

where s is the size of the alphabet. 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.

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 (0.06). The entropy of the state distribution rises with age and reaches its maximum at age 27. At this age, 14% percent of the respondents had not leaved parental home, 29% had leaved parental home but were not married and had no children (state 1), 1% had one or more children without being married, and 28% had one or more children and were married (state 6).

We can also plot the reported entropy measures. For that we need to access the `Entropy` element of the list returned by `seqstatd()`. We first store the outcome of the function in an object named `sd` and extract the entropy measures with `sd$Entropy`. 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 5.4. 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.

```
sd <- seqstatd(biofam.seq)
```

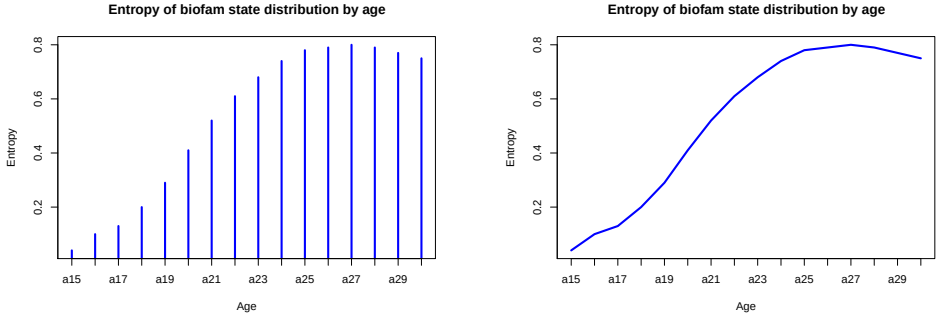


Figure 5.4: Entropy of state distribution by age - *biofam* data set

```
agelab <- paste("a",seq(15,30),sep="")
pdf(file="**PATH**/fg_biofam-entropy.pdf",
    width=8, height=6, pointsize=14)
plot(sd$Entropy,
     main="Entropy of biofam state distribution by age",
     xlab="Age", ylab="Entropy", type="h", lwd=3.5, col="blue",
     axes=F, cex=1.3, frame.plot=T)
axis(1,labels=agelab, at=1:16)
axis(2, at=c(0.0,.2,.4,.6,.8))
dev.off()
```

If you prefer a line you just change the plot type from 'h' to 'l'.

```
plot(sd$Entropy,
     main="Entropy of biofam state distribution by age",
     xlab="Age", ylab="Entropy", type="l", lwd=3.5, col="blue",
     axes=F, cex=1.3, frame.plot=T)
```

In the previous commands, many graphical options are provided to make the plot look nicer, e.g. customize the axis labels and change the line color and width. For a list of available graphical options, type `?plot` and `?par`. However, if you are frightened by all those options you can obtain your first, less sophisticated plot just by typing

```
plot(sd$Entropy, type="b")
```

5.2.3 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. 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 10 most frequent sequences of the `actcal.seq` sequence object. The resulting plot is shown in Figure 5.5. The labels appearing in the plot's legend are those attached to the object page 37.

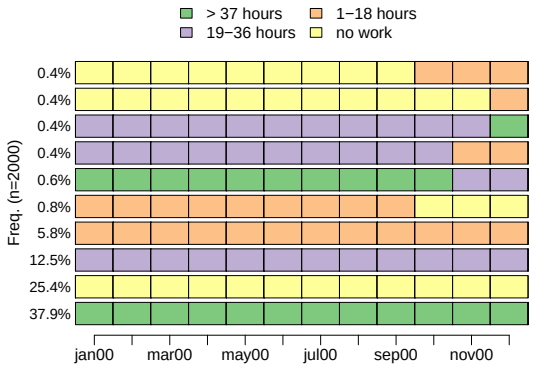


Figure 5.5: Plot of the 10 most frequent sequences in the *actcal* data set

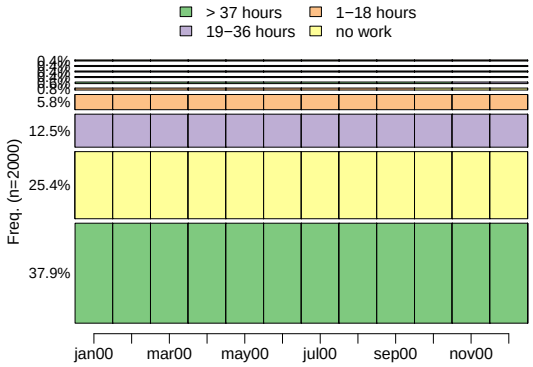


Figure 5.6: Plot of the 10 most frequent sequences in the *actcal* data set, with proportional bar widths

```
seqfplot(actcal.seq)
```

With the `pbarw=TRUE` option the bar widths are set proportional to the sequence frequency as shown in Figure 5.6.

```
seqfplot(actcal.seq, pbarw=TRUE)
```

The frequency plot for the `biofam.seq` sequence object (created from the *biofam* data set) is obtained with the following commands and shown in Figure 5.7. Here we attach the previously defined `bfstates` labels to the sequence object.

```
> data(biofam)
> biofam.seq <- seqdef(biofam,10:25, labels=bfstates)
 [>] distinct states appearing in the data: 0/1/2/3/4/5/6/7
 [>] alphabet: 1=0 2=1 3=2 4=3 5=4 6=5 7=6 8=7
 [>] 2000 sequences in the data set
 [>] min/max sequence length: 16/16
seqfplot(biofam.seq, pbarw=TRUE)
```

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 of the distinct sequences in the data set. Since the number of distinct sequences can be very high, one can limit the table to the most frequent sequences with the `tlim` option. 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 2.1 on page 18). 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.

```
> seqtab(actcal.seq, tlim=10)
         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

```
> seqtab(actcal.seq[,7:9], tlim=10)
        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
```

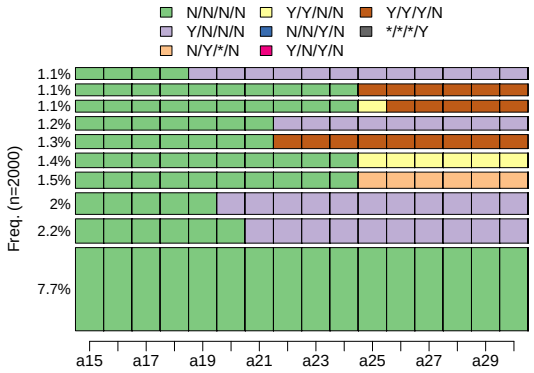


Figure 5.7: Plot of the 10 most frequent sequences in the *biofam* data set, with proportional bar widths

5.2.4 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.

```
> seqtrate(actcal.seq)
 [>] computing transition rates for states A/B/C/D ...
            [-> A]      [-> B]      [-> C]      [-> D]
[A ->] 0.986991870 0.005203252 0.001084011 0.006720867
[B ->] 0.009700665 0.970343681 0.007760532 0.012195122
[C ->] 0.005555556 0.014814815 0.934259259 0.045370370
[D ->] 0.008705580 0.006279435 0.014985015 0.970029970
```

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. First, we store the result of the `seqtrate()` function in an object named `tr`

```
> tr <- seqtrate(actcal.seq)
 [>] computing transition rates for states A/B/C/D ...
> tr
            [-> A]      [-> B]      [-> C]      [-> D]
[A ->] 0.986991870 0.005203252 0.001084011 0.006720867
[B ->] 0.009700665 0.970343681 0.007760532 0.012195122
[C ->] 0.005555556 0.014814815 0.934259259 0.045370370
[D ->] 0.008705580 0.006279435 0.014985015 0.970029970
```

and apply the `rowSums()` function, which returns the sum of the rows, to this object

```
> rowSums(tr)
[A ->] [B ->] [C ->] [D ->]
     1      1      1      1
```

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

```
> rowSums(seqtrate(actcal.seq))
 [>] computing transition rates for states A/B/C/D ...
[A ->] [B ->] [C ->] [D ->]
     1      1      1      1
```

5.3 Describing and visualizing individual sequences

5.3.1 Visualizing individual sequences

The `seqiplot()` function renders individual sequences with stacked bars depicting the statuses over time in the same manner as the `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 5.8). The legend uses the labels attached to the `actcal.seq` object and the color palette is the one set by default.

```
seqiplot(actcal.seq)
```

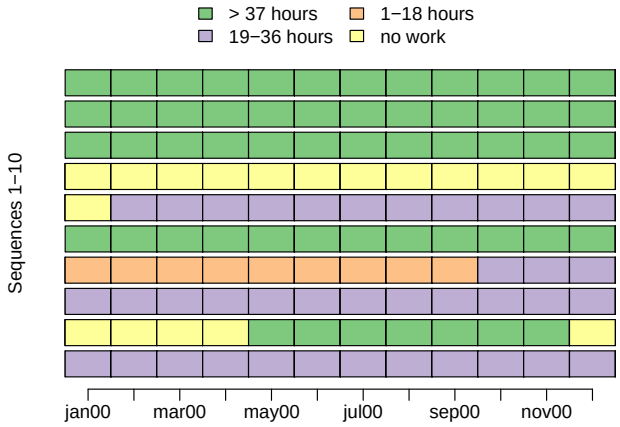


Figure 5.8: Plot of the 10 first sequences of the *actcal* data set (`seqiplot()`)

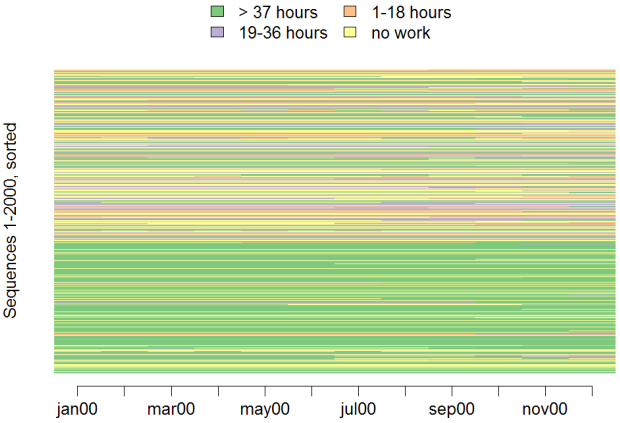


Figure 5.9: Plot of all sequences of the *actcal* data set (`seqiplot()`)

In the next example, we plot all sequences in the previously defined `actcal.seq` sequence object, sorted by sex with the `vsort` option. The `withborder=FALSE` 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 `vsort` option can be useful to distinguish patterns depending on a covariate value. Here the bottom sequences correspond to men (lower value) and the upper ones to women. We may observe that the color corresponding to partial time work is more frequent on the top half, i.e. for the female population (Figure 5.9). This plot of individual sequences complements the "averaged" representation provided by the state distribution plot by rendering the diversity of the sequences.

```
seqiplot(actcal.seq, sortv=actcal$sex, tlim=0,
         withborder=FALSE, space=0)
```

Remark: Outputting figures generated by `seqiplot()` for thousands of sequences produces very heavy files in postscript or pdf. We suggest that you save in that case the figures in png format by using `png()` instead of `postscript()` or `pdf()`. Figure 5.9, for instance, was obtained by issuing

```
png(file=paste(graphdir,"actcal-seqiplot-all.png",sep=""), unit="px",
    width=1024, height=850, pointsize=26)
```

before the `sequiplot()` command, and `dev.off()` afterwards. You may get an ides of the resulting quality degradation by comparing Figure 5.9 with for instance Figure 5.8. This permitted, however, to reduce the size of this manual in pdf format by about 5MB.

5.3.2 State frequencies by sequence

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

```
> seqistatd(actcal.seq[1:10,])
 [>] Computing state distribution for 10 sequences...
      A  B C  D
 [1,]  0 12 0  0
 [2,]  7  0 0  5
 [3,]  0 12 0  0
 [4,]  0  3 9  0
 [5,] 12  0 0  0
 [6,]  0 11 0  1
 [7,]  0  0 0 12
 [8,] 12  0 0  0
 [9,] 12  0 0  0
[10,] 12  0 0  0
```

We may be interested in the mean time spent in each state. This can be done 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 of by columns (2nd dimension) with the `apply` function.

```
> statd <- seqistatd(actcal.seq)
 [>] Computing state distribution for 2000 sequences...
> apply(statd,2,mean)
     A      B      C      D
5.0275 1.9745 1.1780 3.8200
```

We can indeed plot these mean values (Figure 5.10)

```
barplot(apply(statd,2,mean), main="Mean time spent in each state",
        col="lightblue")
```

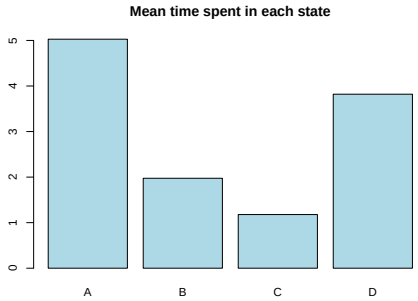


Figure 5.10: Mean time spent in each state, *actcal* data.

5.3.3 Extracting distinct states and durations

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

```
> print(actcal.seq[1:10,],"SPS")
 [>] STS sequences converted to 10 SPS seq./rows
     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)
[7]  (D,12)
[8]  (A,12)
[9]  (A,12)
[10] (A,12)
```

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 5.3.7 on page 64. In the following example we extract this separated information from the 10 first sequences of the `actcal.seq` object. Distinct sequences are obtained with

```
> seqdss(actcal.seq[1:10,])
 [>] alphabet: 1=A 2=B 3=C 4=D
 [>] 10 sequences in the data set
 [>] min/max sequence length: 1/3
     Sequence
[1]  B
[2]  D-A-D
[3]  B
[4]  C-B
[5]  A
[6]  D-B
[7]  D
```

```
[8]  A
[9]  A
[10] A
```

and durations with

```
> seqdur(actcal.seq[1:10,])
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
 [1,]   12   NA   NA   NA   NA   NA   NA   NA   NA    NA    NA    NA
 [2,]    4    7    1   NA   NA   NA   NA   NA   NA    NA    NA    NA
 [3,]   12   NA   NA   NA   NA   NA   NA   NA   NA    NA    NA    NA
 [4,]    9    3   NA   NA   NA   NA   NA   NA   NA    NA    NA    NA
 [5,]   12   NA   NA   NA   NA   NA   NA   NA   NA    NA    NA    NA
 [6,]    1   11   NA   NA   NA   NA   NA   NA   NA    NA    NA    NA
 [7,]   12   NA   NA   NA   NA   NA   NA   NA   NA    NA    NA    NA
 [8,]   12   NA   NA   NA   NA   NA   NA   NA   NA    NA    NA    NA
 [9,]   12   NA   NA   NA   NA   NA   NA   NA   NA    NA    NA    NA
[10,]   12   NA   NA   NA   NA   NA   NA   NA   NA    NA    NA    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.

5.3.4 Sequence length

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

```
> data(famform)
> famform.seq <- seqdef(famform)
 [>] distinct states appearing in the data: M/MC/S/SC/U
 [>] alphabet: 1=M 2=MC 3=S 4=SC 5=U
 [>] 5 sequences in the data set
 [>] min/max sequence length: 2/5
> 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
> seqlength(famform.seq)
[1] [2] [3] [4] [5]
  2   3   4   5   3
```

5.3.5 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 2.1 page 18 for the meaning of the states) into the activity calendar sequence data object.

```
> seqpm(actcal.seq,"DAAD")
 [>] pattern DAAD has been found in 4 sequences
$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).

```
> daad <- seqpm(actcal.seq,"DAAD")
 [>] pattern DAAD has been found in 4 sequences
> actcal.seq[daad$MIndex,]
    Sequence
[1] D-A-A-D-D-D-D-D-D-A-A-A
[2] D-D-A-A-D-D-A-A-A-A-A-A
[3] D-D-B-B-C-D-D-A-A-D-C-C
[4] D-D-D-D-A-A-D-A-B-B-D-D
```

5.3.6 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 5.3.7, 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

$$h(\pi_1, \ldots, \pi_s) = -\sum_{i=1}^{s} \pi_i \log_2 \pi_i$$

where s is the size of the alphabet and π_i the proportion of occurrences of the ith state in the considered sequence. The 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. 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

```
> seqient(actcal.seq[1:10,])
 [>] computing within sequence entropy for 10 sequences...
      Entropy
[1]  0.0000000
```

```
[2]  0.4899344
[3]  0.0000000
[4]  0.4056391
[5]  0.0000000
[6]  0.2069084
[7]  0.0000000
[8]  0.0000000
[9]  0.0000000
[10] 0.0000000
```

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 one vector for each sequence, and aggregate them with the `rbind()` function, obtaining a matrix that we then convert into a sequence object.

```
> s1 <- c("A","A","A","B","B","B","C","C","C","D","D","D")
> s2 <- c("A","D","A","B","C","B","C","B","C","D","A","D")
> s3 <- c("A","B","A","B","A","B","C","D","C","D","C","D")
> ex1 <- rbind(s1,s2,s3)
> 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"
> ex1 <- seqdef(ex1)
 [>] distinct states appearing in the data: A/B/C/D
 [>] alphabet: 1=A 2=B 3=C 4=D
 [>] 3 sequences in the data set
 [>] min/max sequence length: 12/12
```

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

```
> ex1
    Sequence
[1] A-A-A-B-B-B-C-C-C-D-D-D
[2] A-D-A-B-C-B-C-B-C-D-A-D
[3] A-B-A-B-A-B-C-D-C-D-C-D
```

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

```
> seqistatd(ex1)
 [>] Computing state distribution for 3 sequences...
    A B C D
[1,] 3 3 3 3
[2,] 3 3 3 3
[3,] 3 3 3 3
```

Now we are able 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 5.3.7), takes account of the state ordering.

```
> seqient(ex1)
 [>] computing within sequence entropy for 3 sequences...
   Entropy
[1]       1
[2]       1
[3]       1
```

The entropy without normalization is computed using the `norm=FALSE` option

```
> seqient(ex1,norm=FALSE)
 [>] computing within sequence entropy for 3 sequences...
     Entropy
[1] 1.386294
[2] 1.386294
[3] 1.386294
```

Now we are very impatient to plot an histogram of the within entropy of the sequences in the *actcal* data set. We first store the results of the `seqient()` function in an object named `actcal.ient` and plot it using the `hist()` function. 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.

```
> actcal.ient <- seqient(actcal.seq)
 [>] computing within sequence entropy for 2000 sequences...
> summary(actcal.ient)
    Entropy
 Min.   :0.00000
 1st Qu.:0.00000
 Median :0.00000
 Mean   :0.07484
 3rd Qu.:0.00000
 Max.   :0.97957
> hist(actcal.ient,col="cyan",
+ main="Entropy for the sequences in the actcal data set",
+ xlab="Entropy")
```

The histogram can be seen in Figure 5.11. To obtain this figure, we could have spare time by typing only one single command

```
> hist(seqient(actcal.seq),col="cyan",
+ main="Entropy for the sequences in the actcal data set",
+ xlab="Entropy")
 [>] computing within sequence entropy for 2000 sequences...
```

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. 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, which is labeled 5587 in our data set.

```
> max(actcal.ient)
[1] 0.979574
> which(actcal.ient==max(actcal.ient))
[1] 1836
> actcal.seq[1836,]
    Sequence
[1] 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.

```
> actcal[actcal.ient==max(actcal.ient),]
     idhous00 age00       educat00 civsta00 nbadul00 nbkid00 aoldki00 ayouki00
5587   116151    37 apprenticeship  married        2       2       14       10
```

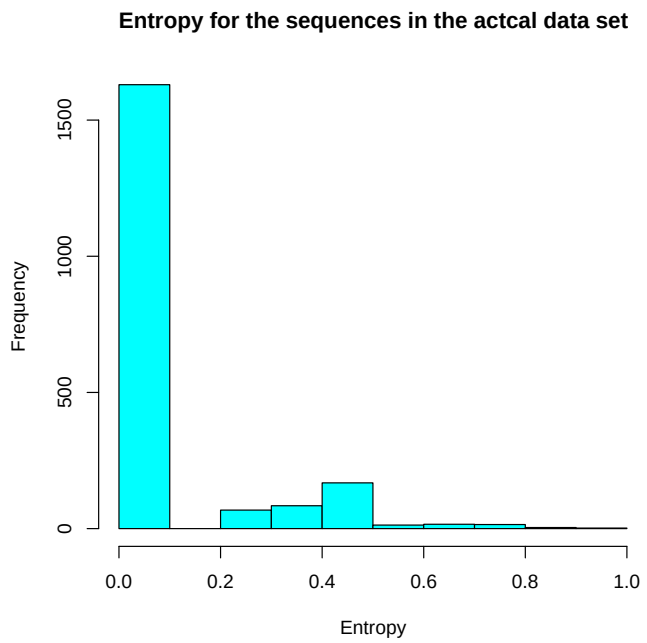


Figure 5.11: Within sequence entropies - Activity calendar

```
                    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 5.12 obtained with the following command

```
> biofam.ient <- seqient(biofam.seq)
 [>] computing within sequence entropy for 2000 sequences...
 [=] entropy computed for 2000 sequences
> hist(biofam.ient,col="cyan",
xlab="Entropy",
main="Entropy for the sequences in the 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.

```
> biofam <- data.frame(biofam, seqient(biofam.seq))
 [>] computing within sequence entropy for 2000 sequences...
```

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

```
> 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"
```

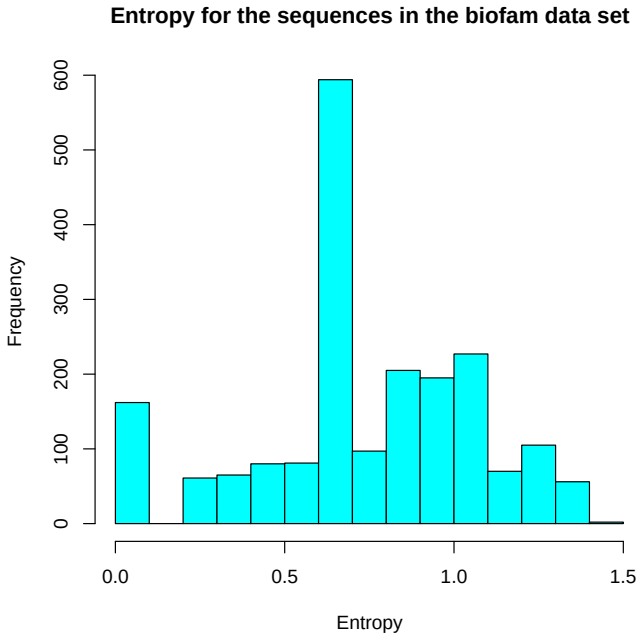


Figure 5.12: Within sequence entropies - *biofam* data set

```
[25] "a30"      "Entropy"
> 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 variables `q1`, `q49`, `q51`, `q99` for later usage.

```
> q1 <- quantile(biofam$Entropy,0.01)
> q49 <- quantile(biofam$Entropy,0.49)
> q51 <- quantile(biofam$Entropy,0.51)
> q99 <- quantile(biofam$Entropy,0.99)
```

Below, we extract three sets of sequences using the percentile values.

```
> ient.min <- biofam.seq[biofam$Entropy<=q1,]
> ient.med <- biofam.seq[biofam$Entropy>=q49 & biofam$Entropy<=q51,]
> ient.max <- biofam.seq[biofam$Entropy>=q99,]
```

Finally, we plot the three sets of sequences separately. We combine the three plots in a single graph, using the `par()` function. Recall that the `seqiplot()` function plots by default only the 10 first sequences, but this is enough. The result is shown in Figure 5.13. It confirms that the more there are different states in the sequence, the higher the entropy.

```
par(mfrow=c(2,2))
seqiplot(ient.min,
         title="10 seq. with low entropy",
         withlegend=FALSE)
```

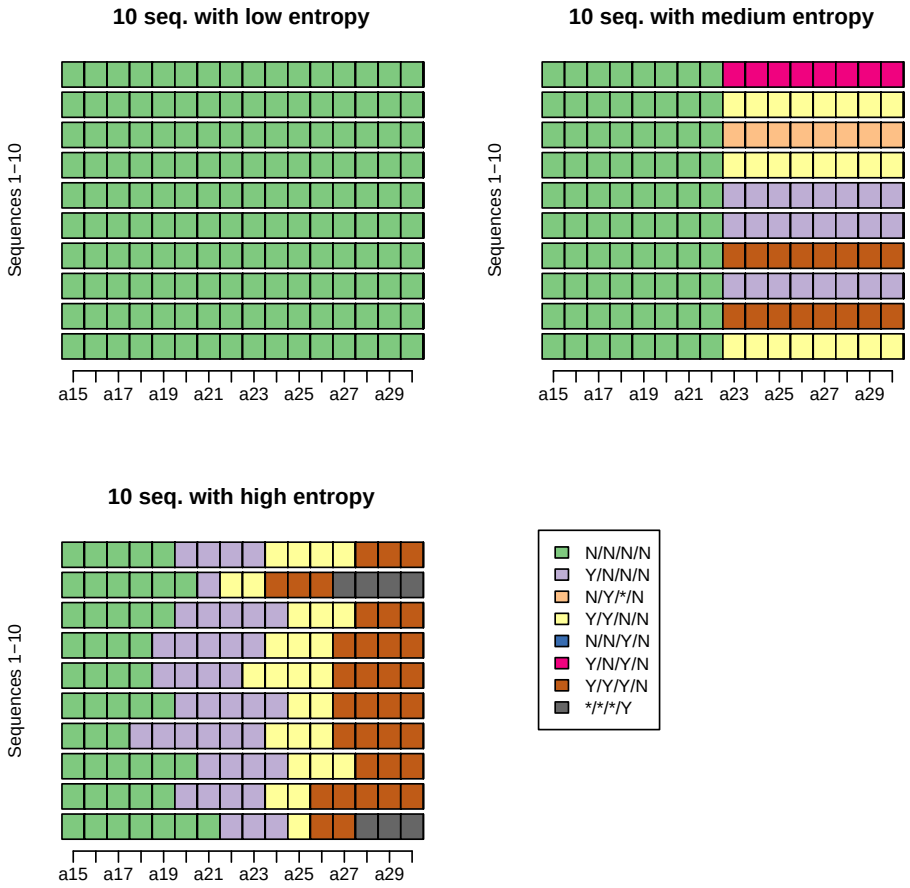


Figure 5.13: Low, median and high sequence entropies - *biofam* data set

```
seqiplot(ient.med,
         title="10 seq. with medium entropy",
         withlegend=FALSE)
seqiplot(ient.max,
         title="10 seq. with high entropy",
         withlegend=FALSE)
seqlegend(biofam.seq)
```

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 `tab()` 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.

```
> table(biofam$birthyr)

1909 1910 1911 1912 1913 1914 1915 1916 1917 1918 1919 1920 1921 1922 1923 1924
   1    1    2    2    3    2    4    5    6    9   16    5   11   16   20   25
1925 1926 1927 1928 1929 1930 1931 1932 1933 1934 1935 1936 1937 1938 1939 1940
```

```
  26   24   19   32   31   44   43   36   53   39   38   45   55   65   48   47
1941 1942 1943 1944 1945 1946 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956
  56   57   69   51   73   72   70   77   60   73   73   74   94   83   92   73
1957
  80
> biofam <- data.frame(biofam,
  ageg=cut(biofam$birthy,c(1909,1918,1928,1938,1948,1958),
  label=c("1909-18","1919-28","1929-38","1939-48","1949-58"),include.lowest=TRUE))
> table(biofam$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 5.14. The `Entropy ~ ageg` part of the command is a formula syntax widely used in R. Here it means 'plot the entropy by age group'.

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

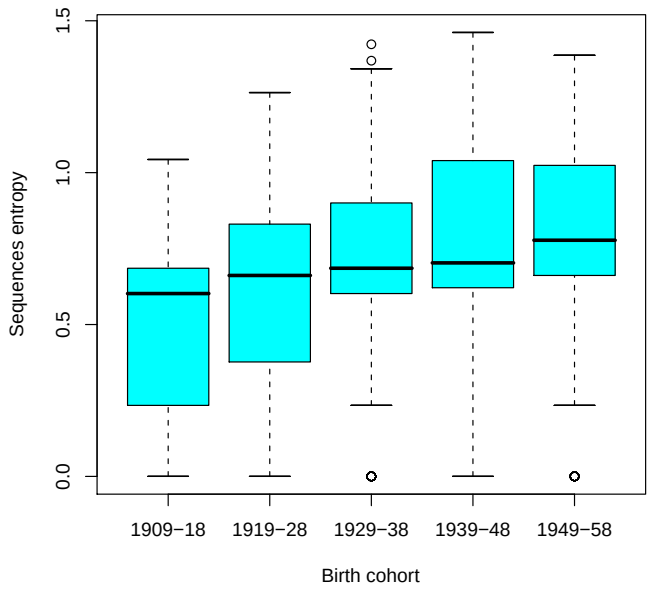


Figure 5.14: Boxplot of the within sequence entropies by birth cohort - *biofam* data set

The mean and median entropy are rising in the more recent birth cohorts. Figure (5.14) shows that the entropy is also sligthly higher in the women family formation history when compared to that of the men.

5.3.7 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

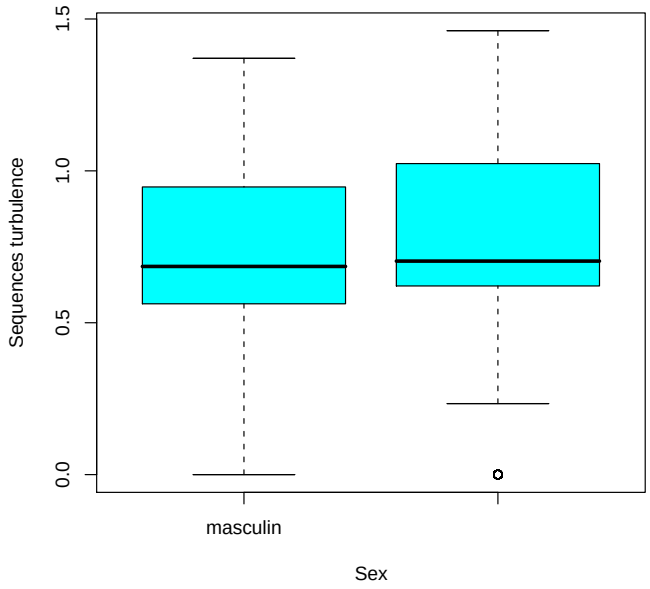


Figure 5.15: Boxplot of the within sequence entropies by sex - *biofam* data set

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

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

where s_t is the variance of the state-duration for the x sequence and $s_{t,max}$ is the maximum value that this variance can take given the total duration of the sequence. This maximum is computed as follow

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

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. As defined by Elzinga, the measure assumes a data set in the sequence-permanence SPS format. 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* dataset 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.

```
> data(actcal)
> actcal.seq <- seqdef(actcal,13:24)
 [>] distinct states appearing in the data: A/B/C/D
 [>] alphabet: 1=A 2=B 3=C 4=D
 [>] 2000 sequences in the data set
 [>] min/max sequence length: 12/12
> actcal.seq[2,]
    Sequence
```

```
[1] D-D-D-D-A-A-A-A-A-A-D
> seqdss(actcal.seq[2,])
 [>] 1 sequences in the data set
 [>] min/max sequence length: 3/3
   Sequence
[1] 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.

```
> seqsubsn(actcal.seq[2,],DSS=FALSE)
[1] 76
> seqsubsn(actcal.seq[2,],DSS=TRUE)
[1] 7
```

The `seqST()` function returns the sequence 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[3]. Hence we give the `informat='SPS'` option to the `seqdef()` function for creating the sequence object.

```
> sp.ex1
     [,1]
[1,] "(000,12)-(0W0,9)-(0WU,5)-(1WU,2)"
[2,] "(000,12)-(0W0,14)-(1WU,2)"
> sp.ex1 <- seqdef(sp.ex1,informat="SPS")
 [>] SPS data converted into 2 STS sequences
 [>] distinct states appearing in the data: 000/0W0/0WU/1WU
 [>] alphabet: 1=000 2=0W0 3=0WU 4=1WU
 [>] 2 sequences in the data set
 [>] min/max sequence length: 28/28
```

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

```
> sp.ex1
    Sequence
[1] 000-000-000-000-000-000-000-000-000-000-000-000-0W0-0W0-0W0-0W0-0W0-
0W0-0W0-0W0-0W0-0WU-0WU-0WU-0WU-0WU-1WU-1WU
[2] 000-000-000-000-000-000-000-000-000-000-000-000-0W0-0W0-0W0-0W0-0W0-
0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-0W0-1WU-1WU
```

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

```
> seqST(sp.ex1)
 [>] extracting symbols and durations
 [>] distinct states appearing in the data: 000/0W0/0WU/1WU
 [>] alphabet: 1=000 2=0W0 3=0WU 4=1WU
 [>] 2 sequences in the data set
 [>] min/max sequence length: 3/4
 [>] Computing turbulence for  2 sequences, please wait...
   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.

[3] see 5.2.1 for the syntax used to create the *sp.ex1* data set

```
> biofam <- data.frame(biofam, seqST(biofam.seq))
 [>] extracting symbols and durations
 [>] distinct states appearing in the data: 0/1/2/3/4/5/6/7
 [>] alphabet: 1=0 2=1 3=2 4=3 5=4 6=5 7=6 8=7
 [>] 2000 sequences in the data set
 [>] min/max sequence length: 1/5
 [>] Computing turbulence for  2000 sequences, please wait...
```

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.

```
> 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 5.16. 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.

```
hist(biofam$Turbulence,
     col="cyan",
     xlab="Turbulence",
     main="Turbulences for the sequences in the biofam data set")
```

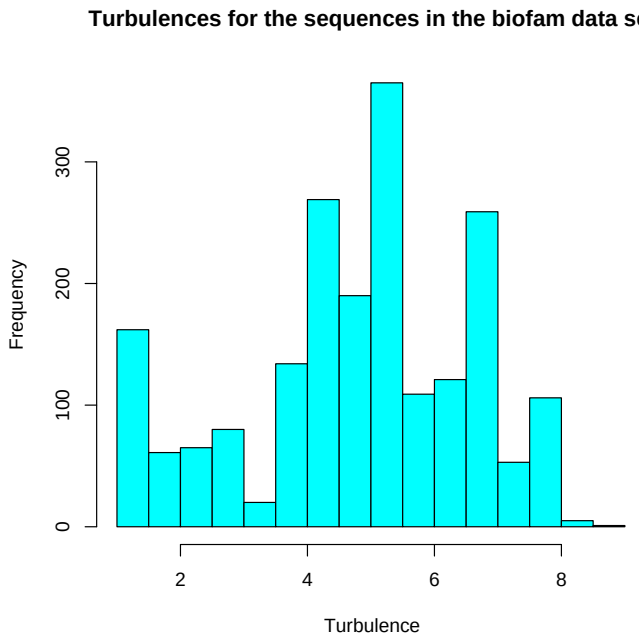


Figure 5.16: Histogram of the sequence turbulences - *biofam* data set

The distribution of the turbulences resembles that of the entropy (see Figure 5.12 on page 62). With the following command we look for the most turbulent sequence.

```
> max.turb <- max(biofam$Turbulence)
> subset(biofam, Turbulence==max.turb)
     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
1098 a few times a year other self-employed   <NA>   0   0   0   0   1   1
     a21 a22 a23 a24 a25 a26 a27 a28 a29 a30 Turbulence
1098   1   1   3   3   3   3   6   6   6   6   8.807355
```

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

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

The sequence with maximum turbulence is not the same as that with maximum entropy (c.f. Section 5.3.6). It contains four subsequences of equal length. This is best shown using the SPS format.

```
> max.seq <- which(biofam$Turbulence==max.turb)
> print(biofam.seq[max.seq,],format='SPS')
 [>] STS sequences converted to 1 SPS seq./rows
    SPS sequence
[1] (0,4)-(1,4)-(3,4)-(6,4)
```

Nonetheless, the correlation between entropy and turbulence measures is reasonably high, wheter we consider the Pearson correlation[4] or the Spearman rank correlation.

```
> cor(biofam$Turbulence,biofam$Entropy)
[1] 0.8078864
> cor(biofam$Turbulence,biofam$Entropy, method='spearman')
[1] 0.731871
```

Figure 5.17 is obtained with the following command and shows the relationship between the two measures.

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

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 1, 49, 51 and 99 percentiles

```
q1  <- quantile(biofam$Turbulence,0.01)
q49 <- quantile(biofam$Turbulence,0.49)
q51 <- quantile(biofam$Turbulence,0.51)
q99 <- quantile(biofam$Turbulence,0.99)
```

and creating three sequence objects containing sequences selected according to their turbulence level

```
turb.min <- biofam.seq[biofam$Turbulence<=q1,]
turb.med <- biofam.seq[biofam$Turbulence>=q49 & biofam$Turbulence<=q51,]
turb.max <- biofam.seq[biofam$Turbulence>=q99,]
```

and next by plotting the first 10 sequences in each of the three object and a legend for the states. The plot is shown in figure 5.18

```
par(mfrow=c(2,2))
seqiplot(turb.min,
         title="10 seq. with low turbulence",
         withlegend=FALSE)
```

[4]The 'pearson' method is the default for the `cor()` function, hence it is not necessary to specify it as an option

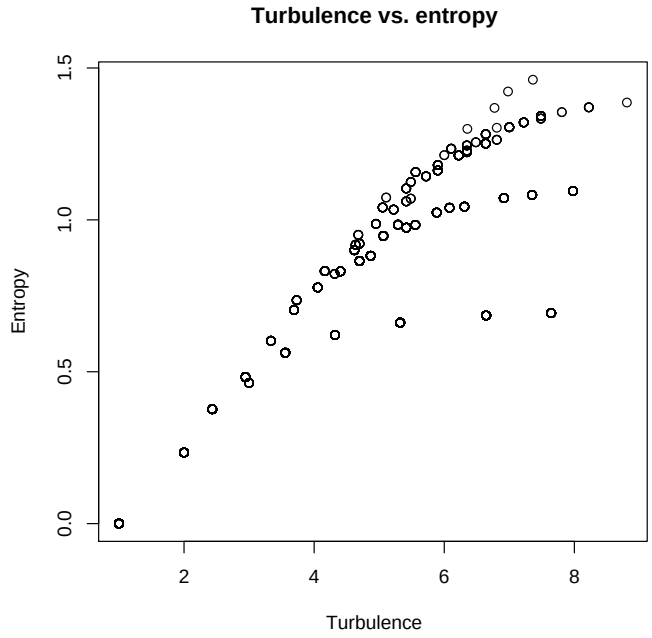


Figure 5.17: Correlation between within sequence turbulence and entropy - *biofam* data set

```
seqiplot(turb.med,
         title="10 seq. with medium turbulence",
         withlegend=FALSE)
seqiplot(turb.max,
         title="10 seq. with high turbulence",
         withlegend=FALSE)
seqlegend(biofam.seq)
```

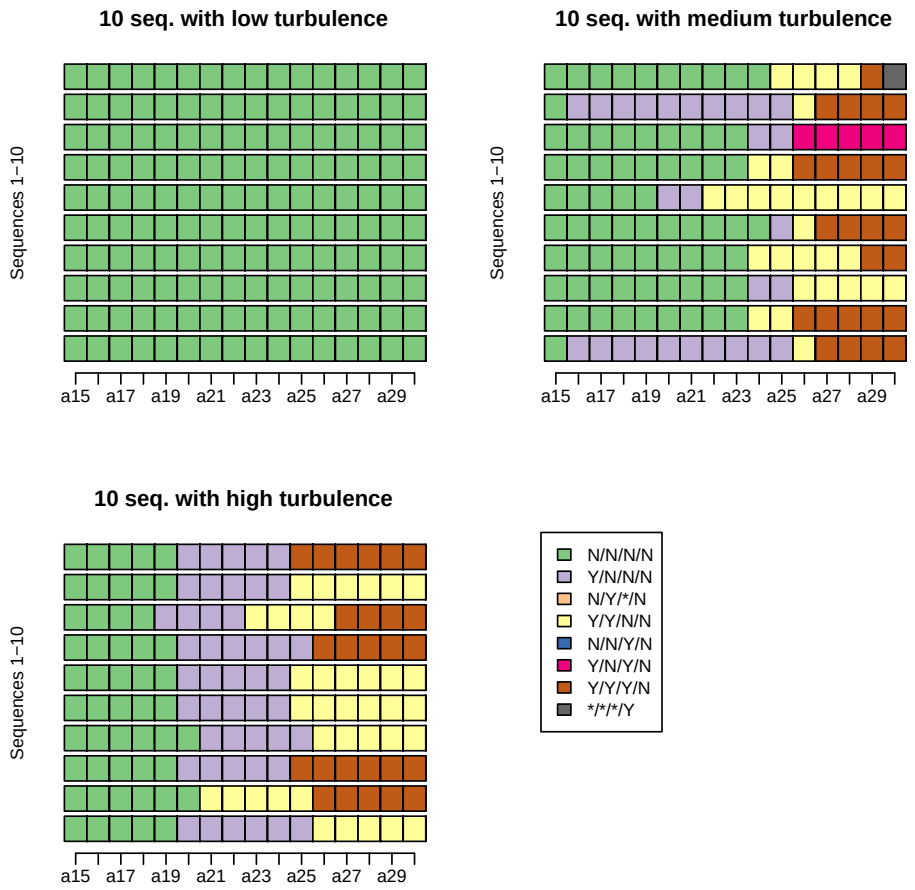


Figure 5.18: Low, median and high sequence turbulences - *biofam* data set

Chapter 6

Measuring similarities and distances between sequences

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.

6.1 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.

```
> data(famform)
> famform.seq <- seqdef(famform)
 [>] distinct states appearing in the data: M/MC/S/SC/U
 [>] alphabet: 1=M 2=MC 3=S 4=SC 5=U
 [>] 5 sequences in the data set
 [>] min/max sequence length: 2/5
> 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
>
> seqmpos(famform.seq[1,],famform.seq[2,])
[1] 2
> seqmpos(famform.seq[2,],famform.seq[4,])
[1] 3
```

6.2 Longest Common Prefix (LCP) distances

The length of the longest common prefix, LCP, is a simple similarity measure measure proposed by Elzinga (2008). The `seqLCP()` function returns the value of this LCP measure for a given couple of sequences. Let us take an example with the *famform* data set. As usual, we first load the data and create a sequence object from it.

```
> data(famform)
> famform.seq <- seqdef(famform)
 [>] distinct states appearing in the data: M/MC/S/SC/U
 [>] alphabet: 1=M 2=MC 3=S 4=SC 5=U
 [>] 5 sequences in the data set
 [>] min/max sequence length: 2/5
> 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
```

Now we compute the LCP for some of the sequences

```
> seqLCP(famform.seq[1,],famform.seq[2,])
[1] 2
> seqLCP(famform.seq[3,],famform.seq[4,])
[1] 4
> seqLCP(famform.seq[3,],famform.seq[5,])
[1] 0
```

The LCP for sequences 1 and 2 is 2, it is 4 for sequences 3 and 4 and 0 for sequences 3 and 5. Based on the LCP, Elzinga proposes as first measure of distance between sequences x and y

$$d_p(x,y) = |x| + |y| - 2\mathcal{A}_p(x,y)|$$

where $\mathcal{A}_p(x,y)$ is the LCP between sequences x and y. 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 part of the matrix in Table 4 in Elzinga (2008):

```
> seqdist(famform.seq,method="LCP")
 [>] 5 sequences with 5 distinct events/states (M/MC/S/SC/U)
 [>] 5 distinct sequences
 [>] min/max sequence length: 2/5
 [>] computing distances using LCP metric ... (0 minutes)
 [>] creating distance matrix ... (0 minutes)
     [,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

$$D_p(x,y) = 1 - S_p(x,y)$$

with

$$S_p(x,y) = \frac{\mathcal{A}_p(x,y)}{\sqrt{|\ x\ | \cdot |\ y\ |}}$$

This normalized metric is obtained with the option `norm=TRUE`

```
> seqdist(famform.seq,method="LCP",norm=TRUE)
 [>] 5 sequences with 5 distinct events/states (M/MC/S/SC/U)
 [>] 5 distinct sequences
 [>] min/max sequence length: 2/5
 [>] computing distances using LCP normalized metric ... (0 minutes)
 [>] creating distance matrix ... (0 minutes)
          [,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.

$$S_p(x, y) = 1 - D_p(x, y)$$

```
> 1-seqdist(famform.seq,method="LCP",norm=TRUE)
 [>] 5 sequences with 5 distinct events/states (M/MC/S/SC/U)
 [>] 5 distinct sequences
 [>] min/max sequence length: 2/5
 [>] computing distances using LCP normalized metric ... (0 minutes)
 [>] creating distance matrix ... (0 minutes)
          [,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 the matrix in Table 4 of Elzinga (2008).

6.3 Longest Common Subsequence (LCS) distances

The Longest Common Subsequence, LCS, based distance is another metric of Elzinga (2008) available through the `seqdist()` function. In the following example, we compute the LCS distances [1] for the *biofam.seq* sequence object previously created from the *biofam* data frame

```
> biofam.lcs <- seqdist(biofam.seq,method="LCS")
 [>] 2000 sequences with 8 distinct events/states (0/1/2/3/4/5/6/7)
 [>] 537 distinct sequences
 [>] min/max sequence length: 16/16
 [>] computing distances using LCS metric ... (0.11 minutes)
 [>] creating distance matrix ... (0.01 minutes)
```

and print the distance matrix for the first 10 sequences

```
> 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
```

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

```
[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
```

6.4 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.

Insertion/deletion cost. The indel cost is a single value specified by the user.

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 (option `method="CONSTANT"`).
- Using the transition rates between states observed in the sequence data (option `method="TRATE"`).

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 75. In the example below, the substitution-cost matrix is generated using the transition rates in the data.

```
> couts <- seqsubm(biofam.seq,method="TRATE")
 [>] creating substitution-cost matrix using transition rates ...
 [>] computing transition rates for states 0/1/2/3/4/5/6/7 ...

> couts
         0        1        2        3        4        5        6        7
0 0.000000 1.945416 1.984656 1.968180 1.999623 1.998931 1.988869 2.000000
```

```
1 1.945416 0.000000 2.000000 1.916578 2.000000 1.996078 1.977362 1.999822
2 1.984656 2.000000 0.000000 1.989674 1.875000 2.000000 1.988880 1.990469
3 1.968180 1.916578 1.989674 0.000000 2.000000 2.000000 1.800558 1.986402
4 1.999623 2.000000 1.875000 2.000000 0.000000 1.937500 2.000000 2.000000
5 1.998931 1.996078 2.000000 2.000000 1.937500 0.000000 1.881944 2.000000
6 1.988869 1.977362 1.988880 1.800558 2.000000 1.881944 0.000000 1.993932
7 2.000000 1.999822 1.990469 1.986402 2.000000 2.000000 1.993932 0.000000
```

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.

```
> range(couts)
[1] 0 2
```

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. In the following example, we use the substitution cost matrix previously computed with the `seqsubm()` command

```
> biofam.om <- seqdist(biofam.seq, method="OM", indel=3, sm=couts)
 [>] 2000 sequences with 8 distinct events/states (0/1/2/3/4/5/6/7)
 [>] 537 distinct sequences
 [>] min/max sequence length: 16/16
 [>] computing distances using OM metric ... (0.13 minutes)
 [>] creating distance matrix ... (0.01 minutes)
```

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[2].

```
> object.size(biofam.om)/1024^2
[1] 30.51768
```

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

```
> 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
```

LCS distance as a special case of OM distance. According to Elzinga (2008), 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 *biofam* data. We begin by creating the substitution matrix with a constant cost of 2, using the `seqsubm()` function with the `method="CONSTANT"` option

[2]The result of the `object.size()` function is in bytes, it is translated into megabytes by dividing it by 1024^2

```
> ccouts <- seqsubm(biofam.seq, method="CONSTANT", cval=2)
 [>] creating 8 x 8 substitution-cost matrix using 2 as constant value
> ccouts
  0 1 2 3 4 5 6 7
0 0 2 2 2 2 2 2 2
1 2 0 2 2 2 2 2 2
2 2 2 0 2 2 2 2 2
3 2 2 2 0 2 2 2 2
4 2 2 2 2 0 2 2 2
5 2 2 2 2 2 0 2 2
6 2 2 2 2 2 2 0 2
7 2 2 2 2 2 2 2 0
```

and compute the OM distances

```
> biofam.om2 <- seqdist(biofam.seq,method="OM",indel=1,sm=ccouts)
 [>] 2000 sequences with 8 distinct events/states (0/1/2/3/4/5/6/7)
 [>] 537 distinct sequences
 [>] min/max sequence length: 16/16
 [>] computing distances using OM metric ... (0.1 minutes)
 [>] creating distance matrix ... (0.01 minutes)
> biofam.om2[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
```

We can see that these displayed OM distances are the same as the extract of the `biofam.lcs` matrix of LCS distances dispalyed in Section 6.3. However, since we may not rely on human brain to compare the two 2000×2000 matrices, we look for a way of checking this more rigorously. This is done with the `all.equal()` function

```
> all.equal(biofam.om2,biofam.lcs)
[1] TRUE
```

Chapter 7

Analysing event sequences

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 for instance 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 just a set of distinct *events* (an event cannot appear more than once in a same transition). A sequence α can be written down as $(\alpha_1 \rightarrow \alpha_2 \rightarrow ... \rightarrow \alpha_q)$, where each α_i is a transition. A sequence with a total of k events is called a *k-sequence*. The distinction between transition and event is a way of accounting for simultaneity of some events.

Agrawal and Srikant (1995) define a subsequence as follows: α is a subsequence of β if each transition of α is a subset of β and the order of the transitions in α respects that in β. We write it down as $\alpha \preceq \beta$. For instance, $(B \rightarrow AC)$ is a subsequence of $(AB \rightarrow E \rightarrow ACD)$ since $B \subseteq (AB)$ — the event B belongs to the transition (AB) — and $(AC) \subseteq (ACD)$ — the events A and C form a subset of the transition (ACD).

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.

TraMineR goes beyond this definition by allowing to specify 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 particular period of the life course, such as the transition to adulthood for instance.

7.1 Creating event sequences

Let us introduce event sequence analysis with a small example. We are interested in analysing frequent transitions occurring in the activity calendar (*actcal* data set). More precisely, we consider the 8 events of the activity calendar between January and December 2000 that are defined in Table 4.2 page 43. These events

are associated to state transitions sequences as described in Table 4.3. The time stamped event data was derived from the state sequences using the process described in Section 4.3.3. Let us just recall here the code used for creating the time stamped event (TSE) data.

```
data(actcal)
transition <- seqetm(actcal[,13:24], method="transition")
transition[1,1:4] <- c("FullTime"         , "Decrease,PartTime",
                       "Decrease,LowPartTime", "Stop")
transition[2,1:4] <- c("Increase,FullTime", "PartTime"         ,
                       "Decrease,LowPartTime", "Stop")
transition[3,1:4] <- c("Increase,FullTime", "Increase,PartTime",
                       "LowPartTime"         , "Stop")
transition[4,1:4] <- c("Start,FullTime"   , "Start,PartTime"   ,
                       "Start,LowPartTime"   , "NoActivity")
actcal.tse <- seqformat(actcal,var=13:24, from='STS',to='TSE',
                        tevent=transition)
```

From the time stamped event data (in TSE format), we create an event sequence object by means of `seqecreate()`.

```
actcal.seqe <- seqecreate(id=actcal.tse$id, time=actcal.tse$time,
                          event=actcal.tse$event)
```

It may be useful to know the time span covered by an event sequence. There are two ways to set this information. We can refer to a special event marking the end of the sequence and use the time until the occurrence of this end event. In that case we specify the end event in `seqecreate()` with the `endEvent` option. Alternatively, we can set the total sequence duration explicitly with the `seqesetlength()` function. It is not mandatory to set this time span.

```
> sl <- numeric()
> sl[1:2000] <- 12
>## All sequences are of length 12
> seqesetlength(actcal.seqe,sl)
> actcal.seqe[1:10]
 [1] (PartTime)-12.00
 [2] (NoActivity)-4.00-(Start,FullTime)-7.00-(Stop)-1.00
 [3] (PartTime)-12.00
 [4] (LowPartTime)-9.00-(PartTime,Increase)-3.00
 [5] (FullTime)-12.00
 [6] (NoActivity)-1.00-(PartTime,Start)-11.00
 [7] (NoActivity)-12.00
 [8] (FullTime)-12.00
 [9] (FullTime)-12.00
[10] (FullTime)-12.00
```

This last step will displays sequences in the 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 second sequence means that the trajectory described starts at time 0 with the "NoActivity" event, which is followed four months later by the events "Start" and "FullTime", and 7 more months later by the event "Stop", which occurs 1 month before the end of the 12 months observation period.

7.2 Searching for frequent event subsequences

The function `seqefsub()` searches for frequent event subsequences. It returns a list containing itself two lists, namely `subseq` a list of frequent event subsequences and `support` a list with the support of these subsequences (the support is the number of event sequences that contain the subsequence). This function takes at least two arguments: A list of event sequences and a minimum support expressed in number of sequences (`minSupport`) or as a percentage by using the `pMinSupport` argument.

```
> fsubseq <- seqefsub(actcal.seqe,minSupport=100)
Step 1:
     Adding sequences (size: 0)
     Simplifying tree (size: 8)
     Tree simplified (size: 6 [added: 6])
Step 2:
     Adding sequences (size: 6)
     Simplifying tree (size: 55)
     Tree simplified (size: 7 [added: 1])
Step 3:
     Adding sequences (size: 7)
     Simplifying tree (size: 16)
     Tree simplified (size: 7 [added: 0])
Counting subseq...(7)
Retrieving subsequences...OK
> fsubseq
$subseq
[1] "(FullTime)"           "(NoActivity)"         "(PartTime)"
[4] "(LowPartTime)"        "(Stop)"               "(Start)"
[7] "(NoActivity)-(Start)"

$support
[1] 929 639 427 307 180 179 131
```

Notice that the subsequences are in the same format as usual event sequences except that they do not hold time information. Hence, the sequence "(NoActivity)-(Start)" means staying first without activity and then starting a new job.

We now use the preceding outcome to compute with the `seqeapplysub()` function the number of occurrences of each frequent subsequence. The `seqeapplysub()` function takes three arguments: a list of subsequences, a list of sequences and a method. 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 show the content of the resulting matrix for subsequences 6 (Start) and 7 (NoActivity)-(Start). Rows of the matrix correspond to the sequences and columns to the specified subsequences `fsubseq$subseq`.

```
> msubcount<-seqeapplysub(fsubseq$subseq, actcal.seqe, method="count")
> #First lines...
> msubcount[1:9,6:7]
                                                 (Start) (NoActivity)-(Start)
(PartTime)                                             0                    0
(NoActivity)-4.00-(Start,FullTime)-7.00-(Stop)         1                    1
(PartTime)                                             0                    0
(LowPartTime)-9.00-(PartTime,Increase)                 0                    0
(FullTime)                                             0                    0
(NoActivity)-1.00-(PartTime,Start)                     1                    1
(NoActivity)                                           0                    0
(FullTime)                                             0                    0
```

```
(FullTime)                                    0                    0
```

7.3 Time constraints

The functions `seqefsub()` (searching frequent subsequences) and `seqeapplysub()` accept time constraints. The following parameters can be set:

`maxGap`: The maximum time gap between two groups of events.

`windowSize`: The maximum window size.

`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:

```
## Using time constraints
## Searching subsequences starting in summer (between June and September)
fsubseq <- seqefsub(actcal.seqe, minSupport=10, ageMin=6, ageMax=9)
fsubseq$subseq[1:10]
## Searching subsequences occurring in summer (between June and September)
fsubseq <- seqefsub(actcal.seqe, minSupport=10, ageMin=6, ageMax=9,
                    ageMaxEnd=9)
fsubseq$subseq[1:10]
## Searching subsequences enclosed in a 6 months period
## and with a maximum gap of 2 months
fsubseq <- seqefsub(actcal.seqe, minSupport=10, maxGap=2, windowSize=6)
fsubseq$subseq[1:10]
```

7.4 Plotting frequencies of event subsequences

The `seqefplot()` function plots the frequencies of a set of subsequences. The following example generates the plot shown in Figure 7.1.

```
## loading data
data(actcal.tse)
## creating sequences
actcal.seqe <- seqecreate(actcal.tse$id,actcal.tse$time,actcal.tse$event)
## Looking for frequent subsequences
fsubseq <- seqefsub(actcal.seqe, pMinSupport=0.01)
## Frequences of 15 first subsequences
seqefplot(fsubseq$subseq[1:15], actcal.seqe, col="cyan")
```

We may also specify a 'group' variable, in which case the `seqefplot()` function provides separate plots of the frequencies of the subsequences by the factor levels of the group variable. The next example generates for instance the plot in Figure 7.2 from which it appears clearly that starting a full-time job is typical for men, while starting a part-time job is typical for women.

```
## Plotting on 2 lines and 3 columns
seqefplot(fsubseq$subseq[1:6],actcal.seqe, group=actcal$sex,
mfrow=c(2,3), col="cyan")
```

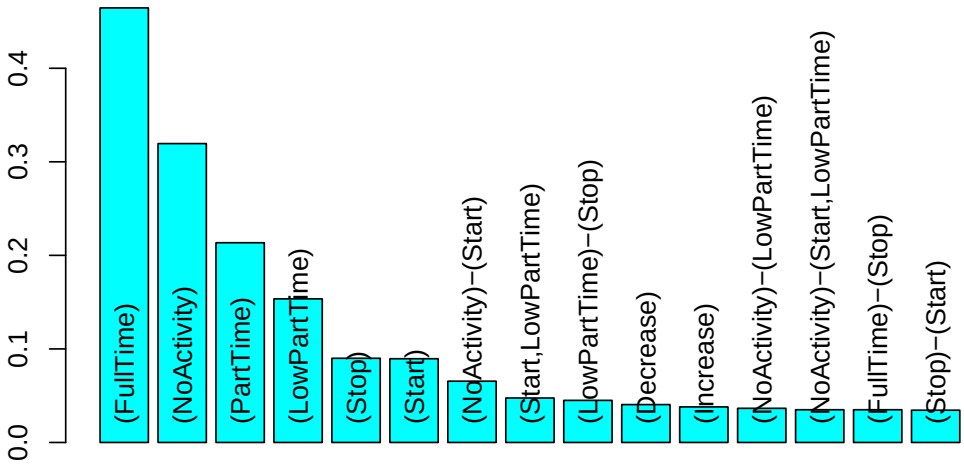


Figure 7.1: Frequencies of 15 most frequent event subsequences

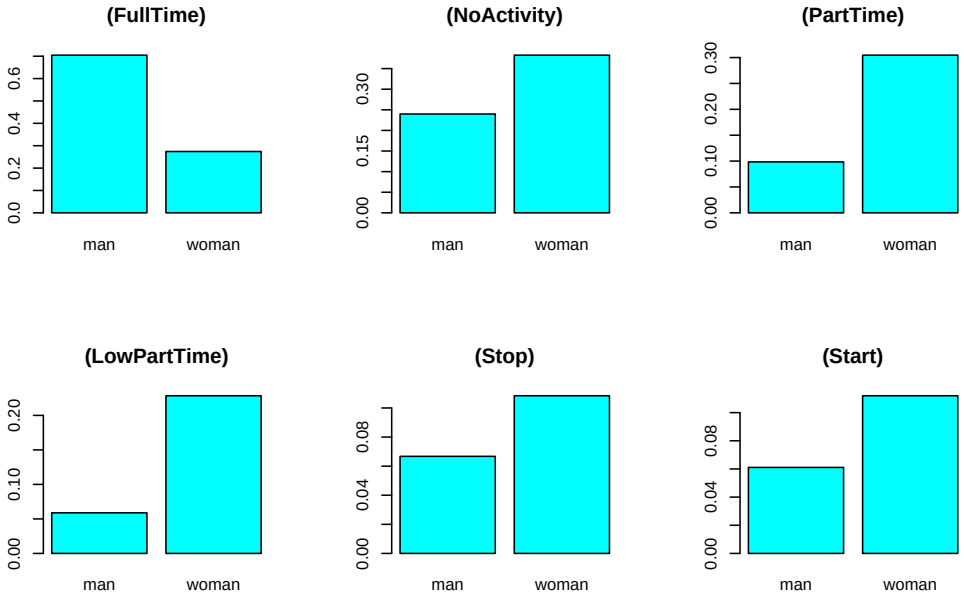


Figure 7.2: Frequencies of first 6 most frequent event subsequences by sex of respondent

7.5 Selecting event subsequences

The function `seqecontain()` helps to select a set of event (sub)sequences. It checks whether a given subsequence contains given events. For instance, we may want to select all subsequences containing the event "Fulltime". The function returns a logical vector with TRUE/FALSE answer for each subsequence.

```
## looking for subsequence with FullTime
seqecontain(fsubseq$subseq, c("FullTime"))
```

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 sequences that contains an event not specified in the `eventList` argument.

7.6 Identifying discriminant event subsequences

The function `seqecmpgroup()` identifies subsequences that are significantly discriminant with the selected test and orders them by decreasing discriminant power. The following methods for testing the discrimination are implemented and can be set through the `method` argument:

1. `bonferroni` determines discrimination on the basis of the Bonferroni corrected p-value of the Chi-square test. With `method=bonferroni`, the function returns the Bonferroni corrected p-values. The correction is based on the number of tests (i.e. the number of subsequences). The parameter `p.valuelimit` can be used to set the p-value threshold (default value is 0.05).

2. `chisq` determines discrimination on the p value without Bonferroni correction. With `method=chisq`, the function returns the value of the Chi-square statistic. Here again, `p.valuelimit` can be used to set the p-value threshold (default value is 0.05).

`seqecmpgroup()` returns a list with three values: The names of the subsequences, the column index numbers of the subsequences in the original matrix and the value of the test statistic or its p-value. In the following example, we look for the most discriminating event subsequences between men and women. The results are then plotted (Figure 7.3).

```
## Looking for subsequences that are present
## in at least 1% (20) of the sequences
fsubseq <- seqefsub(actcal.seqe, pMinSupport=0.01)

## Looking for the discriminating subsequences for sex
discr <- seqecmpgroup(fsubseq$subseq, actcal.seqe, group=actcal$sex,
                      method="bonferroni")
## Plotting the eight most discriminating subsequences in 2 x 4 format
seqefplot(fsubseq$subseq[discr$index[1:8]], actcal.seqe,
          group=actcal$sex, mfrow=c(2,4), col="cyan")
```

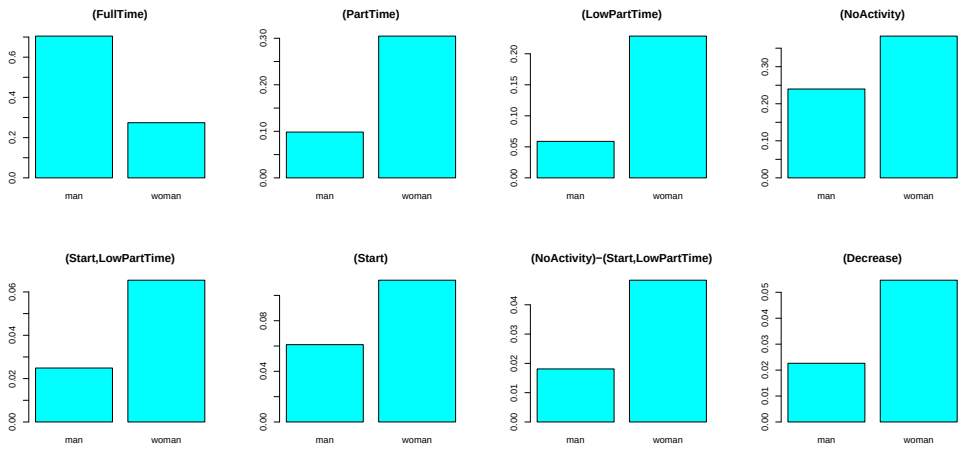


Figure 7.3: Eight most discriminating event subsequences between men and women

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 2. 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

Installing TraMineR

B.1 Installing from binary package

Binary versions of TraMineR are the easiest to install. Such binary versions are available for Linux, MacOS X and Windows. Once the TraMineR package will be available from the CRAN archive (which may not be the case at the time you read this manual), the most straightforward way will be to install it from the CRAN.

B.1.1 Windows

Installing from the CRAN Once the TraMineR package is available from the CRAN archive (it may not be the case at the time you reed this manual), the easiest way to install it is from the Packages menu in R:

1. Run R, if it is not already running.
2. Select Install package(s) from CRAN ... from the Packages menu in R. A window will open asking you to pick a CRAN mirror site for your session; once the mirror is selected, a window will open displaying the various packages available from the CRAN.
3. Using the mouse, select the package or packages that you want to install; if you want to install more than one package, hold down the Control key while you click on the additional packages.
4. When you are finished selecting packages, click the OK button.

Installing from a downloaded zip file Alternatively, you can install binary packages from a previously downloaded zip file:

1. Download the zip file containing the package from http://mephisto.unige.ch/pub/traminer/Windows
2. Run R, if it is not already running.
3. Select Install package(s) from local zip files ... from the Packages menu in R.
4. Navigate to the location of the zip file containing the package.
5. Click the Open button.

B.1.2 Linux

To install the TraMineR package into the standard library location under Linux, you need to be the superuser, otherwise you will get a message like this one:

```
Avis dans install.packages("TraMineR") :
 'lib = "/usr/local/lib/R/site-library"' is not writable
Voulez-vous créer une bibliothèque personnelle
'~/R/i486-pc-linux-gnu-library/2.5'
```

If you don't know what 'superuser' means or want the package to be installed in your home directory or another location, answer 'yes' to the question. In this case, other users of your computer will not be able to use the package, unless they also install it.

Installing from the CRAN

1. Start R.
2. Install from the R command line with the `install.packages()` command. A window will open asking you to pick a CRAN mirror site for your session; once the mirror is selected, a window will open displaying the various packages available from CRAN.
3. Using the mouse, select 'TraMineR' in the list; if you want to install more than one package, hold down the Control key while you click on the additional packages.

Installing from a downloaded tar.gz file

1. Download the tar.gz file containing the package from http://mephisto.unige.ch/pub/traminer/Linux. There are binaries for 32 and 64 bit linux versions. The file names for the 32 bits binaries is `TraMineR_0.*-*_R_i486-pc-linux-gnu.tar.gz` and for the 64 bits versions it is `TraMineR_0.*-*_R_x86_64-pc-linux-gnu.tar.gz`.
2. Start R.
3. Install from the R command line with

   ```
   install.packages(
     '*path*/TraMineR_0.*-*_R_i486-pc-linux-gnu.tar.gz', repos=NULL)
   ```

   where `'*path*/'` is the path to the downloaded tar.gz file. The `repos=NULL` argument must be given for a local install, i.e. one that is not done from a CRAN repository.

B.2 Installing from source package

Though the major part of the package is written in R language, some time-consuming functions (especially those computing distances between sequences) are written in C for better performance. A C compiler is therefore requested for installing TraMineR from source. The installation procedure remains however straightforward.

Note: TraMineR uses some functions provided by other optional R packages, e.g. the RColorBrewer package for creating the color palettes in graphics, **those packages must be installed on your system** in order to compile.

B.2.1 Windows

Under Windows you just need to have the Windows Rtools tool set installed. This tool set includes a C compiler and other tools such as Perl. Thus, for installing from source just proceed as follows:

1. Install the Rtools toolset which can be downloaded from the web page http://www.murdoch-sutherland.com/Rtools/installer.html.

2. Download the package `TraMineR_0.*-*.tar.gz` and remember the name of the directory where the file is saved.

3. Open a DOS terminal (DOS command prompt) and type in

   ```
   cd "C:\Program files\R\R-2.7.0\bin"
   R CMD INSTALL "*path*\TraMineR_0.*-*.tar.gz"
   ```

   The 'cd' DOS command changes the current directory to the folder where the 'R.exe' binary is installed and the second line installs the package. You should indeed adapt the path to the download folder.

B.2.2 Linux

1. The GCC compiler and header files must be installed on your system.

2. Download the package '`TraMineR_0.*-*.tar.gz`' and remember the directory name where the file is saved

3. You must have superuser (root) access for installing the package on the standard library location. Open a terminal and type in (you will be asked for the superuser password):

   ```
   sudo R CMD INSTALL /*path*/TraMineR_0.*-*.tar.gz
   ```

Appendix C

Information about TraMineR content

Below we show the content of the information window as it was obtained with `library(help=TraMineR)` for version 0.4-25 of TraMineR. This information windows 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. You get with `library(help=TraMineR)`.

```
                Information on package 'TraMineR'

Description:

Package:        TraMineR
Version:        1.0
Date:           2008-07-10
Title:          Sequences and trajectories mining for social scientists
Author:         Alexis Gabadinho <alexis.gabadinho@unige.ch>,
                Matthias Studer <matthias.studer@unige.ch>,
                Nicolas S. M\"{u}ller <nicolas.muller@unige.ch>,
                Gilbert Ritschard <gilbert.ritschard@unige.ch>.
Maintainer:     Alexis Gabadinho <alexis.gabadinho@unige.ch>
Depends:        R (>= 2.4.0), RColorBrewer
Suggests:       cluster
Description:    This package is intended for sequence manipulation,
                description and data mining in the field of social
                sciences. In this field, sequences are sets of states or
                events describing life histories, for example family
                formation histories. It provides tools for translating
                sequences from one format to another, statistical
                functions for describing sequences and methods for
                computing distances between sequences using several
                metrics like optimal matching and some other metrics
                proposed by C. Elzinga.
License:        GPL (>= 2)
URL:            http://mephisto.unige.ch/traminer
Packaged:       Thu Jul 10 15:25:32 2008; Jean-Paul II
Built:          R 2.6.1; i386-pc-mingw32; 2008-07-10 15:25:36; windows

Index:
```

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	Retrieve the alphabet of a sequence object
biofam	Example data set: Family life states from the Swiss Household Panel retrospective biographical survey
famform	Example sequences of family formation
read.tda.mdist	Read a distance matrix produced by TDA.
seqLCP	Longest common prefix of two sequences.
seqST	Sequences turbulence
seqconc	Concatenate vectors of states or events into a character string.
seqdcenter	Compute distance to center of a group
seqdecomp	Convert a character string into a vector of states or events.
seqdef	Create a sequence object
seqdim	Returns the dimension of a set of sequences
seqdist	Compute distances between sequences
seqdplot	Graphic presenting the states frequencies
seqdss	Extract distinct states sequence from a sequence object.
seqdur	Extracts states durations from a sequence object.
seqeapplysub	Applying Subsequences to Event Sequences
seqecmpgroup	Identifying discriminating subsequences
seqecontain	Event sequence contain event
seqecreate	Create event sequence objects.
seqefplot	Plot frequencies of subsequences
seqefsub	Searching for frequent subsequences
seqeid	Retrieve id of an event sequence object.
seqelength	Length of event sequences
seqetm	Creating event transition matrix
seqformat	Translation between sequence formats.
seqfplot	Graphic presenting the frequency of sequences
seqfpos	Search for the first occurrence of a given element in a sequence
seqient	Sequence entropy
seqiplot	Visualization of individual sequences.
seqistatd	States frequency for each individual sequence
seqlegend	Plots a legend for the states in a sequence object
seqlength	Sequence length
seqmpos	Number of matching positions between two sequences.
seqnum	Translate a sequence object's alphabet into numerical alphabet, ranging 0-(nbstates-1).
seqpm	Find patterns in sequences
seqsep	Adds separators to sequences stored as character string.
seqstatd	States frequency table and entropy
seqstatl	List of distinct states or events (alphabet) for a sequence data set.
seqsubm	Create a substitution-cost matrix
seqsubsn	Number of distinct subsequences in a sequence.

`seqtab`	`Sequences frequency table`
`seqtrate`	`Transition rates between states.`

Index

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.

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.

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.

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.