Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *doi: 10.18637/jss.v000.i00*

Vignette Template: The goldfish package in R

First Author
ETH Zürich

Second Author
Plus Affiliation

Abstract

This short article illustrates how to write a manuscript for the *Journal of Statistical Software* (JSS) using its LaTeX style files. Generally, we ask to follow JSS's style guide and FAQs precisely. Also, it is recommended to keep the LaTeX code as simple as possible, i.e., avoid inclusion of packages/commands that are not necessary. For outlining the typical structure of a JSS article some brief text snippets are employed that have been inspired by **?**, discussing count data regression in R. Editorial comments and instructions are marked by vertical bars.

Keywords: JSS, style guide, comma-separated, not capitalized, R.

1. Introduction

Social networks change through time, and researchers from various disciplines have an interest in understanding the social mechanisms that drive these dynamics. Often, the data available have the form of relational events where dynamic observations do not stem from consecutively collecting social networks into a panel, but but by observing fine-grained time-stamped interactions between two nodes that may or may not be directly related to social network ties. Examples are automatically collected network data from communication studies or social media studies, social science studies employing social sensors, or archival network studies with detailed information about times of relational actions between two nodes.

Two major conceptual approaches co-exist in statistical network analysis. One approach conveives of the dyad as the unit analyis. The probability of a tie thus depends on its embedding in structures within the social space. The most prominent statistical model

which develops this idea is the he Relational Event Model. The other approach which conceives of ties as controlled by actors that create or propose ties is the Stochastic Actor Oriented Model.These two approaches have also been put forward to studying time-stamped network data. The Relational Event Model by Butts (2008) is essentially in the tradition of tie-oriented models.

2. Relational Event Models

3. Dynamic Network Actor Models (DyNAMs)

The actor-oriented models that **goldfish** implements has been called Dynamic Network Actor Models (DyNAMs).

3.1. DyNAMs for directed relational events

(Stadtfeld and Block 2017)

3.2. DyNAMs for coordination ties

(Stadtfeld, Hollway, and Block 2017)

3.3. Comparing DyNAMs to the Relational Event Model (REM)

4. Data Preparation

4.1. Example: The Social Evolution data

Loading the goldfish package:

```
R> library(goldfish)
```

Loading the data:

```
R> data("Social_Evolution")
```

An initial look at the data:

```
R> head(calls)
```

```
        time   sender receiver increment
1 1220733470 Actor 72 Actor 50         1
```

```
2 1221102974 Actor 43 Actor 51         1
3 1221784293 Actor 43 Actor 51         1
4 1221785882 Actor 43 Actor 22         1
5 1221787264 Actor 43 Actor 55         1
6 1221848443 Actor 43 Actor 51         1
```

The `Social_Evolution` dataset is an abbreviated version of the MIT Reality Commons Social Evolution dataset, spanning a reduced time period and with fewer variables. Dyadic variables include binary friendships at time of survey, and time-stamped phone call occurrences. Individual variables include the floor of the dormitory on which the student resides, and the grade type of each student including freshmen, sophomore, junior, senior, or graduate tutors.

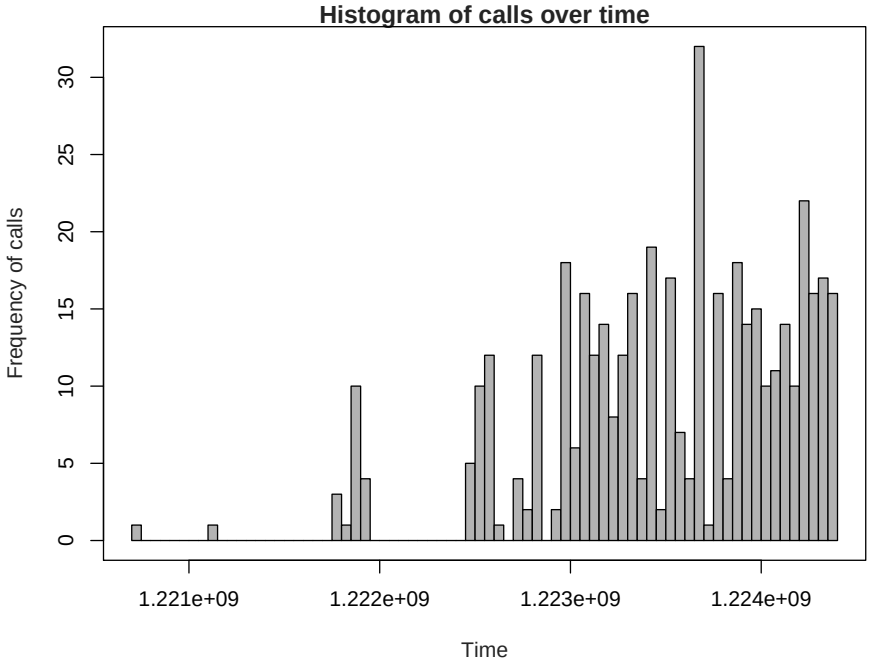

Figure 1: Frequency distribution for number of calls over time.

For simplicity and exhaustivity, the following code examples are run on a small data frame reconstructed from a subset the data `Social_Evolution` provided by **goldfish**.

4.2. Creating Goldfish objects

Defining the nodeset

The `nodes` argument requires a data.frame with information about the nodes that are present throughout the time window of interest. This data.frame must contain at least one vector, which is of the same length as the number of nodes. Other than that it can

contain information about the nodes -`label` (e.g., Actor 1), whether the node is `present` at the start of the data collection as well as the the initial (at T0), or non-changing values for node attributes. For instance, this data.frame could look like this:

```
     label present floor gradeType gender
1  Actor 1    TRUE     4         2      2
2  Actor 2    TRUE     4         2      2
3  Actor 3    TRUE     1         4      2
4  Actor 4    TRUE     8         1      2
5  Actor 5    TRUE     3         1      2
6  Actor 6    TRUE     3         2      2
7  Actor 7    TRUE     4         2      1
8  Actor 8    TRUE     5         5      2
9 Actor 12    TRUE     2         4      2
```

The `defineNodes` function processes and checks the data.frame passed to `nodes` argument.

```
R> actors <- defineNodes(nodes = actors)
```

Please note that **goldfish** uses the rownames of this data.frame as the internal node IDs. The `label` variable must be characters.

If you have changing node-attributes you can link `ChangeEvents` data.frames to the nodeset.

The `changeEvents` data.frames mentioned above must be in the following format:

```
     time node replace
1 2484870    5   FALSE
2 2487266    5    TRUE
```

Then, `ChangeEvents` can be linked to the nodeset in the following way:

```
R> actors_ca <- linkEvents(x = actors,
+                          attribute = "present",
+                          changeEvents = compositionChangeEvents)
```

The `linkEvents` function links events lists of changing attributes (e.g., present, gender) to the nodeset. The nodeset can be specified with the argument `x`, the attribute that is changed with the events list has to be defined in the `attribute` argument as a string. The `changeEvents` contains a data.frame that describes when and how the nodes attributes change. If no data is linked to an attribute, the attribute variables specified in the `nodes` arguments are assumed to stay constant (e.g., gender).

> Note that `present`, if given as a node attribute, must be true at a given time point for any node to be involved in an event. In the case of a friendship and phone call network with other dynamic actor attributes, someone losing their phone should not be specified as present = false, since then their friendships and individual attributes cannot change. A separate attribute and corresponding event list should be added in such situations that only one type of event cannot occur for a given actor.

Here the `time` column in numeric (presented in seconds) or POSIXct format indicates when the change happens. The `node` column must correspond with either the row number of the nodes data.frame or the label string. The column `replace` contains boolean values indicating whether the node is present at that time. For instance, the example above shows that node 5 leaves the network at 2484870 seconds but joins back the network at 2487266 seconds.

It is also possible to link other types of changing attributes in the nodeset, either in numeric or character format (e.g people changing floors). Moreover, in the case of a numeric attribute, we could also add up incremental values to the existing ones by using a column named `increment` instead of `replace`

Define the network

Now that we have defined our nodeset, we want to define a network between the nodes according to an event list. To do so, a second data.frame is needed, which consists in an edgelist of events happening between actors (e.g tie creations). While the nodes argument must be an object defined with the `defineNodes` function, the event list should be a data.frame containing the following columns:

```
      time  sender receiver increment
1  1936225 Actor 5  Actor 8         1
2  2484877 Actor 7  Actor 4         1
3  2484925 Actor 7  Actor 1         1
4  2485369 Actor 7  Actor 6         1
5  2485377 Actor 6  Actor 7         1
6  2487252 Actor 7  Actor 2         1
7  2522527 Actor 3 Actor 12         1
8  2523087 Actor 3 Actor 12         1
9  2524522 Actor 3 Actor 12         1
10 2524804 Actor 3 Actor 12         1
11 2534595 Actor 3 Actor 12         1
```

As a first step, we define a network object from the nodeset:

```
R> callNetwork <- defineNetwork(nodes = actors_ca, directed = T)
```

The argument "directed" is `TRUE` by default, but we need to specify the nodes so that **goldfish** can check for consistency and relate it to that nodeset as needed.

Note that we have not added any network data yet. By default, defineNetwork() just constructs an empty matrix with dimensions defined by the length of the nodeset(s). So we have an empty network as a starting state.

Now that **goldfish** recognises the matrix as a network, we can also associate an event list that updates it. To do this we use the `linkEvents` function, which requires us to identify a goldfish object to be updated, the events that update it and, in this case, also the nodes that the events should relate to.

```
R> callNetwork <- linkEvents(x = callNetwork, changeEvent = calls_subset,
+                            nodes = actors_ca)
```

Define the independent networks

Networks that function as independent variables can also be defined with the following function:

```
R> friendshipNetwork <- defineNetwork(matrix =friendship.w1, nodes = actors_ca,
+                                     directed = T)
```

Here the sociomatrix `friendship.w1` describes the initial state of the network at T0.

Table 1: Friendship Sociomatrix Subset

friendship.w1	Actor 1	Actor 2	Actor 3	Actor 4	Actor 5	Actor 6	Actor 7	Actor 8	Actor 12
Actor 1	0	0	0	0	0	1	1	0	0
Actor 2	0	0	0	0	0	0	0	0	0
Actor 3	0	0	0	0	0	0	0	0	1
Actor 4	0	0	0	0	0	0	0	0	0
Actor 5	0	0	0	0	0	0	0	0	0
Actor 6	1	0	0	0	0	0	1	0	0
Actor 7	1	1	0	0	0	1	0	0	0
Actor 8	0	0	0	0	0	0	0	0	0
Actor 12	0	0	1	0	0	0	0	0	0

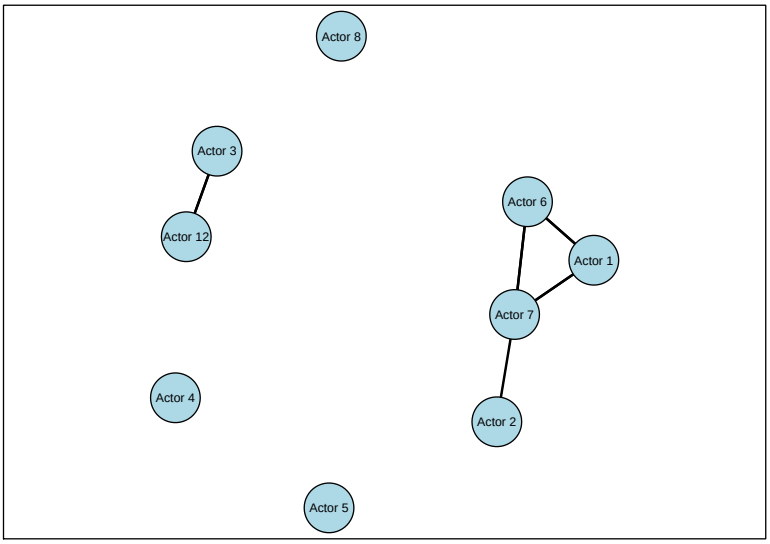

Figure 2: Network representation of the Friendship sociomatric subset.

This matrix must contain the same nodeset as defined with the `defineNodes` function and the order of the rows must correspond. The matrix must be binary(**?**) and can be directed or undirected (as specified with the directed argument). If no matrix is provided, **goldfish** assumes the initial state to be empty (i.e., a matrix containing only 0s).

If this network is updated over time (e.g., a new wave of friendship data is collected), these changes can be added with the `linkEvents` function - similar to link changing attribute events to a nodeset. This time, the user needs to provide the network and the associated nodeset.

The data.frame passed to the changeEvents argument must be in the following format:

```
R> friendshipChange
```

```
     time  sender receiver replace
1 4374400 Actor 1  Actor 4       1
2 4374400 Actor 7  Actor 4       1
3 4374400 Actor 6  Actor 1       0
4 4374400 Actor 7  Actor 1       0
```

Then the events can be linked again.

```
R> friendshipNetwork <- linkEvents(x = friendshipNetwork,
+                                  nodes = actors_ca,
+                                  changeEvents = friendshipChange)
```

Alternatively to the `replace` column, a column called `increment` can be used, as we mentioned earlier. For friendship networks, 1 = tie creation, -1 = tie dissolution. The columns `sender` and `receiver` can be in the format numeric or character. For the numeric case, the entries must correspond to the rownumbers of the nodes data.frame. When character strings are used, they must correspond to the labels in the nodes data.frame. In the example shown above, Actor 4 nominates actor 7 as a friend at timepoint 100 and at timepoint 103 Actor 3 drops the tie to Actor 2.

Global attributes

Imagine that you want to include a variable that is identical for each node but changes over time. For instance, seasonal climate changes. Such a changing global attribute can be defined with the `defineGlobalAttribute` function.

```
R> seasons <- defineGlobalAttribute(data.frame(time = 1:12, replace = 1:12))
```

Define the dependent event list

The final step in defining the data objects is to identify the dependent events. Here we would like to model the calls between individuals as the dependent variable using `calls_subset`.

While the nodes argument must be an object defined with the `defineNodes` function, the dependent events defined by `defineDependentEvents` should take a data.frame as input, containing the following columns:

```
R> callsDependent <- defineDependentEvents(events = calls_subset,
+                                          nodes = actors_ca,
+                                          defaultNetwork = callNetwork)
```

Object visualisation and plot options

5. Estimation

- Explanation of the rate and the choice models in the goldfish model
- Formulas and mathematical description of each model.
- The following code serves as an example and application to the previously explained models.

In previous section we have defined our dependent network, in the example case this was the call network. The next stage is to specify and fit a model. This step will be performed on the `Social_Evolution` dataset. It can be broken up into three steps: the model specification, the preprocessing and the estimation.

The three first steps have been presented in the previous sections.

5.1. Step 1: Model specification

The first step of the estimation procedure consists in specifying our model from the effects and variables available using the standard R formula.

In the case of the rate model, the formula can be specified with attribute effects:

```
R> formula_rate<-(callsDependent ~ alter(actors$gradeType)
+                 + same(actors$floor))
```

while in the choice model, we would rather specify the formula with network effects, to which we can add arguments, such as `window` and `ignoreRep`.

```
R> formula_choice<-(callsDependent ~ inertia(callNetwork, window=3600)
+                   + recip(callNetwork, ignoreRep=T)
+                   + trans(callNetwork, weighted=T))
```

Beyond the tilde we specify effects and the variables for which the effects are expected to occur. To see the possible effects, use the function `goldfishEffects`.

5.2. Step 2: Preprocessing

The second step consists of preprocessing which calculates the change statistics associated with the chosen effects.To have a look at the preprocessed objects, we need to activate the `preprocessingOnly` argument to `TRUE`:

```
R> (DynamM_est_rate<- estimate(formula_rate, model="DyNAM",
+                               subModel="rate",
+                               preprocessingOnly = T))
```

The same can be done for the choice model simply by replacing `subModel="rate"` by `subModel="choice"`.

5.3. Step 3: Estimation

The third step consists in estimation. The `estimate` function fit an appropriate model to these statistics.

The rate model can be estimated by specifying the model as `"DyNAM"` and the submodel as `"choice"`:

```
R> (DynamM_est_rate<- estimate(formula_rate, model="DyNAM",
+                               subModel="rate"))

   name               V2         est  std.err sig          t
1 alter actors$gradeType -19.8746576 3.003861 *** -6.6163698
2  same     actors$floor   0.9876948 1.298199      0.7608191
  Log likelihood -1923.3349
  Converged with max abs. score of 8e-05
  AIC  3850.66981
  BIC  3858.83881
  Model type: DyNAM-M-Rate
```

Again, by replacing `"rate"` by `"choice"` in the `subModel` argument, we can obtain the estimation results for the choice model:

```
R> (DynamM_est_choice<- estimate(formula_choice, model="DyNAM",
+                                 subModel="choice"))

     name               V2 ignoreRep weighted window       est
1 inertia callNetwork_3600                      3600 6.2793659
2   recip      callNetwork         B                2.9821517
3   trans      callNetwork                 W        0.3234464
    std.err sig         t
1 0.2132116 *** 29.451331
2 0.2679389 *** 11.129970
3 0.1924422      1.680745
  Log likelihood -1232.9717
  Converged with max abs. score of 9e-05
  AIC  2471.94344
  BIC  2484.19694
  Model type: DyNAM-M
```

However, in **goldfish** we also have the option of accelerating this process and using memory more efficiently by combining these three sub-steps in one. Nonetheless, it can be helpful to think of 2a separately, and recognise steps 2b and 2c as **goldfish** does them.

5.4. Additional models

Note that **goldfish** provides additionnal models such as the (`"coordination"`) submodel (Stadtfeld, Hollway and Block, 2017) and the `"REM"` model(Butts, 2008).

6. Interpreting and diagnosing results

7. Extending DyNAMs and REMs with own effects

8. Conclusions

9. Summary and discussion

- As usual ...

Computational details

If necessary or useful, information about certain computational details such as version numbers, operating systems, or compilers could be included in an unnumbered section. Also, auxiliary packages (say, for visualizations, maps, tables, ...) that are not cited in the main text can be credited here.

The results in this paper were obtained using R 3.6.0 with the **MASS** 7.3.51.4 package. R itself and all packages used are available from the Comprehensive R Archive Network (CRAN) at https://CRAN.R-project.org/.

Acknowledgments

All acknowledgments (note the AE spelling) should be collected in this unnumbered section before the references. It may contain the usual information about funding and feedback from colleagues/reviewers/etc. Furthermore, information such as relative contributions of the authors may be added here (if any).

References

Butts CT (2008). "A relational event framework for social action." *Sociological Methodology*, **38**(1), 155–200.

A. More technical details

Appendices can be included after the bibliography (with a page break). Each section within the appendix should have a proper section title (rather than just *Appendix*).

For more technical style details, please check out JSS's style FAQ at https://www.jstatsoft.org/pages/view/style#frequently-asked-questions which includes the following topics:

- Title vs. sentence case.
- Graphics formatting.
- Naming conventions.
- Turning JSS manuscripts into R package vignettes.
- Trouble shooting.
- Many other potentially helpful details...

B. Using BibTeX

References need to be provided in a BibTeX file (`.bib`). All references should be made with `\cite`, `\citet`, `\citep`, `\citealp` etc. (and never hard-coded). This commands yield different formats of author-year citations and allow to include additional details (e.g., pages, chapters, ...) in brackets. In case you are not familiar with these commands see the JSS style FAQ for details.

Cleaning up BibTeX files is a somewhat tedious task – especially when acquiring the entries automatically from mixed online sources. However, it is important that informations are complete and presented in a consistent style to avoid confusions. JSS requires the following format.

- JSS-specific markup (`\proglang`, `\pkg`, `\code`) should be used in the references.
- Titles should be in title case.
- Journal titles should not be abbreviated and in title case.
- DOIs should be included where available.
- Software should be properly cited as well. For R packages `citation("pkgname")` typically provides a good starting point.

Affiliation:

Achim Zeileis
Journal of Statistical Software
and
Department of Statistics
Faculty of Economics and Statistics
Universität Innsbruck
Universitätsstr. 15
6020 Innsbruck, Austria
E-mail: Achim.Zeileis@R-project.org
URL: https://eeecon.uibk.ac.at/~zeileis/

Journal of Statistical Software http://www.jstatsoft.org/
published by the Foundation for Open Access Statistics http://www.foastat.org/
MMMMMM YYYY, Volume VV, Issue II *Submitted:* yyyy-mm-dd
doi:10.18637/jss.v000.i00 *Accepted:* yyyy-mm-dd