`EloRating` - a brief tutorial

Christof Neumann & Lars Kulik

2020-03-12 (v. 0.46.11)

Contents

Preliminary remarks

The `EloRating` package is work in progress. If you have any criticism, suggestions or bugs to report, please let us know.

We describe here the main functionality of the `EloRating` package. For the sake of this tutorial, we first present an example with the minimal amount of data required: a sequence of decided dominance interactions along with the dates of these interactions.[1] Even though the package is capable of dealing with undecided interactions (in fact the example file contains this information), we decided to omit this aspect for the sake of clarity in the first part (section Using `EloRating`). In addition, this first example is not linked to 'presence' data. In other words, here we assume that all individuals that occur in the data set were present over the entire study period. For the same example utilizing information about presence/absence of individuals and undecided interactions/draws see section on presence data and undecided interactions.

In the section Further notes on the stability index, we present a detailed description of the updated stability index S.

The fictional data set presented here comprises 250 dominance interactions of 10 individuals.

Requirements and package installation

We recommend that you have a fairly recent version of R available (v > 3.2).

To install the package, just use (given you have a working internet connection):

```r
install.packages("EloRating")
```

There are a number of additional packages that are required before `EloRating` will work. If you used the approach just shown, this is nothing to worry about because all the required packages will be installed alongside `EloRating`.

If you want the latest development version of the package you can install it from GitHub. For this to work, you need at least the following packages installed: `zoo`, `sna`, `network`, `Rcpp`, `RcppArmadillo`, `knitr`, `Rdpack` and `devtools`. It might be necessary to install these first before you can install the GitHub version of `EloRating`, e.g. by `install.packages("zoo")` etc. Furthermore you will need a set of tools:

- **Windows**: Rtools from https://cran.r-project.org/bin/windows/Rtools/
- **Mac**: Xcode from the App Store

```r
library(devtools)
install_github("gobbios/EloRating")
install_github("gobbios/EloRating", build_vignettes = TRUE) # with pdf tutorial
```

Data preparation

We assume that you store your data on dominance interactions in some sort of spreadsheet software. While it is possible to read data directly from Excel files (.xls or .xlsx) or SPSS files (.sav),[2] we suggest that you store your data in simple (tab-separated) text files. For example, from Excel this is possible via File>Save as... and then choosing 'tab-delimited text file' as file format.[3]

In the simplest case, you need three columns in your data set, one for the date and one each for winner and loser IDs. Note that the interactions have to be in the correct sequence, i.e. sorted by date (and time if available): **the functions in this package will not sort the data according to dates.**[4] The actual names of the columns are not fixed, so you can use whatever you want as long as they conform to naming rules of column names in R (start with a letter, no spaces, etc.).

Date	winner	loser
2000-01-01	d	w
2000-01-01	k	w
2000-01-01	n	z
2000-01-07	k	n

[1] Dealing with calendar dates in 'R' is prone to unexpected behaviour. We decided to stick to a specific format ("YYYY-MM-DD") and the functions assume that dates appear in this format in the objects from which the functions work.

[2] see the R packages `gdata`, `xlsx`, `readxl` and `foreign`

[3] you may also save your file as comma delimited or something similar, but note that you then may need to modify the arguments to `read.table()` or use `read.csv()`

[4] But the `seqcheck()` function will check whether interactions are sorted by date, see below.

Date	winner	loser
2000-01-07	c	g
2000-01-07	n	g

Optional columns that may be required for a more refined assessment of ratings are a column for draws and for interaction type (intensity). The `Draw` column should contain only `TRUE` and `FALSE` to indicate whether an interaction ended undecided/tied.

Date	winner	loser	Draw	intensity
2000-01-01	d	w	FALSE	fight
2000-01-01	k	w	TRUE	threat
2000-01-01	n	z	TRUE	threat
2000-01-07	k	n	FALSE	fight
2000-01-07	c	g	FALSE	threat
2000-01-07	n	g	FALSE	fight

Using `EloRating`

Start by loading the package and reading the raw data.[5]

```
library(EloRating)
```

```
xdata <- read.table(system.file("ex-sequence.txt", package = "EloRating"), header = TRUE)
```

Keep in mind that as soon as you use your own data it might be necessary to include absolute paths with the file name.[6] For example:

```
# on Windows
xdata <- read.table("c:\\temp\\ex-sequence.txt", header = TRUE, sep = "\t")
# on Mac
xdata <- read.table("~/Documents/ex-sequence.txt", header = TRUE, sep = "\t")
```

Data checks

We then go on and check whether the data meet the formatting requirements for the remaining functions of the package to work. If there is something not quite right with your data, this function will tell you. 'Warnings' can sometimes be ignored (see below), whereas 'errors' need to be fixed before the next step. More details on the possible warning and error messages can be found in the help files (`?seqcheck`).

```
seqcheck(winner = xdata$winner, loser = xdata$loser, Date = xdata$Date)
```

```
## No presence data supplied
## Everything seems to be fine with the interaction sequence...OK
```

Elo-rating calculations

This doesn't give any error message, and so we can go on and calculate the actual Elo-ratings and store the results of the calculations in an object we name `res`. Note that in order to ignore possible warnings from `seqcheck()` the argument `runcheck = FALSE` has to be set.

```
res <- elo.seq(winner = xdata$winner, loser = xdata$loser, Date = xdata$Date, runcheck = TRUE)
summary(res)
```

```
## Elo ratings from 10 individuals
## total (mean/median) number of interactions: 250 (50/49)
## range of interactions: 19 - 75
## date range: 2000-01-01 - 2000-09-06
## startvalue: 1000
## uppon arrival treatment: average
## k: 100
## proportion of draws in the data set: 0
```

[5] The example files are in the above described tab-delimited text format and can be found in the package directory. If you don't know where that is check `.libPaths()`

[6] see also `?setwd`

Extract Elo-ratings

The most obvious task perhaps is to obtain Elo-ratings of all or a specific set of individuals on a specific date. This can be achieved by running the function `extract.elo()` on the object `res` that we just created. In the output, individuals are ordered by descending Elo-ratings.

```
extract_elo(res, extractdate = "2000-05-28")
```

```
##    c    d    a    f    k    s    g    n    w    z
## 1342 1214 1161 1133 1011 1000  958  844  799  538
```

```
extract_elo(res, extractdate = "2000-05-28", IDs = c("s", "a", "c", "k"))
```

```
##    c    a    k    s
## 1342 1161 1011 1000
```

If you omit the arguments regarding dates and/or individuals, the function will return the ratings of all individuals on the last day of the study period.

```
extract_elo(res)
```

```
##    s    a    c    d    f    g    k    n    w    z
## 1381 1362 1205 1084 1077 1049  940  771  681  450
```

```
# the same as because 2000-09-06 is the last date in the sequence:
extract_elo(res, extractdate = "2000-09-06")
```

```
##    s    a    c    d    f    g    k    n    w    z
## 1381 1362 1205 1084 1077 1049  940  771  681  450
```

Finally, it is possible to extract ratings to matching combinations of IDs and dates. This comes handy if you have a data set ready that contains measurements of some variable of interest on different dates and/or for different individuals. Let's look at an example:

```
parasites <- read.table(system.file("ex-parasites.txt", package = "EloRating"), header = TRUE)
parasites$Date <- as.Date(as.character(parasites$Date))
head(parasites)
```

```
##   id       Date parasites
## 1  c 2000-01-04         2
## 2  n 2000-01-04         3
## 3  n 2000-01-07         2
## 4  n 2000-01-09         2
## 5  w 2000-01-11         4
## 6  n 2000-01-13         1
```

In this fictional example, I counted ecto-parasites for certain individuals on different days[7]. You can see that among the six lines of data I've shown, there is data for two individuals on the first day and individual n was observed on multiple days. The entire data set contains 46 observations. In all, this is a fairly typical example of data set that I encounter regularly. Now, the crucial thing to do here is to get for each individual its rating on the day of the respective observation. To do so, we need to supply to `extract_elo()` a vector of dates and a vector of IDs. For this, we simply take the columns in our data set `parasites` and write the values as a new column in that data frame:

```
parasites$elo <- extract_elo(res, extractdate = parasites$Date, IDs = parasites$id)
head(parasites)
```

```
##   id       Date parasites  elo
## 1  c 2000-01-04         2 1043
## 2  n 2000-01-04         3 1000
## 3  n 2000-01-07         2  907
## 4  n 2000-01-09         2  907
## 5  w 2000-01-11         4 1034
## 6  n 2000-01-13         1  870
```

With this, we can then produce a figure and perhaps run some model (e.g. in figure 1). Again, this is a typical thing to do in my work.

Plotting Elo-ratings

`eloplot()` produces quick plots that visualize the development of Elo-ratings over time. Note that the example data set contains a rather modest number of interactions and individuals. With larger data sets (both in terms of interactions

[7] e.g. Duboscq et al. (2016)

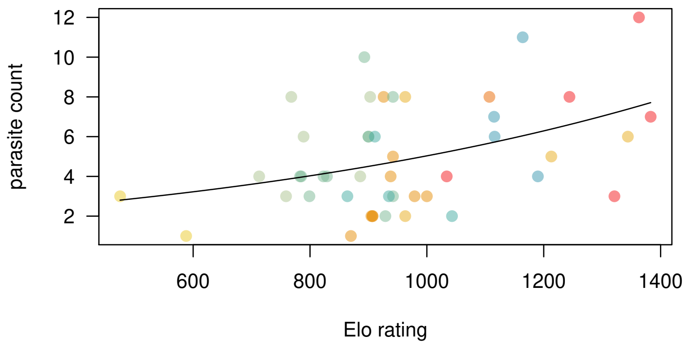

Figure 1: Parasite count as a function of day-specific Elo ratings. Each individual has its own colour. Code to produce the figure is in the Appendix.

and individuals), such plots can become messy quickly. Even though it is possible to restrict plotting to date ranges and subsets of individuals, we recommend to create custom plots by directly accessing the `res` object. Specifically, `res$mat` contains raw Elo-ratings in a day-by-ID matrix, while the original dates can be found in `res$truedates`. You can find more details on how to proceed with custom figures in the section on custom figures.

The following code produces figure 2.[8]

```
eloplot(res)
```

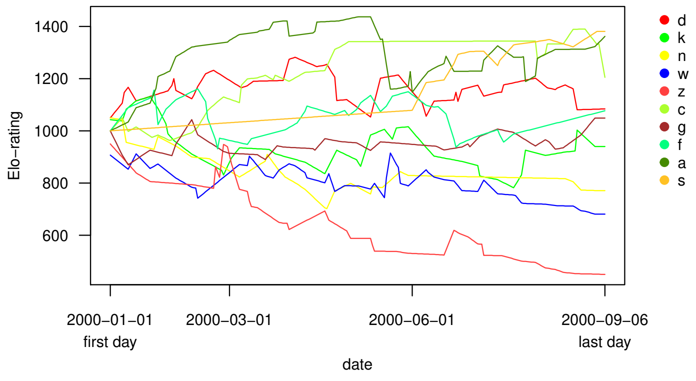

Figure 2: Elo-ratings of 10 individuals over the entire study period.

Restricting the date range and selecting only a subset of individuals results in figure 3.

```
eloplot(eloobject = res, ids = c("s", "a", "w", "k", "c"), from = "2000-06-05", to = "2000-07-04")
```

Please note, `eloplot()` will plot a maximum of 20 individuals. This is because we meant the plotting function to be an exploratory tool, but you can also select `ids="random.20"` if you have more than 20 individuals. Please note also that individuals for which you have observed interactions on only one day in the selected date range[9], such individuals will be omitted from the plot. If you wish to plot such individuals as single points in the graph, you will have to use the approach mentioned above, i.e. use the `res$mat` and `res$truedates` objects. If you need help with that, please get in touch with us or have a look at the section on customizing figures.

Incorporating presence data and undecided interactions

This section demonstrates how to incorporate presence data and undecided interactions. Please note that the presence data needs to cover *every* day during your data collection, i.e. also those days on which no interactions were observed. Also, the column that contains the dates must be labelled `Date`, otherwise you will receive an error message. We start

[8]If you look carefully at figure 2, you can see that there are multiple individuals which have long, very straight lines, for example individual s from the beginning up to about June 2000. There could be two reasons for that. Either that individual was not present during that time (and hence could not interact), or it was present and was simply not observed interacting. In this case, s was not actually present in the group during this time, but the plotting function won't know that unless you supply the relevant data (we will deal with this further below).

[9]regardless of how many interactions such individuals may have had on such days!

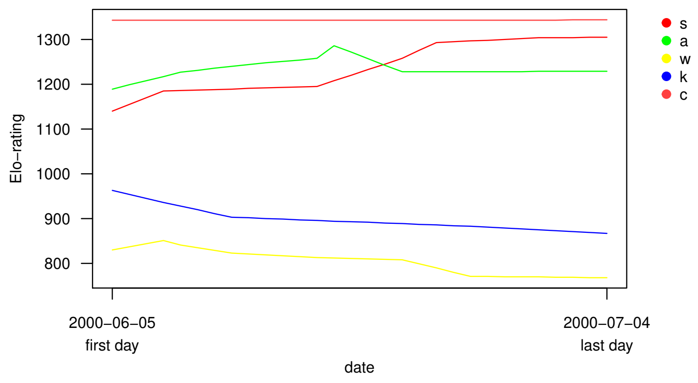

Figure 3: Elo-ratings of 5 individuals over a month.

by reading the additional 'presence matrix', followed by reformatting the date column in this object to a date format that R is capable of dealing with. And then, just to get a feeling for how these data are supposed to look like, we look at first few lines.

```r
xpres <- read.table(system.file("ex-presence.txt", package = "EloRating"), header = TRUE)
xpres$Date <- as.Date(as.character(xpres$Date))
head(xpres)
```

```
##         Date a c d f g k n s w z
## 1 2000-01-01 1 1 1 1 1 1 1 0 1 1
## 2 2000-01-02 1 1 1 1 1 1 1 0 1 1
## 3 2000-01-03 1 1 1 1 1 1 1 0 1 1
## 4 2000-01-04 1 1 1 1 1 1 1 0 1 1
## 5 2000-01-05 1 1 1 1 1 1 1 0 1 1
## 6 2000-01-06 1 1 1 1 1 1 1 0 1 1
```

Next, we rerun `seqcheck()` and `elo.seq()` with the additional `presence=` argument as well as incorporating the information about undecided interactions `draw=` into the latter function.

```r
seqcheck(winner = xdata$winner, loser = xdata$loser, Date = xdata$Date, presence = xpres, draw = xdata$Draw)
```

```
## Presence data supplied, see below for details
## Everything seems to be fine with the interaction sequence...OK
##
## #####################################
##
## Presence data seems to be fine and matches interaction sequence...OK
##
## #####################################
```

```r
res2 <- elo.seq(winner = xdata$winner, loser = xdata$loser, Date = xdata$Date, presence = xpres, draw = xdata$Draw)
```

Extracting Elo-ratings takes advantage of the presence data by either omitting absent IDs from the output or returning them as `NA`. The differences in ratings stem from incorporating undecided interactions.

```r
extract_elo(res2, extractdate = "2000-05-28")
```

```
##    c    d    f    a    k    g    n    w    z
## 1340 1211 1136 1092  962  960  873  860  566
```

```r
# note that "s" is absent and omitted
extract_elo(res2, extractdate = "2000-05-28", IDs = c("s", "a", "c", "k"))
```

```
##    c    a    k    s
## 1340 1092  962   NA
```

```r
# note that "s" is absent and returned as NA
```

Likewise, `eloplot()` omits absent IDs from the resulting plots (figure 4 and figure 5).

```r
eloplot(res2)
```

```r
eloplot(res2, ids = c("s", "a", "w", "k", "c"), from = "2000-06-05", to = "2000-07-04")
```

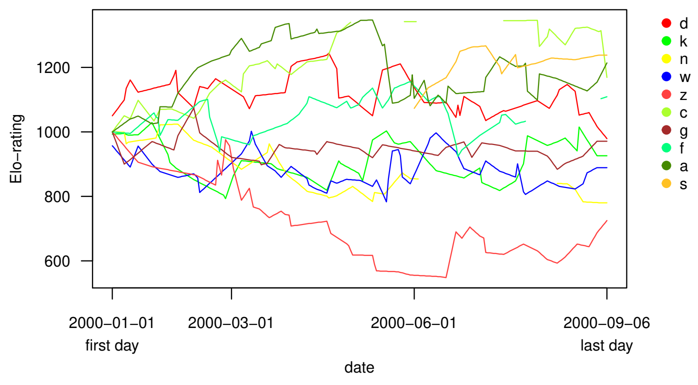

Figure 4: Elo-ratings of 10 individuals over the entire study period. Note that several individuals were absent during parts of the date range and therefore appear with gaps in the plot (e.g. *c* and *f*). Compare to figure 2.

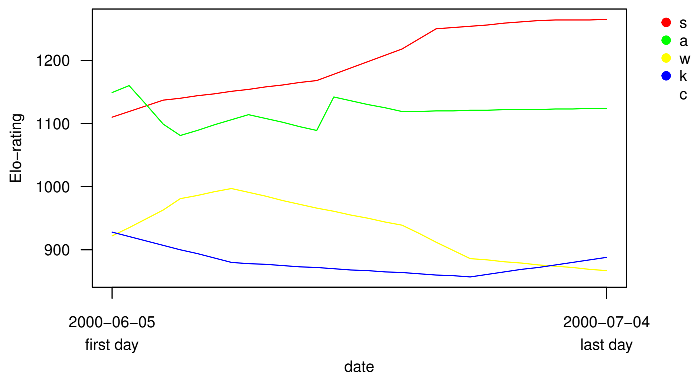

Figure 5: Elo-ratings of 5 individuals over a month. Note that individual *c* is not displayed in the plot, since it has not been present during the date range supplied to `eloplot()`. Compare to figure 3.

Customizing Elo-rating with prior knowledge and optimization

In principle, there are two different tuning knobs with which the calculation of Elo-ratings is controlled. First there is the k parameter and second there is the issue of which start values to assign the individuals at the beginning of the rating process. So far, we simply assumed a single k parameter as well as using the same starting value for each individual throughout. Now, this is probably not ideal because we make at least two implicit assumptions here. First, using the same k implies that all interactions have equal consequences in the sense that we can't distinguish for example between mild and severe aggression. A physical fight is probably more relevant for the assessment of an individual's status than say a mild threat and leave interaction or a displacement without any overt physical aggression. This should/could be reflected in different k values that assign larger k to fights and lower k to mild interactions.[10] Likewise, by assigning each individual the same starting value we assume that, at least potentially, all individuals could have the same status. For example, assigning females and males the same initial score seems not very plausible in many animal species and chances are that starting values that assign higher values to males than to females leads to a more realistic ratings/rankings.

There are two ways to address this. We can either incorporate such reasoning based on prior knowledge of the study system, or alternatively, we can try to find settings that optimize some objective fit criterion. In this section we will go through both approaches. We start by incorporating prior knowledge to set different start and k values. And the then we will look into ways of using maximum likelihood to set these parameters.

Prior knowledge

Starting values

In this section, we incorporate prior knowledge of individual status (Newton-Fisher 2017). This prior knowledge may come in several forms. You may have information on prior ordinal ranks of your individuals or you may know rank classes of individuals. Theoretically, you could also know prior Elo-ratings, but in that case it seems likely that you could actually also obtain the actual interaction data from which these ratings are derived. In the latter case it seems more convenient to actually include these data in your interaction sequence and simply proceed from there, rather than incorporate ratings from a prior analysis as starting point.

The essential idea is that you calculate custom starting values (on the scale of Elo-ratings) in a first step, and then supply this information to the `elo.seq()` function. Your prior information can be either ordinal ranks, or rank classes. We start by using ordinal ranks. Here you need to create a numeric vector with names, where the numbers reflect the ranks and the names the individual IDs. For example:

```r
myranks <- c(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)
# vector needs to have names!
names(myranks) <- c("a", "c", "d", "f", "g", "k", "n", "s", "w", "z")
myranks
```

```
##  a  c  d  f  g  k  n  s  w  z
##  1  2  3  4  5  6  7  8  9 10
```

The `createstartvalues()` function then takes these ranks and translates them into Elo-ratings. The crucial point here is the `shape=` parameter that determines how differentiated these ranks are in terms of magnitude of differences between individuals (figure 6).

Now we can supply these start values to the `elo.seq()` function and then plot the results.

```r
startvals <- createstartvalues(ranks = myranks, shape = 0.3)
res3 <- elo.seq(winner = xdata$winner, loser = xdata$loser, Date = xdata$Date, presence = xpres, startvalue = startvals$res)
```

```r
eloplot(res3)
```

In the next example, we first calculate Elo-ratings without prior knowledge, i.e. like in the examples above. But then we use the final ratings of this step to calculate ranks and based on these ranks, we calculate Elo-ratings again but now with this 'prior' knowledge. Now, this is a somewhat circular approach, but it serves well to show how prior information can flatten Elo-rating trajectories (figure 8). For this example, we use a smaller data set to have less cluttered figures.

```r
data(adv2)
# no prior knowledge
res1 <- elo.seq(winner = adv2$winner, loser = adv2$loser, Date = adv2$Date)
extract_elo(res1)
```

[10] To use a sports metaphor: a tennis match in a Grand Slam tournament should be more impactful on the ranking than a training match (even if it's against the same opponent).

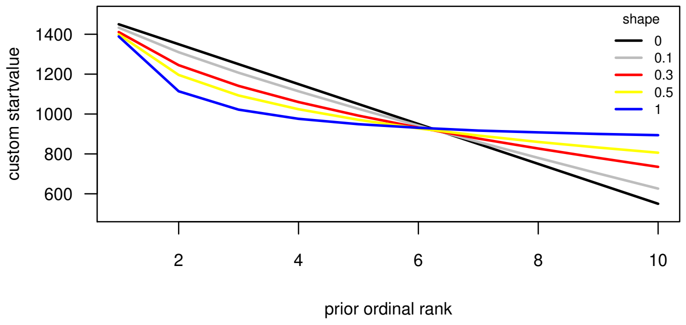

Figure 6: Different shape parameters supplied to the `createstartvalues()` function. Code to produce the figure is in the Appendix.

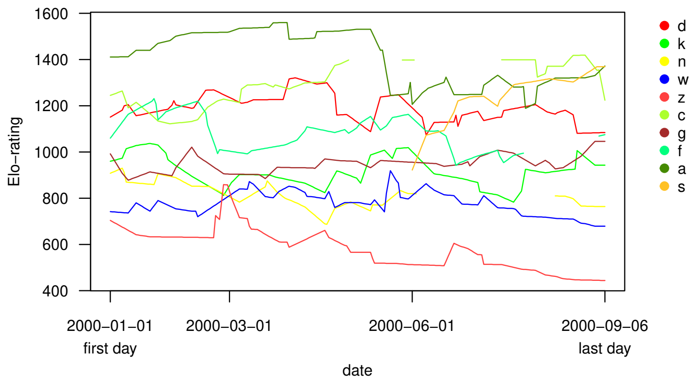

Figure 7: Elo-ratings of 10 individuals over the entire study period with prior knowledge in the form of known ranks incorporated. Compare to figure 4.

```
##    b    c    d    f    e    g    a
## 1203 1148 1116 1004  982  843  704
```

```r
# use the above calculated ratings as known 'ranks'
myranks <- 1:length(extract_elo(res1))
names(myranks) <- names(extract_elo(res1))
mystart <- createstartvalues(myranks, startvalue = 1000, k = 100)
res2 <- elo.seq(winner = adv2$winner, loser = adv2$loser, Date = adv2$Date, startvalue = mystart$res)
extract_elo(res2)
```

```
##    b    c    d    f    e    g    a
## 1321 1233 1111  956  939  795  645
```

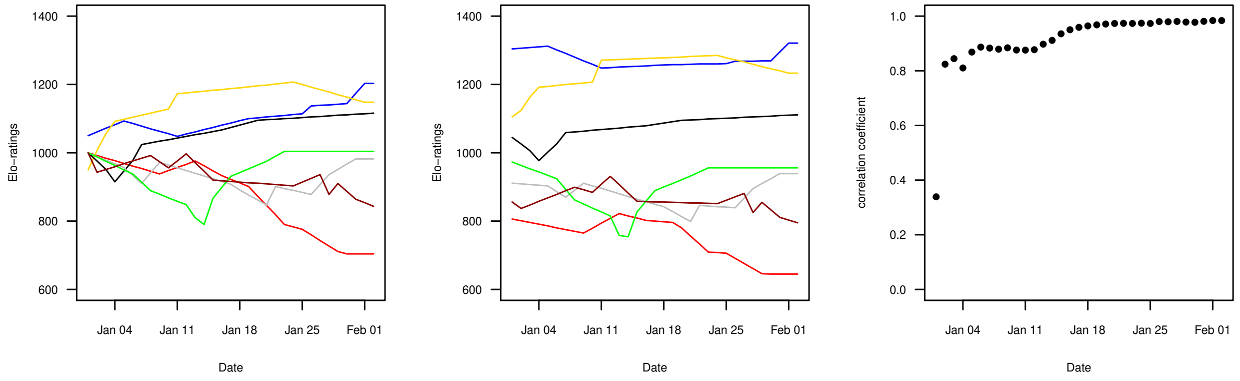

Figure 8: Elo-ratings from a group of seven individuals. On the left without prior knowledge and in the center with prior knowledge (here: the ordinal ranking at the end of the sequence without prior knowledge). The right panel shows the correlation between ratings on each date between the two runs. Code to produce the figure is in the Appendix.

When we look at figure 8 we can see several things. First, the actual order of individuals at the end of both runs is identical and final ratings are highly correlated ($r = 0.98$). Further, several patterns among the final ratings are very similar, e.g. gold and blue are close to each other, as are green and grey[11], while the two red individuals are relatively further spaced apart. The only major difference is the black individual which is close to blue and gold in the left panel but further away from them in the centre panel. Equally interesting are the correlation coefficients for each day between the two approaches (figure 8, right panel). Not surprisingly, correlations become close over time (keep in mind that on the first day, there was only one interaction observed and as such, in the left panel, only two individuals (blue and gold) had ratings different from 1000).

You can also supply rank classes. These rank classes will then be transformed into intermediate 'ranks', which then are transformed into custom start values according to the same principal as above with regards to the shape parameter. The major caveat here is that currently you have to specify **four** rank classes. If you have less, two for example, you still need to specify all four, but leave unused classes `NULL`. The class-rank conversion is done via the following formula: class=1: 1; class=2: $N/4$; class=3: $N/2$; class=4: $N - N/4$, where N is the group size. Please note that rank classes have to be given in descending order, i.e. the first class is the highest class (e.g. 'high-ranking'), but the actual labels of the rank classes are meaningless and are simply used for illustration.

```
# with four rank classes
myrankclasses <- list(alpha = "a", high = c("b", "c"), mid = c("d", "e"), low = c("f", "g"))
createstartvalues(rankclasses = myrankclasses)$res
```

```
##    a    b    c    d    e    f    g
## 1202 1100 1100  952  952  846  846
```

```
# with two rank classes
myrankclasses2 <- list(class1 = NULL, high = c("a", "b", "c"), class3 = NULL,
                       low = c("d", "e", "f", "g"))
createstartvalues(rankclasses = myrankclasses2)$res
```

```
##    a    b    c    d    e    f    g
## 1169 1169 1169  873  873  873  873
```

```
# with two rank classes
myrankclasses3 <- list(high = c("a", "b", "c"), mid = NULL,
                       low = c("d", "e", "f", "g"), superlow = NULL)
createstartvalues(rankclasses = myrankclasses3)$res
```

```
##    a    b    c    d    e    f    g
## 1143 1143 1143  893  893  893  893
```

These start values can be saved as above and then be used in `elo.seq()`. Please note that currently all individuals that are part of the sequence have to have an entry in the prior ranks, otherwise `the elo.seq()` function will fail. For example, the following will result in an error, because only two out of ten individuals have prior ratings:

```
mypriors <- c(2000, 0); names(mypriors) <- c("a", "g")
elo.seq(winner = adv2$winner, loser = adv2$loser, Date = adv2$Date,
                  startvalue = mypriors)
```

In future package versions it hopefully will be allowed to include subsets of individuals in the context of prior ratings (such that the above example would work).

In general, I (CN) am still unsure about the general validity of this approach. It adds information without much apparent justification: from an ordinal ranking (or rank classes) with meaningless differences between the ranks (or rank classes) to cardinal scores where the differences between any two pairs of individuals can be compared in terms of magnitude. For example, in an ordinal ranking no distinction can be made between the difference of rank 1 versus rank 2 or rank 11 versus rank 12. In a cardinal system (like David's score), such differences are meaningful, such that rank 1 and rank 2 can be relatively more close to each other than rank 11 and 12, or the other way around. That being said, if there is some knowledge of the species in this respect, i.e. if we know how pronounced power differentials are between individuals (compare also to steepness), the approach may be justified. Otherwise, for the moment I would advise either to set the shape parameter to 0 (which implies that differences between all adjacent pairs of individuals are identical, steepness = 1) or to refrain from incorporating ordinal ranks altogether. A better founded reasoning for the approach is needed in my opinion, and also it would be really interesting to achieve the inclusion of cardinal scores into the system, for example prior knowledge of David's scores.

[11] my apologies to colour blind people

Adjusting *k*

Here we describe how to set different k values to different types of interactions. Typically, setting k would allow differentiating interactions of different intensity. Similarly, we can envision that k could be set according to the value of the resource that is contested in a given interaction. Larger ks would be set for interactions of higher intensity or higher resource value.

In our small example of 33 interactions between 7 individuals, we have observed interactions of two different intensities: displace (threat/leave) interactions and fights (fight/flee interactions). For the sake of this example, we consider that displacing another individual is milder and should have less of an effect on dominance status, hence we assign such interactions a relatively low k value, here $k = 50$. In contrast, 'real' fights should be much more consequential, hence we chose $k = 200$. In the example here I chose the two k values arbitrarily. For an objective way for how to choose varying k (or even if you have only a single k, i.e. you don't distinguish between different aggression types), look at the section on optimization.

In the current implementation, different k have to be supplied as a named list, where the names have to correspond to the intensity/interaction type specified. In our example we need a list with two items:

```
table(adv2$intensity) # remind ourselves how the intensity categories are named
```

```
##
## displace    fight
##       26        7
```

```
myk <- list(displace = 50, fight = 200)
res3 <- elo.seq(winner = adv2$winner, loser = adv2$loser, Date = adv2$Date, intensity = adv2$intensity, k = myk)
extract_elo(res3)
```

```
##    b    f    d    c    e    g    a
## 1149 1130 1126 1089  989  799  718
```

Finally, we compare the final ratings from the different approaches (figure 9).

```
plot(0, 0, "n", xlim = c(1, 7), ylim = c(600, 1400), xlab = "individual", ylab = "Elo-rating",
     axes = FALSE)
axis(1, at = 1:7, labels = res1$allids, lwd = NA); axis(2, las = 1)

x <- extract_elo(res1); x <- x[sort(names(x))]
points(1:7, x, pch = 16, col = "red")

x <- extract_elo(res2); x <- x[sort(names(x))]
points(1:7, x, pch = 16, col = "blue")

x <- extract_elo(res3); x <- x[sort(names(x))]
points(1:7, x, pch = 16, col = "gold")
box()
legend(4, 1400, legend = c("no prior knowledge", "prior ranks", "custom k"), lwd = 1,
       col = c("red", "blue", "gold"), ncol = 3, xpd = TRUE, xjust = 0.5, yjust = 0,
       cex = 0.8, bg = "white")
```

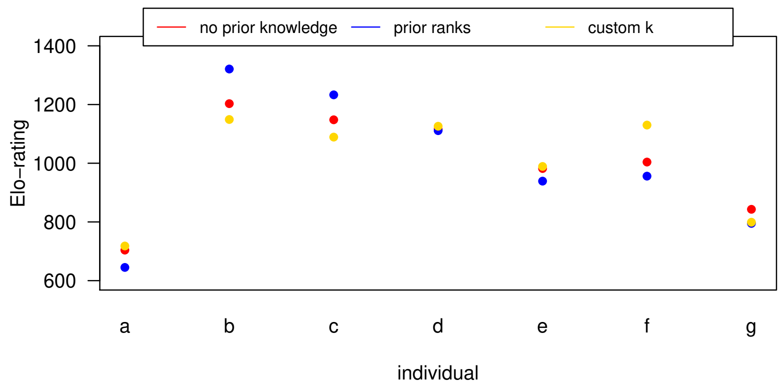

Figure 9: Final ratings of 7 individuals when calculated without any prior knowledge (red), prior knowledge of 'ranks' (blue) and accounting for different interaction intensities (gold).

And this is just a fun exercise, in which we repeat the same as above, i.e. calculate ratings (1) without prior information, (2) with prior information but also (3) with a completely wrong prior rank order (we reverse the order). We use the bigger data set here. The code for the figure is in the appendix.

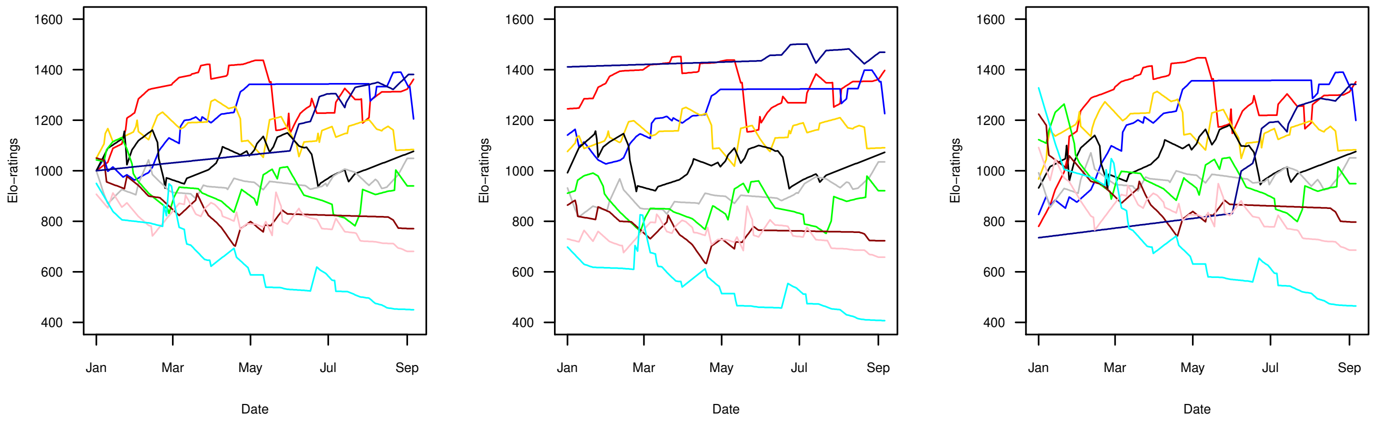

Figure 10: Elo-ratings from a group of ten individuals. On the left without prior knowledge and in the center with prior knowledge (here: the ordinal ranking at the end of the sequence without prior knowledge). The right panel shows the ratings as they develop if we use some obviously wrong prior ranking. Code to produce the figure is in the Appendix.

Optimization

Optimizing *k*

Another approach of adjusting k is to optimize its value with a likelihood approach (see the very cool articles by Franz et al. (2015) and Foerster et al. (2016)). The fundamental idea is that we can optimize an objective criterion, i.e. find the k that leads to the best possible value for that criterion. The criterion we can look at is the maximum likelihood of winning probabilities. For this, we set a specific k value, calculate the ratings from our observed interactions and then extract the expected winning probabilities for the eventual winner in each interaction (see also figure 18). In the best possible case, all the winning probabilities are 1,[12] i.e. the expected winning probabilities are optimal with respect to the eventual outcomes of all interactions, or in other words, the expected winning probabilities match perfectly the eventual outcomes. In the worst case scenario, all the expected winning probabilities are 0, i.e. we would always predict the wrong winner when looking at the winning probabilities. The objective criterion we can look at is the product of these expected winning probabilities in a sequence, i.e. the likelihood. In the first example where all probabilities were 1, the product would be 1 too. In the second case, with all probabilities being 0, the likelihood would be 0. In practice, it is often useful to use logged probabilities, i.e. the log likelihood and all the functions in the package by default return log likelihoods rather than likelihoods.

To recap: the expected winning probability for an individual depends on the rating difference prior to an interaction and k determines how much a rating can change after a single interaction. Hence, using different k will lead to different winning probabilities. Just to be clear, we calculate the rating sequence multiple times and just change k for each iteration. So, each iteration of the sequence and using different k will lead to different sets of expected winning probabilities and hence to different (log) likelihoods. The k value that results in the largest (= maximal) log likelihood is the k that leads to the largest expected winning probabilities. The function that achieves this in the package is `opimizek()`.

`opimizek()` does exactly what is described above: calculate ratings multiple times with different k (`resolution=` determines how often) and extract the product of the expected winning probabilities. The other thing to specify is the range of k you want to test (the `krange=` argument), and here limits of 10 until 500 seems a reasonable choice. Finally, note that the `optimizek()` function requires as its main argument the result of a call to `elo.seq()`.

So, here we calculate the ratings from a sequence with 250 interactions and just use the default $k = \mathit{100}$ in this step (for the purpose of finding the optimal k, its value in this step is of no relevance). Then we apply `optimizek()` with k limits of 10 and 500 and with a resolution of 491.[13] The result of this is a list with two entries: `$best` contains the optimal k for that sequence, and `$complete` contains the entire set of tested k along with the log likelihood for each tested k.

[12]Winning probabilities will never be exactly 1, but realistically they might come *very* close to 1. Also, they will never be exactly 0, but could very well be *very* close to 0.

[13]I chose this odd resolution to obtain steps 1 of the tested k: check out `seq(from = 10, to = 500, length.out = 491)`.

```
eres <- elo.seq(xdata$winner, xdata$loser, xdata$Date)
ores <- optimizek(eres, krange = c(10, 500), resolution = 491)
ores$best
```

```
##     k    loglik
## 84 93 -111.4457
```

So the k that leads to the largest likelihood is 93 (figure 11). You could also aim for more precision, i.e. increase the `resolution=` argument: with an increment of 0.25 (i.e. `resolution = 1961`) we find the best k at 93.25, with a marginally larger log likelihood. We can display the results of this procedure with the following code:

```
plot(ores$complete$k, ores$complete$loglik, type = "l", las = 1, xlab = bquote(italic(k)), ylab = "log likelihood")
abline(v = ores$best$k, col = "red")
```

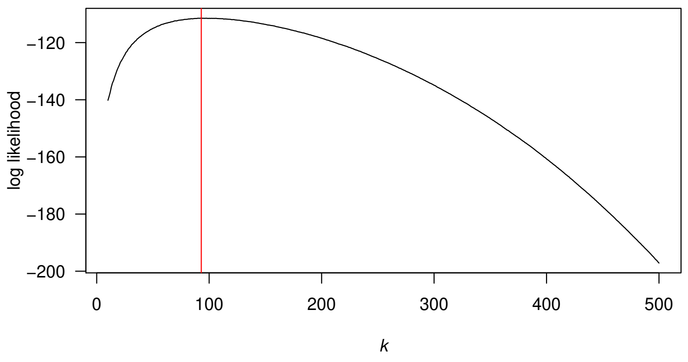

Figure 11: Optimal k for an interaction sequence with 250 interactions among 10 individuals. The k that maximizes the likelihood is 93.

Please note that this procedure can result in non-optimal results, i.e. there might be no local maximum along the range of k values tested, i.e. the maximum lies at one of the edges of the k range. For example, if we test only the range from, say, 200 to 400, we find that the maximum is at the edge of the range, i.e. 200 (figure 12). In a more realistic scenario, you might find that the optimal k is 0 (assuming you start your range at 0). Whether this is sensible needs to be decided on a case to case basis, I believe (see also the discussion in Foerster et al. (2016)).

```
ores1 <- optimizek(eres, krange = c(200, 400), resolution = 501)
ores1$best
```

```
##     k    loglik
## 1 200 -118.4614
```

```
plot(ores1$complete$k, ores1$complete$loglik, type = "l", las = 1, xlab = bquote(italic(k)), ylab = "log likelihood")
abline(v = ores1$best$k, col = "red")
```

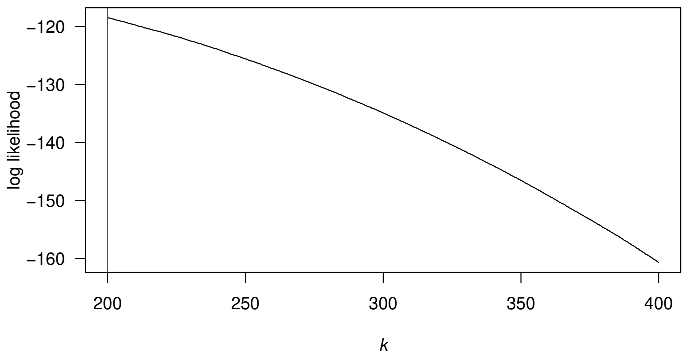

Figure 12: Optimization procedure that did not result in a local maximum. The maximum likelihood estimate for k here is 200, which lies at the boundary of the tested values.

More generally speaking, the shape of the likelihood function can vary substantially between data sets. Figure 13 shows a few examples of data sets that come with this package.

The next step in this approach is to incorporate any knowledge you may have about interaction types. As stated above, it seems plausible to assume that interactions of different intensities will have different consequences on individuals'

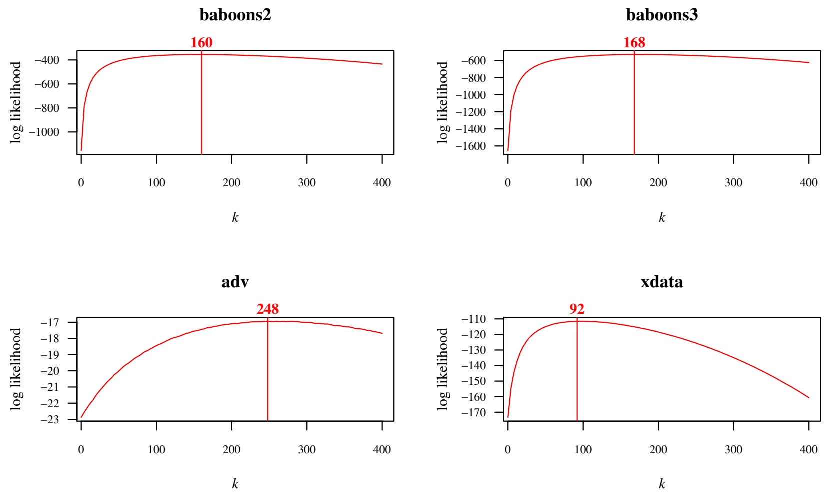

Figure 13: Likelihood functions for six example data sets. The panel titles reflect the names of the data sets as stored in the package.

rating trajectories, which may be reflected in different k values for different interaction types. But the same question remains as above, i.e. what values to choose? One solution is to simply extend the optimization approach to multiple dimensions (for example finding two k values if there are two interaction types). Implementing this is also done with the `optimizek()` function. There are two things that need to be changed compared to the case with only finding one k: we need to supply a list with the to-be-tested k values with one list item for each interaction type and we need to specify the vector/column that contains the information about which interaction type belongs to each interaction.[14]

Two more things to note. (1) The *names* of the list need to match the entries in the interaction type vector/column. (2) The resolution setting works combinatorially, i.e. if you set this to 5 with two interaction types the function will run $5 * 5 = 25$ times, and if you have three interaction types with a resolution of 100 the function will iterate $100 * 100 * 100 = 1000000$ times. Just keep in mind that higher resolutions here lead to longer run times, so it might be wise to start with small values to see whether everything works.

```
eres <- elo.seq(xdata$winner, xdata$loser, xdata$Date, intensity = xdata$intensity)
# two list items: 'fight' and 'threat', because these are the two interaction types specified in xdata$intensity
mykranges <- list(fight = c(10, 500), threat = c(10, 500))
ores2 <- optimizek(eres, krange = mykranges, resolution = 91)
ores2$best
```

```
##         fight   threat    loglik
## 1294 113.4444 86.22222 -111.1532
```

So what we find here is that the optimal setting of two different k is as follows: $k = 113.4$ for fights, and $k = 86.2$ for threats. Incidentally, this matches our general expectation that more intense interactions (fights) should lead to larger changes.[15]

We can also illustrate these results with a heatmap (figure 14):

```
heatmapplot(loglik ~ threat + fight, data = ores2$complete)
```

Finally, we can also investigate how taking into account interaction type compares to ignoring interaction type (figure 15)[16]. The idea here is that a rating model that accounts for some variable (interaction type) should be better than a model that doesn't account for it. So here we visualize the likelihoods as a function of different k. The thick blue line represents the results with one k, ignoring interaction types (the same as in figure 11). For the red lines we look at how the likelihood changes according to k_{threat} when the k for fights (k_{fight}) is fixed. The continuous line reflects this at $k_{fight} = 113.4$, i.e. at the optimal value for k_{fight}, while the dashed line corresponds to $k_{fight} = 10$. The grey curves represent the same thing, just that k_{fight} varies, while k_{threat} is fixed (continuous line: $k_{threat} = 86.2$ and dashed line $k_{threat} = 500$). The arrows below the horizontal axis depict the maximum point of each curve. There are at least two points to make here. First, the k values at the peaks of the continuous curves (indicated by the arrows) reflect the k values we found in the optimization step above. This is logical because I chose the fixed values to depict in the figure

[14] Handing over the interaction type vector can be done either when running `elo.seq()` or it can be done in the `optimizek()` function call.
[15] Of course, I made up these interactions with the goal of showing what one would expect, so this is hardly surprising.
[16] Code for this figure is in the appendix

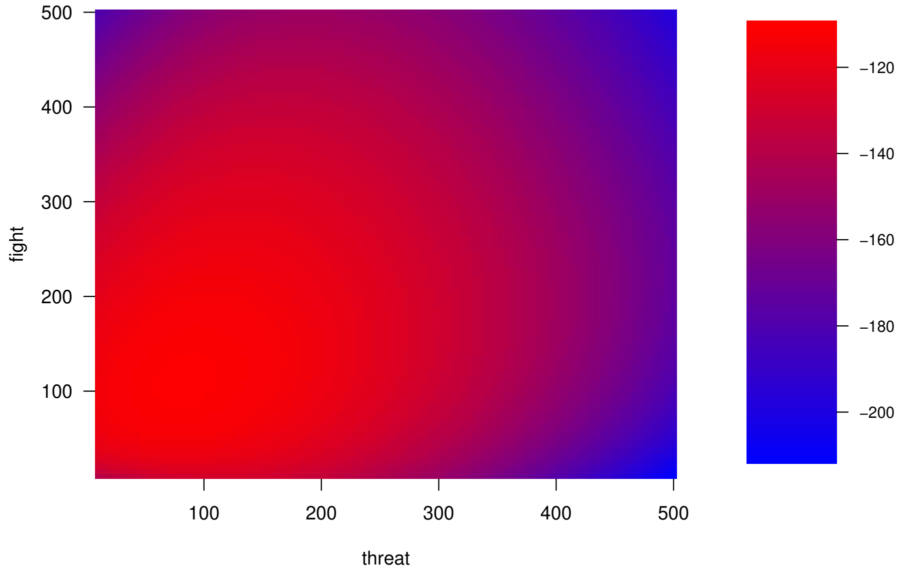

Figure 14: Optimal k values for an interaction sequence with two different interaction types (fights and threats). Large likelihoods are red, small likelihoods are blue.

post-hoc, i.e. after I knew what the optimal values were. Second, in absolute terms, neither the continuous red and grey curves lead to a maximum likelihood that is substantially larger than that at the maximum of the blue curve (-111.15 versus -111.45). Hence, in this example it appears that the added complexity of incorporating two interaction types doesn't lead to a substantially better fit (in terms of winning probabilities on which the ML is based). You could even formally test this with a likelihood ratio test, although I don't see a practical reason to do so in real life.

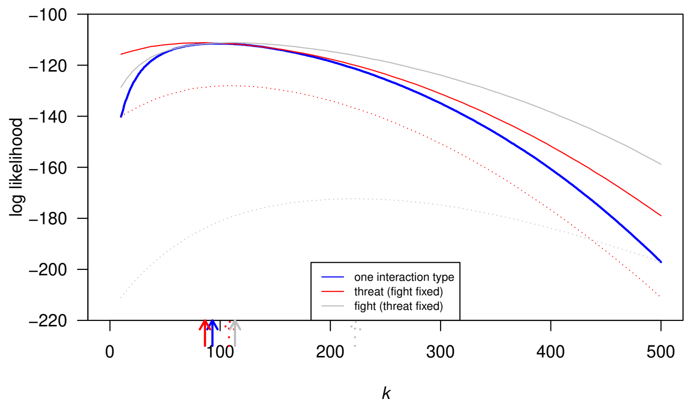

Figure 15: Incorporating versus ignoring interaction type leads to different shapes in likelihood functions. The arrows below the horizontal axis indicate the optimal k for each curve. See text for more explanations.

Optimizing start values

It is also possible to use a maximum likelihood approach to try out multiple sets of starting values in order to find one the maximizes the likelihood of winning probabilities. Actually though, this is more of an approximate approach because we cannot go through all possible combinations of start ratings. So what we rather do is to generate a large number of sets of starting values and then assess the likelihood of these sets. For example a set of *a = 1100, b = 1000, c = 900* may be better than the set of *a = 1200, b = 1000, c = 800*. As it is implemented, this is a very inefficient method, simply because it tries randomly selected sets of start values. As such, using the term *optimizing* is probably a bit of a stretch here!

```
orires <- elo.seq(winner = adv$winner,
                  loser = adv$loser,
                  Date = adv$Date,
                  runcheck = FALSE)
xres <- optistart(eloobject = orires,
                  runs = 5000)

# using 'good' ('optimal') start values
xres1 <- elo.seq(winner = adv$winner,
                 loser = adv$loser,
                 Date = adv$Date,
                 runcheck = FALSE,
                 startvalue = xres$resmat[c(which.max(xres$logliks)), ])
# using 'bad' start values
xres2 <- elo.seq(winner = adv$winner,
                 loser = adv$loser,
                 Date = adv$Date,
                 runcheck = FALSE,
                 startvalue = xres$resmat[c(which.min(xres$logliks)), ])
```

```
eloplot(xres1)
eloplot(xres2)
```

```
eloplot(xres1)
```

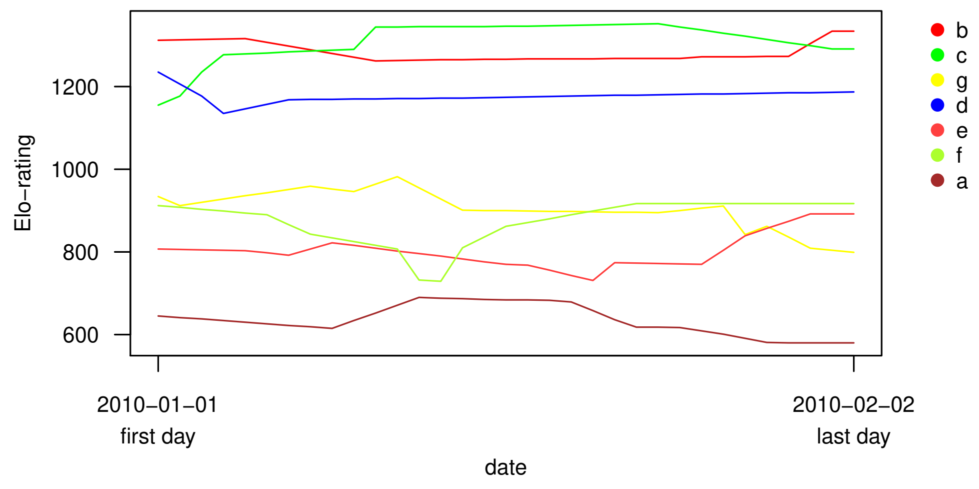

Figure 16: Ratings with starting values that were created using the `optistart()` function.

In the following piece of code we run a simulation to compare the performance of informed start ratings (either via prior knowledge or via optimization) with uninformed start ratings. We begin with creating an interaction sequence for individuals for which we *know* the true ranks (via a function in the `aniDom` package). Then we calculate ratings in three different ways. (1) Use start values based on known ranks (`ores1`). (2) Use start values based on a best set of random start values (as outlined in the paragraph above, `ores2`). (3) Use default constant start values (`ores4`). In the code there is a fourth option where the number of sets for randomly assigned start values is larger, but I commented this out to speed up the simulation. The expectation is that the higher the number of possible start values we try the higher the probability that a 'good' set is among them. In the end, we compare the correlations between the final ratings from each of the three approaches and the true ranks. A higher correlation indicates that the relationship between true ranks and final ratings is better.

```
library(aniDom)
set.seed(123)
resmat <- matrix(ncol = 4, nrow = 50)

for (i in 1:nrow(resmat)) {
  # create interactions from known ranks
  xd <- generate_interactions(N.inds = 10, N.obs = 200, b = -2, a = 1, id.biased = TRUE)
  allids <- letters[1:10]
  w <- allids[xd$interactions$Winner]
  l <- allids[xd$interactions$Loser]
  D <- seq.Date(as.Date("2000-01-01"), by = "day", length.out = length(w))

  # informed by known ranks
  myranks <- 1:10
  names(myranks) <- allids
  kvals <- rep(100, length(w))
  svals <- createstartvalues(ranks = myranks, shape = 0.5)$res
  ores1 <- fastelo(w, l, allids, kvals, svals)
  ores1 <- ores1[[1]][allids]

  # informed by optimized start values
  templist <- list(allids = allids,
                   misc = c(normprob = "1"),
                   logtable = data.frame(winner = w, loser = l),
                   kvals = rep(100, length(w)),
                   startvalues = rep(1000, length(allids)))
  svals <- optistart(templist, runs = 200)$best
  ores2 <- fastelo(w, l, allids, kvals, svals)
  ores2 <- ores2[[1]][allids]

  # with more runs
  # svals <- optistart(templist, runs = 2000)$best
  # ores3 <- fastelo(w, l, allids, kvals, svals)
  # ores3 <- ores3[[1]][allids]

  # uninformed
  svals <- rep(1000, length(allids))
  ores4 <- fastelo(w, l, allids, kvals, svals)
  ores4 <- ores4[[1]]

  # store results
  resmat[i, 1] <- cor(ores1, myranks, method = "s")
  resmat[i, 2] <- cor(ores2, myranks, method = "s")
  # resmat[i, 3] <- cor(ores3, myranks, method = "s")
  resmat[i, 4] <- cor(ores4, myranks, method = "s")

  # clean up
  rm(xd, allids, w, l, D, myranks, kvals, svals, ores1, ores2, ores4, templist)
}

# 'inverse', so that correlations are positive
resmat <- resmat * (-1)
boxplot(resmat, axes = FALSE, boxwex = 0.4, lty = 1)
axis(1, at = 1:4, tcl = 0, cex.axis = 0.7, padj = 0.5,
     labels = c("informed by\nknown ranks",
                "informed by\noptimized start values\n(200 runs)",
                "informed by\noptimized start values\n(2000 runs)",
                "uninformed"))
axis(2, las = 1)
title(ylab = "Spearman correlation with true ranks")
box()
```

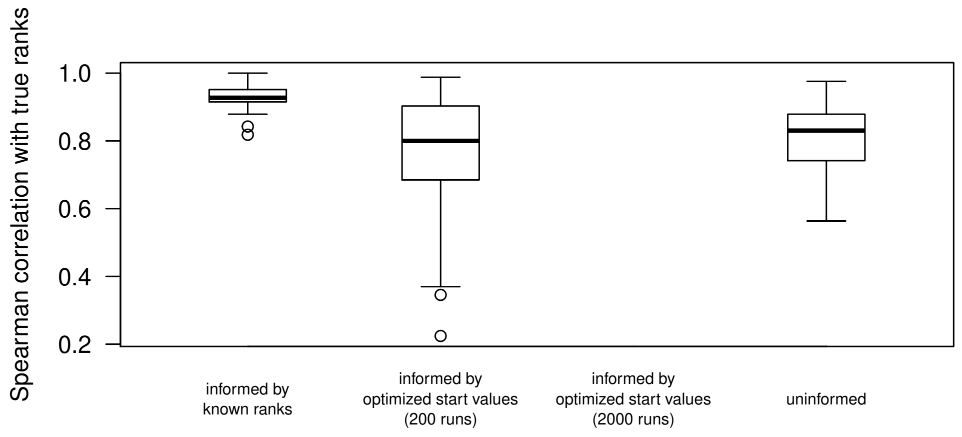

Figure 17: Comparison of performance between different approaches to assign custom start values. Note that the third boxplot is omitted for performance reasons. If you want to see it, you need to uncomment the relevant lines in the code block.

Figure 17 shows the results of this simulation. We created 50 different data sets. The best performance is achieved by using approach (1), i.e. assigning ratings based on prior knowledge of ranks. The remaining two approaches don't differ much in terms of their performance. Although, if you increase the number of trial sets in `optistart()` to a higher value, performance improves a bit. The take-home message here then is that if you know prior ranks, use them.[17] That said, if you don't know prior ranks then there seems to be only a very small advantage of using procedure (2). One more thing to note here is that this simulation is far from comprehensive. For example, group size was fixed (10) as was the number of interactions per data set (200). I suspect that for larger group sizes approach (2) becomes (even more) inefficient.

More functions related to Elo-rating

In addition to calculate, extract and display/plot Elo-ratings, our package also provides some more functions that may be useful in some contexts.

Hierarchy stability with `stab.elo()`

`stab.elo()` can be used to calculate an index of hierarchy stability (S, see Neumann et al. 2011) and McDonald and Shizuka (2013)). Please note that in contrast to the original publication, S now is limited to a range between 0 and 1, where 1 indicates a stable hierarchy in which no rank changes occurred (see this section for more information).

```
res2 <- elo.seq(winner = xdata$winner, loser = xdata$loser, Date = xdata$Date, presence = xpres,
               draw = xdata$Draw)
stab_elo(res2, from = "2000-05-05", to = "2000-06-05")
```

```
## [1] 0.9658
```

Individual rating trajectories with `traj.elo()`

`traj.elo()` provides information about Elo-rating trajectories of individuals over time. These trajectories are simply regression slopes of ratings as a function of days. To calculate these slopes we use ratings per day, which may differ from those ratings that would be returned by `extract_elo()`. For `extract_elo()`, we return ratings on a given day regardless of whether there were actually observed interactions for a given individual (i.e. the most recent rating). For `traj_elo()`, we use only ratings from days *with* observed interactions. As a consequence, we need at least two days with interactions to calculate a slope. If there was no or only one day with observed interactions, the slope is returned as `NA` in the output and a message is produced (s and n in the examples).

```
traj_elo(res2, ID = c("s", "f", "n", "z"), from = "2000-05-05", to = "2000-06-05")
```

```
## no (or only one) observation for s during specified date range

##   ID   fromDate     toDate     slope Nobs
## 1  s 2000-05-05 2000-06-05        NA    1
## 2  f 2000-05-05 2000-06-05  1.696998    6
## 3  n 2000-05-05 2000-06-05  3.904463    5
## 4  z 2000-05-05 2000-06-05 -2.172414    5
```

```
traj_elo(res2, ID = c("s", "f", "n", "z"), from = "2000-06-05", to = "2000-07-05")
```

```
## no (or only one) observation for n during specified date range

##   ID   fromDate     toDate     slope Nobs
## 1  s 2000-06-05 2000-07-05  5.505214    6
## 2  f 2000-06-05 2000-07-05 -7.128125    8
## 3  n 2000-06-05 2000-07-05        NA    0
## 4  z 2000-06-05 2000-07-05  6.862589    7
```

`individuals()`

`individuals()` provides information about which/how many individuals were present on specific dates. When applied over a date *range*, the average number of individuals can be returned as can the coefficient of variation of the number of individuals present on each date. Note that this function has little relevance if the calculation of Elo-ratings (see above) is *not* supplemented by presence data.

```
individuals(res2, from = "2000-05-05", to = "2000-05-05", outp = "N")
```

```
## [1] 8
```

[17] But if you actually have interaction data from *before* your study, I would rather incorporate those interactions in my sequence directly rather than discarding them in favour of using ordinal ranks

```r
individuals(res2, from = "2000-05-05", to = "2000-06-05", outp = "N")
```

```
## [1] 8.3125
```

```r
individuals(res2, from = "2000-05-05", to = "2000-06-05", outp = "CV")
```

```
## [1] 0.07125283
```

```r
individuals(res2, from = "2000-05-05", to = "2000-06-05", outp = "IDs")
```

```
##  [1] "d" "k" "n" "w" "z" "c" "g" "f" "a" "s"
```

`winprob()`

`winprob()` simply returns the expected probability of an individual winning given its own rating and that of its opponent. Note that there are two major ways of calculating these probabilities. Without going into details too much: the package default assumes a normal distribution by default (following Albers and de Vries (2001), see Elo (1978) for more details), the alternative being based on an exponential distribution (Elo 1978).

```r
winprob(1000, 1200)
```

```
## [1] 0.2397501
```

```r
winprob(1000, 1200, normprob = FALSE)
```

```
## [1] 0.2402531
```

```r
winprob(1200, 1000)
```

```
## [1] 0.7602499
```

```r
winprob(1200, 1000, normprob = FALSE)
```

```
## [1] 0.7597469
```

```r
winprob(1200, 1200)
```

```
## [1] 0.5
```

```r
winprob(1200, 1200, normprob = FALSE)
```

```
## [1] 0.5
```

Elo (1978) discussed both, but it turns out both approaches produce very similar results (figure 18), specifically when rating differences are relatively small (up to about 200 points difference).

There is at least one more way to calculate winning probabilities, which was used by Foerster et al. (2016) and Sánchez-Tójar, Schroeder, and Farine (2018) and is implemented in the `EloOptimized` package (Feldblum, Foerster, and Franz 2019) and in the `aniDom` package (Farine and Sánchez-Tójar 2019).

Furthermore, we can simulate random interaction sequences, then calculate ratings based on both winning probabilities, and then extract the ratings from a randomly selected individual. If we do this 100 times, we can see that there is very little difference between the two approaches (figure 19).

From sequence to matrix with `creatematrix()`

`creatematrix()` returns a square matrix which can be used with other, matrix-based algorithms to calculate dominance scores or ranks (e.g. I&SI de Vries 1998) or David's score (David 1987; Gammell et al. 2003; de Vries, Stevens, and Vervaecke 2006), see sections on David's scores and I&SI).

The function works either from the results of `elo.seq()` or from vectors of winners and losers. First, we look at creating matrices from the results from `elo.seq`. If undecided interactions (ties/draws) are present in the data, you can decide on how to treat them (either 0.5 or 1 for both individuals, or they are omitted (default)). Individuals that were absent during the specified date range are excluded from the matrix by default. In addition, the matrix can be restricted to individuals that had interactions (i.e. *observed* interactions) in the date range.

```r
creatematrix(res2)
```

```
##   a c d f g k n s  w  z
## a 0 5 5 2 9 4 2 1 10  6
## c 0 0 4 7 3 4 1 1  5  2
## d 2 0 0 2 5 5 4 0  8 10
## f 0 2 0 0 2 6 4 0  6  5
```

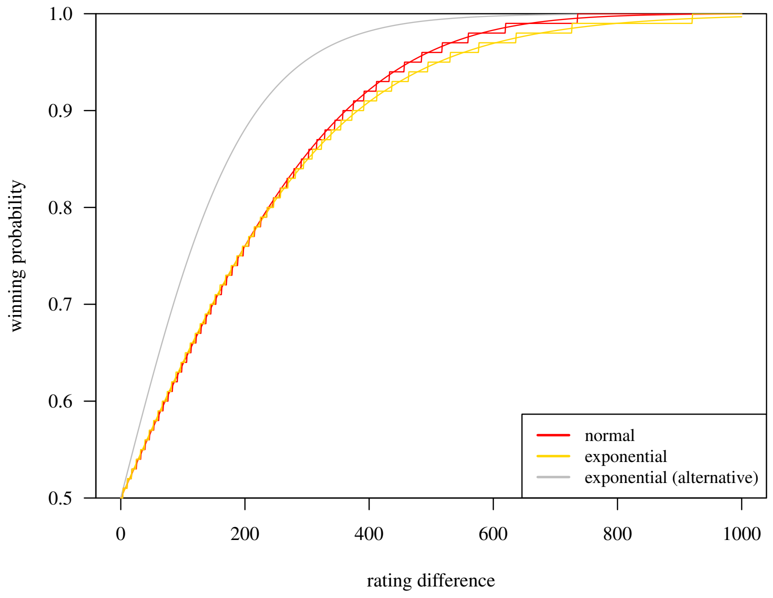

Figure 18: Three ways of calculating winning probabilities. Up to a rating difference of 200 points, the two curves proposed by Elo 1978 (red and gold) are virtually indistinguishable. The step-wise curves are taken from Elo 1978 and Albers and de Vries 2001, which provide tables in intervals, for example, the winning probability is 0.5 if the rating difference is between 0 and 3. The curve for the algorithm used by Feldblum and colleagues and Farine and colleagues, in contrast, is much steeper (grey line). Code to produce the figure is in the Appendix.

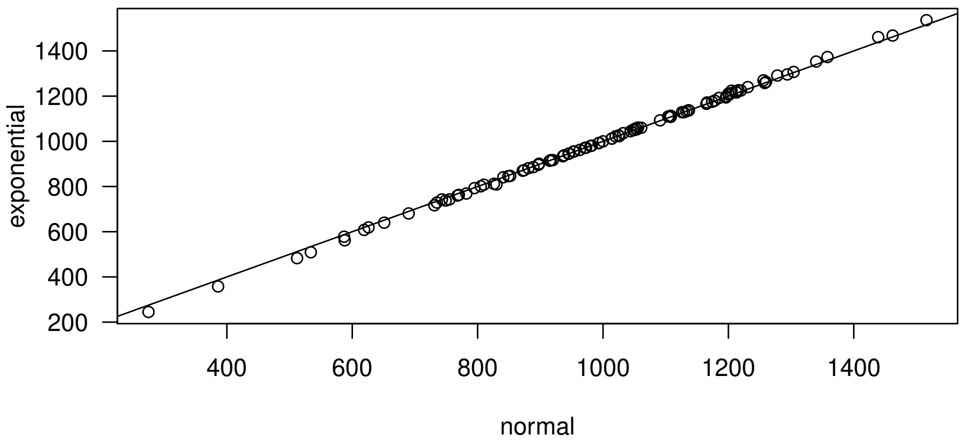

Figure 19: Ratings from individuals that were based on either normally distributed winning probabilities or exponentially distributed winning probabilities. Code for the simulation and figure is in the Appendix.

```
## g 0 0 0 0 0 4 3 0  6  2
## k 1 0 3 0 0 0 2 0  2  6
## n 0 0 0 0 2 0 0 0  2  3
## s 3 0 2 1 3 0 0 0  2  2
## w 2 0 0 0 0 1 1 0  0 11
## z 0 0 0 2 1 1 0 0  0  0
```

```
sum(creatematrix(res2))
```

```
## [1] 200
```

```
creatematrix(res2, drawmethod = "0.5")
```

```
##     a   c   d   f    g   k   n   s    w    z
## a 0.0 6.0 6.0 2.5 10.0 4.0 2.0 1.0 13.0  6.5
## c 1.0 0.0 4.0 7.5  3.0 4.0 1.0 1.5  6.0  2.5
## d 3.0 0.0 0.0 3.5  5.0 5.5 4.0 0.5  8.0 12.0
## f 0.5 2.5 1.5 0.0  2.5 6.5 5.0 0.5  6.5  5.0
## g 1.0 0.0 0.0 0.5  0.0 4.0 3.0 0.0  8.0  2.5
## k 1.0 0.0 3.5 0.5  0.0 0.0 2.5 0.0  3.0  7.0
## n 0.0 0.0 0.0 1.0  2.0 0.5 0.0 0.0  2.5  4.0
## s 3.0 0.5 2.5 1.5  3.0 0.0 0.0 0.0  2.5  2.0
## w 5.0 1.0 0.0 0.5  2.0 2.0 1.5 0.5  0.0 12.0
## z 0.5 0.5 2.0 2.0  1.5 2.0 1.0 0.0  1.0  0.0
```

```
sum(creatematrix(res2, drawmethod = "0.5"))
```

```
## [1] 250
```

```
# "c" and "n" are omitted
creatematrix(res2, daterange = c("2000-06-10", "2000-06-16"))
```

```
##   a d f g k s w z
## a 0 0 0 1 0 0 0 0
## d 0 0 0 0 0 0 0 0
## f 0 0 0 0 1 0 0 1
## g 0 0 0 0 0 0 0 0
## k 0 0 0 0 0 0 0 0
## s 0 0 0 0 0 0 0 0
## w 0 0 0 0 0 0 0 0
## z 0 0 0 0 0 0 0 0
```

```
creatematrix(res2, daterange = c("2000-06-10", "2000-06-16"), onlyinteracting = TRUE)
```

```
##   a f g k z
## a 0 0 1 0 0
## f 0 0 0 1 1
## g 0 0 0 0 0
## k 0 0 0 0 0
## z 0 0 0 0 0
```

If you want to create a matrix directly from winner and loser vectors, you need to take care that both the `winners=` and the `losers=` arguments are named. So the following will work:

```
creatematrix(winners = xdata$winner, losers = xdata$loser)
```

```
##   a c d f  g k n s  w  z
## a 0 7 7 3 11 4 2 1 16  7
## c 0 0 4 7  3 4 1 1  6  3
## d 2 0 0 5  5 6 4 0  8 14
## f 0 3 0 0  3 7 6 0  7  5
## g 0 0 0 0  0 4 3 0 10  3
## k 1 0 3 0  0 0 3 0  4  8
## n 0 0 0 0  2 0 0 0  3  4
## s 3 1 3 2  3 0 0 0  3  2
## w 2 1 0 0  0 1 1 0  0 13
## z 0 0 0 2  1 1 1 0  0  0
```

But this one will not work:

```
creatematrix(xdata$winner, xdata$loser)
```

Also note that if you work with vectors directly, there is no way to take into account date ranges and hence co-residency. If you want to make sure that your matrices adhere to presence information, you need to run `elo.seq()` first and include the relevant presence information. Then create the matrix from its results.

`randomsequence()`

`randomsequence()` creates random data sets, which can be used for simulations for example. It returns a list with two `data.frames` (named `seqdat` and `pres` for the actual sequence and presence data, respectively). By default, it creates a sequence of 100 interactions between 10 individuals. All IDs are present the entire time and there are no undecided interactions. Also by default, IDs are simply single letters and in order to produce realistic data, IDs that appear earlier in alphabetic order are more likely to win any given interaction (`alphabet = TRUE`). The proportion of reversals (against that order) is by default set to `reversals = 0.1`.

```
rdata <- randomsequence()
xres <- elo.seq(winner = rdata$seqdat$winner, loser = rdata$seqdat$loser, Date = rdata$seqdat$Date,
                presence = rdata$pres)
summary(xres)
```

```
## Elo ratings from 10 individuals
## total (mean/median) number of interactions: 100 (20/19)
## range of interactions: 14 - 30
## date range: 2000-01-01 - 2000-04-09
## startvalue: 1000
## uppon arrival treatment: average
## k: 100
## proportion of draws in the data set: 0
```

From dominance matrix to sequence with `randomelo()`

This is an experimental function to generate a set of random sequences based on an interaction matrix. Based on the randomly generated sequences, Elo-ratings are calculated (in the example 5 times, `runs = 5`)[18]. For inspection, we save the average Elo-ratings across all the randomized sequences in a new object called `res`. If you plan to use this function, you may want to have a look at the `EloChoice` package, which does the same, but is substantially faster.

```
data(bonobos)
xdata <- randomelo(bonobos, runs = 5)
res <- data.frame(ID = colnames(xdata[[1]]), avg = round(colMeans(xdata[[1]]), 1))
```

Now, compare that to David's scores (figure 20).[19]

```
ds <- DS(bonobos)
ds <- ds[order(ds$ID), ]
plot(ds$normDS, res$avg, xlab = "David's score", ylab = "randomized average Elo-rating", las = 1,
     xlim=c(0, 6))
```

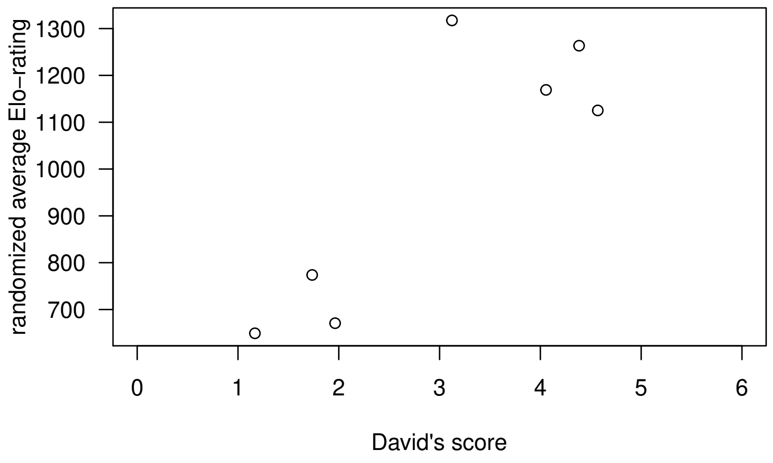

Figure 20: David's scores and average randomized Elo-ratings from seven bonobos (data from de Vries et al 2006).

[18]Typically, you would want to set a substantially higher number, for example 1000
[19]Note, the plot you will get will differ because the generation of Elo-ratings is based on *random* sequences

Utilities

Proportion of unknown relationships with `prunk()`

This function lets you determine how large the proportion of dyads in your data set is for which no interactions have been observed. You can use this function on both the results of `elo.seq()` or an interaction matrix. If used on an `eloobject`, you will see as a result the unknown relationships for all dyads that were found in the date range, and additionally restricted to those dyads that were actually co-resident at some point during the date range. In the example, this results in the identical output since all dyads were co-resident at some point. Of course, the accuracy of the second part of the output depends on presence data being supplied. Note for matrices, we cannot control for co-residency, so the second part of the output is returned as `NA` if `prunk()` is used with a matrix.

```
data(adv); data(advpres)
x <- elo.seq(winner = adv$winner, loser = adv$loser, Date = adv$Date, presence = advpres)
prunk(x, daterange = c("2010-01-01", "2010-01-15"))
```

```
##      pu.all   dyads.all    pu.cores dyads.cores
##       0.524      21.000       0.524      21.000
```

```
mat <- creatematrix(x, daterange = c("2010-01-01", "2010-01-15"))
prunk(mat)
```

```
##      pu.all   dyads.all    pu.cores dyads.cores
##       0.524      21.000          NA          NA
```

David’s scores and steepness

With the `DS()` function you can calculate David’s scores (David 1987; Gammell et al. 2003; de Vries, Stevens, and Vervaecke 2006). Note that this function only works on square matrices (see `creatematrix()` for how to create a matrix from a sequence). There are two ways by which the dyadic winning proportions can be calculated. The default way is to calculate proportions corrected for chance (`prop = "Dij"`). Alternatively, you can use raw proportions with `prop = "Pij"` (see de Vries, Stevens, and Vervaecke (2006) for details).

```
data(bonobos)
DS(bonobos, prop = "Dij")
```

```
##   ID         DS   normDS
## 1 He  10.987484 4.569641
## 2 Dz   9.689126 4.384161
## 3 Ho   7.392677 4.056097
## 4 De   0.860578 3.122940
## 5 Ko  -7.255948 1.963436
## 6 Re  -8.849421 1.735797
## 7 Ki -12.824495 1.167929
```

With the `steepness()` function we can calculate steepness based on David’s scores (de Vries, Stevens, and Vervaecke 2006).

```
steepness(bonobos, nrand = 1000)
```

```
##        steep     expected            p        nrand
##    0.6283758    0.2652995    0.0010000 1000.0000000
```

However, you may want to be somewhat careful about its interpretation because steepness is known to depend on matrix sparseness (proportion of unknown relationships, Klass and Cords (2011), figure 21):

```
plot(0, 0, "n", xlab = "sparseness", ylab = "steepness", las = 1, xlim = c(0, 1), ylim = c(0, 1))
for(i in 1:100) {
  x <- randomsequence(nID = 15, avgIA = 40)
  xmat <- creatematrix(winners = x$seqdat$winner, losers = x$seqdat$loser)
  # remove a random number of cells (replace by 0)
  xmat[sample(1:225, sample(0:200, 1))] <- 0
  # calculate and plot sparseness and steepness
  points(prunk(xmat)[1], steepness(xmat)[1])
}
```

Directional consistency with `DCindex()`

You can also calculate the Directional Consistency Index (cf. van Hooff and Wensing 1987).

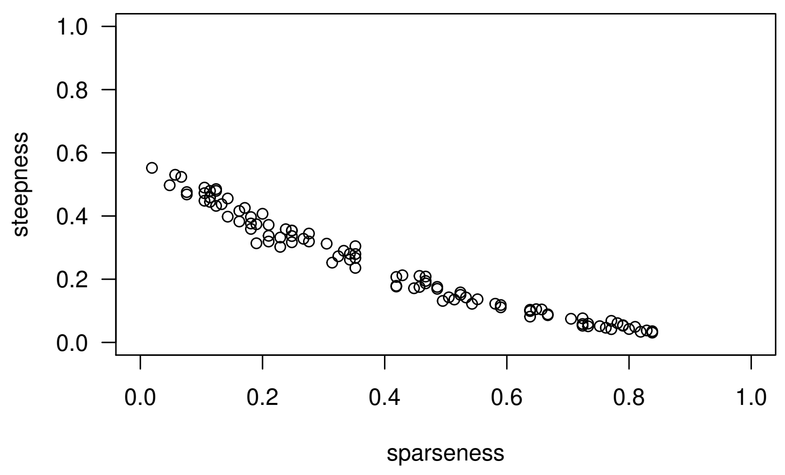

Figure 21: Relationship between sparseness and steepness (data from 100 random matrices).

```
DCindex(devries98)
```

```
## [1] 0.9056604
```

```
DCindex(bonobos)
```

```
## [1] 0.9473684
```

Linearity, transitivity and ordinal ranks

Linearity with `h.index()`

The function `h.index()` allows calculating the linearity indices h and h' (Appleby 1983; de Vries 1995)[20]. The significance test as described by de Vries (1995) is also included. In order to get it to work, you need to extract a matrix from your eloobject (see `creatematrix`).

```
mat <- creatematrix(winners = adv$winner, losers = adv$loser)
h.index(mat, loops = 1000)
```

```
##          variable     value
## 1               N    7.0000
## 2         h index    0.4821
## 3        h' index    0.5536
## 4      expected h    0.3721
## 5         p right    0.2270
## 6 randomizations 1000.0000
## 7            tied    1.0000
## 8         unknown    4.0000
```

Or you can use a matrix directly, as with the matrix of seven bonobos that is included in this package (data from de Vries, Stevens, and Vervaecke (2006)).

```
data(bonobos)
h.index(bonobos, loops = 1000)
```

```
##          variable     value
## 1               N    7.0000
## 2         h index    0.8036
## 3        h' index    0.8571
## 4      expected h    0.3725
## 5         p right    0.0260
## 6 randomizations 1000.0000
## 7            tied    0.0000
```

[20] As noted above for steepness, also h and h' are affected by the number unknown relationships (Shizuka and McDonald 2015, @neumann2018a).

```
## 8        unknown    3.0000
```

Triangle transitivity (`transitivity()`)

Shizuka and McDonald (2012) suggested an alternative measure to quantify the degree of linearity in dominance networks.

```
data(devries98)
set.seed(123)
transitivity(devries98, runs = 1000)
```

```
##       Pt     ttri        p     runs
##    0.907    0.630    0.011 1000.000
```

```
data(adv)
mat <- creatematrix(winners = adv$winner, losers = adv$loser)
set.seed(123)
transitivity(mat, runs = 1000)
```

```
##       Pt     ttri        p     runs
##    0.923    0.692    0.095 1000.000
```

Linear hierarchy with the I&SI algorithm (`ISI()`)

It is also possible to order the individuals in a matrix according to a linear hierarchy (Appleby 1983; de Vries 1995, 1998). This is implemented in the function `ISI()` that re-orders the matrix according to the I&SI algorithm suggested by de Vries (1998)). Strictly speaking, applying this algorithm is only justified if there is linearity in the matrix in the first place. This can be tested with `h.index()` function (see above). For illustration, I also present an example for a non-linear hierarchy. Note that the application of the I&SI method does *not* necessarily result in a unique solution. This is likely related to the assumption that actually the matrix has to be linear. In other words, we can apply the I&SI algorithm to any matrix, but whether this reflects a linear ordering depends on whether there actually is such a linear order in the first place. Let's start with de Vries' (1998) example, which can be ordered linearly according to de Vries' randomization test (de Vries 1995).

```
data(devries98)
set.seed(123)
h.index(devries98)
```

```
##         variable     value
## 1              N   10.0000
## 2        h index    0.5818
## 3       h' index    0.6424
## 4     expected h    0.2743
## 5        p right    0.0130
## 6 randomizations 1000.0000
## 7           tied    1.0000
## 8        unknown   10.0000
```

```
ISI(devries98)
```

```
## I = 2
## SI = 7
```

```
## [[1]]
##   a b v g w h k e c y
## a 0 4 5 3 0 6 2 2 3 1
## b 0 0 0 1 1 0 2 1 2 2
## v 0 0 0 2 1 0 0 2 7 7
## g 0 0 0 0 2 1 0 4 3 0
## w 2 0 0 0 0 3 0 0 2 1
## h 0 0 3 0 0 0 0 6 2 5
## k 0 0 0 0 0 0 0 0 2 1
## e 0 0 0 0 0 0 0 0 0 4
## c 0 0 0 0 1 0 2 0 0 6
## y 0 0 0 0 0 0 0 0 2 0
```

Now let's look at an example for which the linearity assumption is not met. Incidentally, these data come from the Elo-rating example given by Albers and de Vries (2001).[21] We first bring the sequence into matrix form, then test

[21] Remember, this is exactly the point of Elo-rating, i.e. it does not assume a linear order across the entire time period, but rather handles temporal changes and assigns individual scores, not ranks.

for linearity and run the `ISI()` function. Note that here we get three possible orderings that equally well fit a linear hierarchy (though that linear ranking is actually not justified). All three possibilities contain one inconsistency with a strength of 2.

```r
data(adv)
mat <- creatematrix(winners = adv$winner, losers = adv$loser)
h.index(mat)
```

```
##         variable     value
## 1              N    7.0000
## 2        h index    0.4821
## 3       h' index    0.5536
## 4     expected h    0.3751
## 5        p right    0.2550
## 6 randomizations 1000.0000
## 7           tied    1.0000
## 8        unknown    4.0000
```

```r
set.seed(123)
res <- ISI(mat)
```

```
## more than 1 solution
```

```
## I = 1
## SI = 2
```

With `ISIranks()` we can have the actual ranks returned in a more readable format. The actual sorting (either by ID or average rank) can be controlled via `sortbyID=`. The results here indicate that four individuals had the same rank assigned in each of the three rankings (a, b, c and d). Three others held different ranks in each of the three possible solutions (e, f and g).

```r
ISIranks(res, sortbyID = TRUE)
```

```
##   ID avg rnkg1 rnkg2 rnkg3
## 1  a   7     7     7     7
## 2  b   1     1     1     1
## 3  c   2     2     2     2
## 4  d   3     3     3     3
## 5  e   5     6     4     5
## 6  f   5     5     6     4
## 7  g   5     4     5     6
```

Custom plots of Elo-ratings

This section gives a few hints on how to plot Elo-ratings in case your not satisfied with the results of the standard `eloplot()` function. The main thing to know is that the ratings for each individual are stored in the results of `elo.seq()`, specifically in the list item `cmat`.[22] The actual object is a matrix, and contains a column for each individual and a row for each date. For example:

```r
data(adv); data(advpres)
SEQ <- elo.seq(winner = adv$winner, loser = adv$loser, Date = adv$Date, presence = advpres)
ratings <- SEQ$cmat
head(ratings)
```

```
##         b    c    g    d    e    a   f
## [1,] 1050  950 1000 1000 1000 1000  NA
## [2,] 1061 1007  943  976  989  992  NA
## [3,] 1072 1056  951  951  978  984 938
## [4,] 1082 1092  959  915  968  977 938
## [5,] 1093 1098  968  946  957  969  NA
## [6,] 1086 1104  976  977  934  961 938
```

Note that this approach is illustrated in case you use the default Elo-ratings, i.e. when `init="average"` is set in the `elo.seq()` step (which it is by default). I have not tested it, but this plotting approach should in principle also work for the two other modes of `init=`.

The only thing left that we need for creating our plot are the actual dates, which are not part of the `\$cmat` matrix. The dates can be found in the item `truedates`, which again is part of the output of `elo.seq()` (which we stored in the object `SEQ`).

[22] list items can be accessed with the `$` character

```
dates <- SEQ$truedates
head(dates)
```

```
## [1] "2010-01-01" "2010-01-02" "2010-01-03" "2010-01-04" "2010-01-05" "2010-01-06"
```

So now we can do the plot. We start by setting up an empty plot,[23] which we subsequently fill with our data, specifically looping through each individual (i.e. columns in `ratings`). Then we add the axes and draw a box around the plot.

```
plot(0, 0, xlim = range(dates), ylim = range(ratings, na.rm = T), axes = FALSE, xlab = "Date",
     ylab = "Elo-ratings")
for(i in 1:ncol(ratings)) points(dates, ratings[, i], type = "l")
axis.Date(1, x = dates, cex.axis = 0.8)
axis(2, las = 1, cex.axis = 0.8)
box()
```

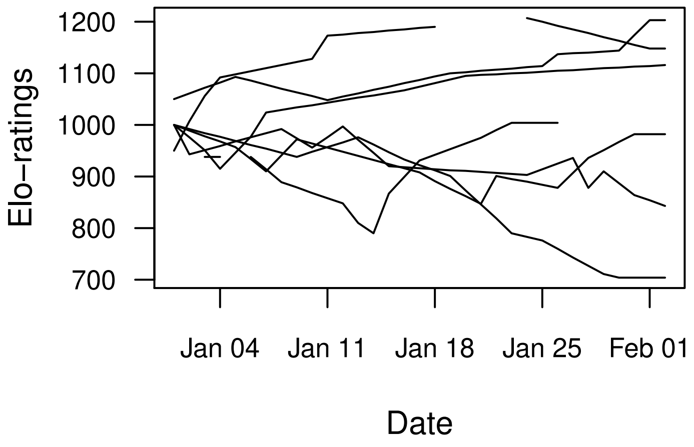

Figure 22: Elo-ratings of 7 individuals across one month.

Now, we can modify the colors for individuals. Note that I will also change the line types, just so the logic becomes clear. The important thing here is that you need as many colors/line types as you have individuals (here seven) and you need to specify the colors in the order in which individual occur in `ratings`.

```
plot(0, 0, xlim = range(dates), ylim = range(ratings, na.rm = T), axes = FALSE, xlab = "Date",
     ylab = "Elo-ratings")
mycols <- c("red", "green", "blue", "gold", "black", "grey", "darkred")
myltys <- c(1, 2, 3, 1, 2, 3, 1)
for(i in 1:ncol(ratings)) points(dates, ratings[, i], type = "l", col = mycols[i], lty = myltys[i])
axis.Date(1, x = dates, cex.axis = 0.8)
axis(2, las = 1, cex.axis = 0.8)
box()
```

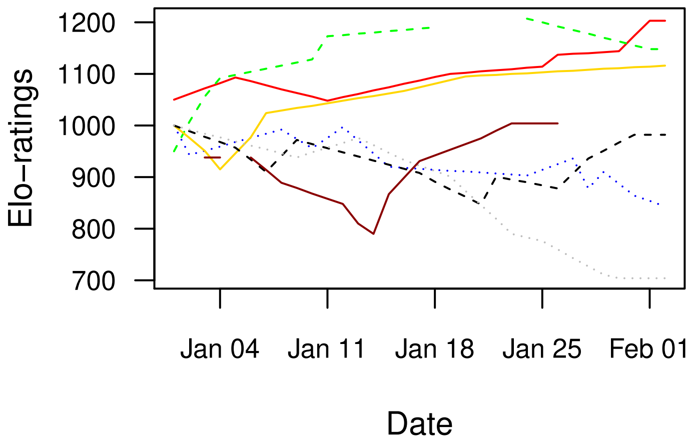

Figure 23: Elo-ratings of 7 individuals across one month. Individuals are coded by color and line type.

[23]The main reason for this seemingly complicated way is that we want custom axes. There might be a more straightforward way of doing this, but I am not aware of it.

If we want to have a legend it gets a bit more tricky. For this, we need to set up a plot layout, which basically creates two plotting areas, the first (left) of which we use for the plot, and the other (right) for the legend. Setting up the exact location for the legend is the actual tricky part, and requires a little trial and error in my experience because the way your final figure looks like depends how your plotting system is set up. Specifically, you may want to experiment with the `x` and `y` values in the `legend(x=<...>, y=<...>)` call.

```
layout(matrix(c(1, 2), ncol = 2), heights = c(5, 5), widths = c(4, 1))
plot(0, 0, xlim = range(dates), ylim = range(ratings, na.rm = T), axes = FALSE, xlab = "Date",
     ylab = "Elo-ratings")
mycols <- c("red", "green", "blue", "gold", "black", "grey", "darkred")
myltys <- c(1, 2, 3, 1, 2, 3, 1)
for(i in 1:ncol(ratings)) points(dates, ratings[, i], type = "l", col = mycols[i], lty = myltys[i])
axis.Date(1, x = dates, cex.axis = 0.8)
axis(2, las = 1, cex.axis = 0.8)
box()
# set margins for legend plot
par(mar = c(5, 0.5, 3.8, 0.5))
plot(1:2, 1:2, xaxt = "n", yaxt = "n", type = "n", bty = "n", ylab = "", xlab = "")
legend(x = 1, y = 2.04, colnames(ratings), cex = 0.8, bty = "n", pch = 16, pt.cex = 1, col = mycols,
       lty = myltys)
```

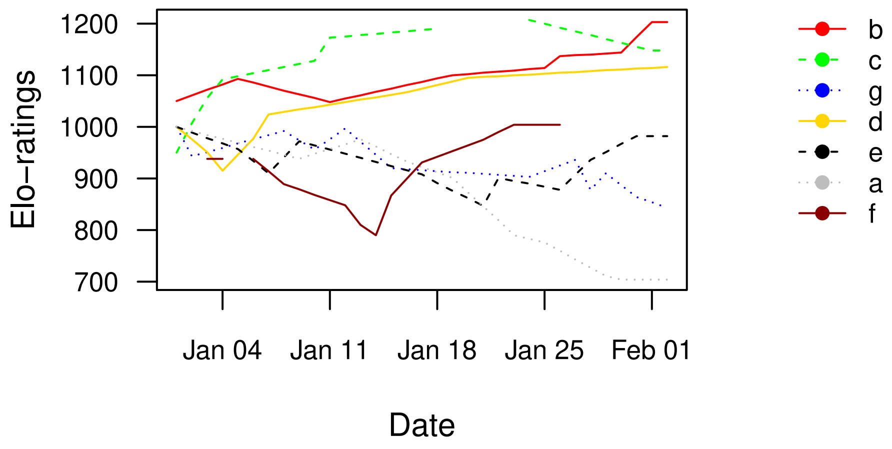

Figure 24: Elo-ratings of 7 individuals across one month. Individuals are coded by color and line type, which are noted in the legend.

Finally, we may want to add symbols on top of the lines so that distinction between individuals becomes clearer in addition or replacing the colour approach. Because putting symbols on for each single interaction may clutter the plot, I find it useful to use only every Xth data point. In the example we use every 3rd point (`stp <- 3`). The data come from yet another list item in the results (`$nmat`).

```
ias <- apply(SEQ$nmat, 2, cumsum)
stp <- 3
layout(matrix(c(1, 2), ncol = 2), heights = c(5, 5), widths = c(4, 1))
plot(0, 0, xlim = range(dates), ylim = range(ratings, na.rm = T), axes = FALSE, xlab = "Date", ylab = "Elo-ratings")
mycols <- c("red", "green", "blue", "gold", "black", "grey", "darkred")
myltys <- c(1, 2, 3, 1, 2, 3, 1)
mysymbs <- c(15:21)
for(i in 1:ncol(ratings)) {
  points(dates, ratings[, i], type = "l", col = mycols[i], lty = myltys[i])
  pos <- sapply(unique(ias[, i] %/% stp),
                function(X)min(which(ias[, i] %/% stp == X))
                )[-1]
  points(dates[pos], ratings[pos, i], pch = mysymbs[i], col = mycols[i])
}
axis.Date(1, x = dates, cex.axis = 0.8)
axis(2, las = 1, cex.axis = 0.8)
box()
par(mar = c(5, 0.5, 3.8, 0.5))
plot(1:2, 1:2, xaxt = "n", yaxt = "n", type = "n", bty = "n", ylab = "", xlab = "")
legend(1, 2.04, colnames(ratings), cex = 0.8, bty="n", pch = mysymbs, pt.cex = 1, col = mycols, lty = myltys)
```

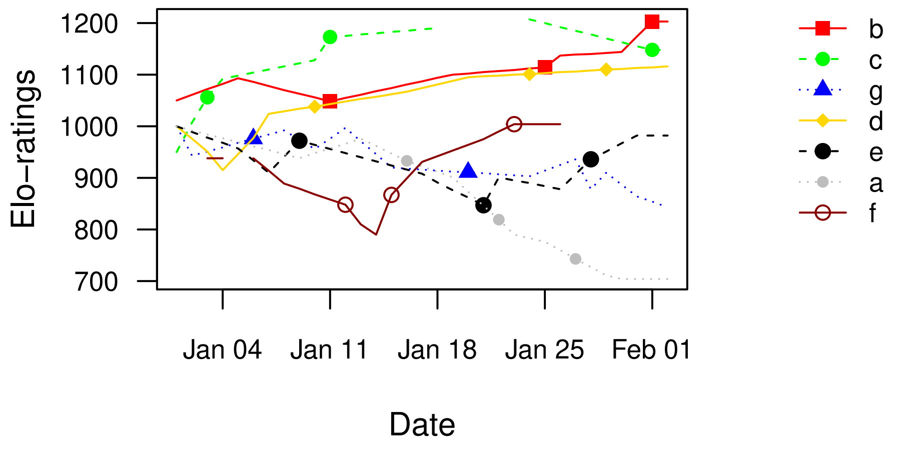

Figure 25: Elo-ratings of 7 individuals across one month. Individuals are coded by color and line type, which are noted in the legend. For each individual, ratings after every third interaction are highlighted.

Further notes on the stability index

Originally, Neumann et al. (2011) defined the stability index S as:

$$S = \frac{\sum_{i=1}^{d}(C_i \times w_i)}{\sum_{i=1}^{d}(N_i)} \tag{1}$$

with:

C_i: the sum of absolute differences between rankings of two consecutive days

w_i: a weighting factor determined as the standardized Elo-rating of the highest-ranked individual involved in a rank change

N_i: the number of individuals present on both days

d: the number of days

This approach was (justifiably so) criticized by McDonald and Shizuka (2013), who pointed out that (1) S is coded against intuition, i.e., $S = 0$ indicates a stable hierarchy and larger values unstable situations, and (2) S is not standardized, i.e., its maximal value depends on the number of individuals present.

McDonald and Shizuka (2013) suggested the following modification:

$$S_t = 1 - \frac{S}{2n} \tag{2}$$

where S is the stability index as defined above (equation 1) and n is the group size.

As far as I can see, this approach only applies to situations in which group size is stable, and as such does not do justice to one of the major advantages of Elo-rating, i.e., its ability to work on data sets in which group size changes.

A stability index that includes the suggestions of McDonald and Shizuka (2013) but preserves the ability to deal with varying group sizes can be defined as follows:

$$S = 1 - \frac{\sum_{i=2}^{d}(C_i \times w_i)}{\sum_{i=2}^{d}(M_i)} \tag{3}$$

where C_i and w_i are defined as above and M_i is the sum of absolute rank changes per day if the hierarchy completely reversed.[24] This index now ranges between 0 and 1, where 1 indicates total stability (no changes) and 0 indicates total

[24] In a group of three animals this would amount to 4, if $n = 4$ $M_i = 8$, if $n = 5$ $M_i = 12$, if $n = 6$ $M_i = 18$, etc. In other words, this is maximally possible magnitude of the sum of rank changes. M_i will always be positive and $M_i \geq C_i$.

instability (complete reversal of rank order every other day). Note that M_i depends on group size, but can take into account varying group sizes because it is calculated for each single day. Note also that the index can only be calculated from the second day onward (see the index i), because it is based on changes between two consecutive days (it does not make sense to calculate S for the first day of a study because there is no day before from which rank changes can be assessed). This was incorrectly displayed in the original equation 1, but was correctly implemented in the functions supplied in the supplementary material for the Animal Behaviour paper (Neumann et al. 2011).

A final note. I did not change the symbol for the modified index, i.e., I kept it as 'S' for now. The functions in the current version of the `EloRating`-package (v. 0.43) calculate S following equation 3, i.e., S is standardized between 0 and 1, and 1 indicates a stable hierarchy.

Appendix

This code produces figure 1.

```r
# run model (note that for simplicity this is a GLM and not a mixed model)
mod <- glm(parasites ~ elo, data = parasites, family = "poisson")
# calculate predicted values
pdata <- data.frame(elo = seq(min(parasites$elo), max(parasites$elo), length.out = 51))
pdata$par <- predict(mod, newdata = pdata, type = "r")
# add some colour
xcols <- rainbow(n = 9, alpha = 0.5)[parasites$id]
# plot and draw model on top
plot(parasites$elo, parasites$parasites, pch = 16, col = xcols,
     xlab = "Elo rating", ylab = "parasite count", las = 1)
points(pdata$elo, pdata$par, type = "l")
```

This code produces figure 6.

```r
plot(0, 0, "n", xlim = c(1,10), ylim = c(500, 1500), xlab = "prior ordinal rank",
     ylab = "custom startvalue", cex.axis = 0.8, cex.lab = 0.8, las = 1)
shapes <- c(0, 0.1, 0.3, 0.5, 1)
xcols <- c("black", "grey", "red", "yellow", "blue")
for(i in 1:length(shapes)) {
  points(myranks, createstartvalues(ranks = myranks, shape = shapes[i])$res, type = "l",
         col = xcols[i], lwd = 2)
}
legend("topright", lty = 1, col = xcols, legend = shapes, lwd = 2, title = "shape", bty = "n",
       cex = 0.7)
```

This code produces figure 8.

```r
par(mfrow = c(1, 3))

dates <- res1$truedates
mycols <- c("red", "blue", "gold", "black", "grey", "green", "darkred")


ratings1 <- res1$cmat
ratings1 <- ratings1[, sort(colnames(ratings1))]
plot(0, 0, type = "n", xlim = range(dates), ylim = c(600, 1400), axes = FALSE, xlab = "Date",
     ylab = "Elo-ratings", cex.lab = 0.7)
for(i in 1:ncol(ratings1)) points(dates, ratings1[, i], type = "l", col = mycols[i])
axis.Date(1, x = dates, cex.axis = 0.7)
axis(2, las = 1, cex.axis = 0.7)
box()

ratings2 <- res2$cmat
ratings2 <- ratings2[, sort(colnames(ratings2))]
plot(0, 0, type = "n", xlim = range(dates), ylim = c(600, 1400), axes = FALSE, xlab = "Date",
     ylab = "Elo-ratings", cex.lab = 0.7)
for(i in 1:ncol(ratings2)) points(dates, ratings2[, i], type = "l", col = mycols[i])
axis.Date(1, x = dates, cex.axis = 0.7)
axis(2, las = 1, cex.axis = 0.7)
box()


plot(0, 0, type = "n", xlim = range(dates), ylim = c(0, 1), axes = FALSE, xlab = "Date",
     ylab = "correlation coefficient", cex.lab = 0.7)
for(i in 1:nrow(ratings1)) points(dates[i], cor(ratings1[i, ], ratings2[i, ]), pch = 16)
axis.Date(1, x = dates, cex.axis = 0.7)
axis(2, las = 1, cex.axis = 0.7)
box()
```

This code produces figure 10.

```r
xdata <- read.table(system.file("ex-sequence.txt", package = 'EloRating'), header = TRUE)
# no prior knowledge
s1 <- elo.seq(winner = xdata$winner, loser = xdata$loser, Date = xdata$Date)

# use the above calculated ratings as known 'ranks'
myranks <- 1:length(extract_elo(s1))
names(myranks) <- names(extract_elo(s1))
mystart <- createstartvalues(myranks, startvalue = 1000, k = 100)
s2 <- elo.seq(winner = xdata$winner, loser = xdata$loser, Date = xdata$Date,
              startvalue = mystart$res)

# reverse
myranks[1:10] <- 10:1
mystart <- createstartvalues(myranks, startvalue = 1000, k = 100)
s3 <- elo.seq(winner = xdata$winner, loser = xdata$loser, Date = xdata$Date,
              startvalue = mystart$res)

par(mfrow = c(1, 3))

dates <- s1$truedates
mycols <- c("red", "blue", "gold", "black", "grey", "green", "darkred", "darkblue", "pink", "cyan")

# do the plots
ratings1 <- s1$cmat
ratings1 <- ratings1[, sort(colnames(ratings1))]
plot(0, 0, type = "n", xlim = range(dates), ylim = c(400, 1600), axes = FALSE, xlab = "Date",
     ylab = "Elo-ratings", cex.lab = 0.7)
for(i in 1:ncol(ratings1)) points(dates, ratings1[, i], type = "l", col = mycols[i])
axis.Date(1, x = dates, cex.axis = 0.7)
axis(2, las = 1, cex.axis = 0.7)
box()

ratings2 <- s2$cmat
ratings2 <- ratings2[, sort(colnames(ratings2))]
plot(0, 0, type = "n", xlim = range(dates), ylim = c(400, 1600), axes = FALSE, xlab = "Date",
     ylab = "Elo-ratings", cex.lab = 0.7)
for(i in 1:ncol(ratings2)) points(dates, ratings2[, i], type = "l", col = mycols[i])
axis.Date(1, x = dates, cex.axis = 0.7)
axis(2, las = 1, cex.axis = 0.7)
box()

ratings3 <- s3$cmat
ratings3 <- ratings3[, sort(colnames(ratings3))]
plot(0, 0, type = "n", xlim = range(dates), ylim = c(400, 1600), axes = FALSE, xlab = "Date",
     ylab = "Elo-ratings", cex.lab = 0.7)
for(i in 1:ncol(ratings3)) points(dates, ratings3[, i], type = "l", col = mycols[i])
axis.Date(1, x = dates, cex.axis = 0.7)
axis(2, las = 1, cex.axis = 0.7)
box()
```

This code produces figure 15.

```r
plot(0, 0, type = "n", xlim = c(0, 500), ylim = c(-220, -100), las = 1, xlab = bquote(italic(k)),
     ylab = "log likelihood", yaxs = "i")
points(ores$complete$k, ores$complete$loglik, type = "l", col = "blue", lwd = 2)
arrows(x0 = ores$best$k, y0 = -230, x1 = ores$best$k, y1 = -220, col = "blue", lwd = 2, xpd = TRUE, length = 0.1)

pdata <- ores2$complete[ores2$complete$fight == ores2$best$fight, ]
points(pdata$threat, pdata$loglik, type = "l", col = "red")
arrows(x0 = ores2$best$threat, y0 = -230, x1 = ores2$best$threat, y1 = -220, col = "red", lwd = 2, xpd = TRUE, length = 0.1)

pdata <- ores2$complete[ores2$complete$fight == 10, ]
points(pdata$threat, pdata$loglik, type = "l", col = "red", lty = 3)
x <- pdata$threat[which.max(pdata$loglik)]
arrows(x0 = x, y0 = -230, x1 = x, y1 = -220, col = "red", lwd = 2, xpd = TRUE, length = 0.1, lty = 3)

pdata <- ores2$complete[ores2$complete$threat == ores2$best$threat, ]
points(pdata$fight, pdata$loglik, type = "l", col = "grey")
arrows(x0 = ores2$best$fight, y0 = -230, x1 = ores2$best$fight, y1 = -220, col = "grey", lwd = 2, xpd = TRUE, length = 0.1)
```

```r
pdata <- ores2$complete[ores2$complete$threat == 500, ]
points(pdata$fight, pdata$loglik, type = "l", col = "grey", lty = 3)
x <- pdata$fight[which.max(pdata$loglik)]
arrows(x0 = x, y0 = -230, x1 = x, y1 = -220, col = "grey", lwd = 2, xpd = TRUE, length = 0.1, lty = 3)

legend("bottom", legend = c("one interaction type", "threat (fight fixed)", "fight (threat fixed)"),
       col = c("blue", "red", "grey"), lty = 1, cex = 0.7)
```

This code produces figure 18.

```r
elotable <- list(0:3, 4:10, 11:17, 18:24, 25:31, 32:38, 39:45, 46:52, 53:59, 60:66, 67:74, 75:81, 82:88, 89:96,
                 97:103, 104:111, 112:119, 120:127, 128:135, 136:143, 144:151, 152:159, 160:168, 169:177,
                 178:186, 187:195, 196:205, 206:214, 215:224, 225:235, 236:246, 247:257, 258:269, 270:281,
                 282:294, 295:308, 309:323, 324:338, 339:354, 355:372, 373:391, 392:412, 413:436, 437:463,
                 464:494, 495: 530, 531:576, 577:636, 637:726, 727:920, 921:1000)
alberstable <- list(0:3, 4:10, 11:17, 18:25, 26:32, 33:39, 40:46, 47:53, 54:61, 62:68, 69:76, 77:83, 84:91,
                    92:98, 99:106, 107:113, 114:121, 122:129, 130:137, 138:145, 146:153, 154:162, 163:170,
                    171:179, 180:188, 189:197, 198:206, 207:215, 216:225, 226:235, 236:245, 246:256, 257:267,
                    268:278, 279:290, 291:302, 303:315, 316:328, 329:344, 345:357, 358:374, 375:391, 392:411,
                    412:432, 433:456, 457:484, 485:517, 518:559, 560:619, 620:735, 736:1000)

elotable <- data.frame(rtgdiff = unlist(elotable),
                       P = rep(seq(0.5, 1, by = 0.01), unlist(lapply(elotable, length))))
alberstable <- data.frame(rtgdiff = unlist(alberstable),
                          P = rep(seq(0.5, 1, by = 0.01), unlist(lapply(alberstable, length))))

w <- rep(0, 1001) # winner rating: constant
l <- w - 0:1000 # loser rating: varying

elonorm <- numeric(length(w))
eloexpo <- numeric(length(w))
eloopti <- numeric(length(w))

for(i in 1:length(w)) {
  elonorm[i] <- winprob(w[i], l[i], normprob = TRUE)
  eloexpo[i] <- winprob(w[i], l[i], normprob = FALSE)
  eloopti[i] <- winprob(w[i], l[i], normprob = FALSE, fac = 0.01)
}

plot(0, 0, type = "n", las = 1, yaxs = "i",
     xlim = c(0, 1000), ylim = c(0.5, 1),
     xlab = "rating difference",
     ylab = "winning probability")
points(abs(l), elonorm, "l", col = "red")
points(abs(l), eloexpo, "l", col = "gold")
points(abs(l), eloopti, "l", col = "grey")

points(alberstable$rtgdiff, alberstable$P, type="l", col="red")
points(elotable$rtgdiff, elotable$P, type="l", col="gold")
legend("bottomright",
       legend = c("normal", "exponential", "exponential (alternative)"),
       col = c("red", "gold", "grey"),
       lwd = 2,
       cex = 0.9)
```

This code produces figure 19.

```r
set.seed(123)
n <- 100
xres <- data.frame(nid = sample(8:26, n, TRUE),
                   r1 = NA, r2 = NA,
                   k = sample(50:300, n, TRUE))

for (i in 1:length(xres$nid)) {
  xd <- randomsequence(nID = xres$nid[i], avgIA = sample(3:100, 1),
                       reversals = runif(1, 0, 0.4))$seqdat
  allids <- letters[1:xres$nid[i]]
  w <- allids[xd$winner]
  l <- allids[xd$loser]
  kvals <- rep(res$k[i], length(w))
  svals <- rep(1000, length(allids))
  xres1 <- fastelo(w, l, allids, kvals, svals, NORMPROB = TRUE)
  xres2 <- fastelo(w, l, allids, kvals, svals, NORMPROB = FALSE)
  xind <- sample(1:xres$nid[i], 1)
  xres$r1[i] <- xres1[[1]][xind]
  xres$r2[i] <- xres2[[1]][xind]
}

plot(xres$r1, xres$r2, xlab = "normal", ylab = "exponential", las = 1)
abline(0, 1)
```

References

Albers, Paul C H, and Han de Vries. 2001. "Elo-Rating as a Tool in the Sequential Estimation of Dominance Strengths." *Animal Behaviour* 61: 489–95. https://doi.org/10.1006/anbe.2000.1571.

Appleby, Michael C. 1983. "The Probability of Linearity in Hierarchies." *Animal Behaviour* 31: 600–608. https://doi.org/10.1016/S0003-3472(83)80084-0.

David, Herbert A. 1987. "Ranking from Unbalanced Paired-Comparison Data." *Biometrika* 74: 432–36. https://doi.org/10.1093/biomet/74.2.432.

de Vries, Han. 1995. "An Improved Test of Linearity in Dominance Hierarchies Containing Unknown or Tied Relationships." *Animal Behaviour* 50: 1375–89. https://doi.org/10.1016/0003-3472(95)80053-0.

———. 1998. "Finding a Dominance Order Most Consistent with a Linear Hierarchy: A New Procedure and Review." *Animal Behaviour* 55: 827–43. https://doi.org/10.1006/anbe.1997.0708.

de Vries, Han, Jeroen M G Stevens, and Hilde Vervaecke. 2006. "Measuring and Testing the Steepness of Dominance Hierarchies." *Animal Behaviour* 71: 585–92. https://doi.org/10.1016/j.anbehav.2005.05.015.

Duboscq, Julie, Valéria Romano, Cédric Sueur, and Andrew J J MacIntosh. 2016. "Scratch That Itch: Revisiting Links Between Self-Directed Behaviour and Parasitological, Social and Environmental Factors in a Free-Ranging Primate." *Royal Society Open Science* 3: 160571. https://doi.org/10.1098/rsos.160571.

Elo, Arpad E. 1978. *The Rating of Chess Players, Past and Present.* New York: Arco.

Farine, Damien R, and Alfredo Sánchez-Tójar. 2019. *AniDom: Inferring Dominance Hierarchies and Estimating Uncertainty.* https://CRAN.R-project.org/package=aniDom.

Feldblum, Joseph T, Steffen Foerster, and Mathias Franz. 2019. *EloOptimized: Optimized Elo Rating Method for Obtaining Dominance Ranks.* https://github.com/jtfeld/EloOptimized.

Foerster, Steffen, Mathias Franz, Carson M Murray, Ian C Gilby, Joseph T Feldblum, Kara K Walker, and Anne E Pusey. 2016. "Chimpanzee Females Queue but Males Compete for Social Status." *Scientific Reports* 6: 35404. https://doi.org/10.1038/srep35404.

Franz, Mathias, Emily McLean, Jenny Tung, Jeanne Altmann, and Susan C Alberts. 2015. "Self-Organizing Dominance Hierarchies in a Wild Primate Population." *Proceedings of the Royal Society B: Biological Sciences* 282: 20151512. https://doi.org/10.1098/rspb.2015.1512.

Gammell, Martin P, Han de Vries, Dómhnall J Jennings, Caitríona M Carlin, and Thomas J Hayden. 2003. "David's Score: A More Appropriate Dominance Ranking Method Than Clutton-Brock et Al.'s Index." *Animal Behaviour* 66: 601–5. https://doi.org/10.1006/anbe.2003.2226.

Klass, Keren, and Marina Cords. 2011. "Effect of Unknown Relationships on Linearity, Steepness and Rank Ordering of Dominance Hierarchies: Simulation Studies Based on Data from Wild Monkeys." *Behavioural Processes* 88: 168–76. https://doi.org/10.1016/j.beproc.2011.09.003.

McDonald, David B, and Daizaburo Shizuka. 2013. "Comparative Transitive and Temporal Orderliness in Dominance Networks." *Behavioral Ecology* 24: 511–20. https://doi.org/10.1093/beheco/ars192.

Neumann, Christof, Julie Duboscq, Constance Dubuc, Andri Ginting, Ade Maulana Irwan, Muhammad Agil, Anja Widdig, and Antje Engelhardt. 2011. "Assessing Dominance Hierarchies: Validation and Advantages of Progressive Evaluation with elo-Rating." *Animal Behaviour* 82: 911–21. https://doi.org/10.1016/j.anbehav.2011.07.016.

Neumann, Christof, David B McDonald, and Daizaburo Shizuka. 2018. "Dominance Ranks, Dominance Ratings and Linear Hierarchies: A Critique." *Animal Behaviour* 144: e1–e16. https://doi.org/10.1016/j.anbehav.2018.07.012.

Newton-Fisher, Nicholas E. 2017. "Modeling Social Dominance: Elo-Ratings, Prior History, and the Intensity of Aggression." *International Journal of Primatology* 38: 427–47. https://doi.org/10.1007/s10764-017-9952-2.

Sánchez-Tójar, Alfredo, Julia Schroeder, and Damien R Farine. 2018. "A Practical Guide for Inferring Reliable Dominance Hierarchies and Estimating Their Uncertainty." *Journal of Animal Ecology* 87: 594–608. https://doi.org/10.1111/1365-2656.12776.

Shizuka, Daizaburo, and David B McDonald. 2012. "A Social Network Perspective on Measurements of Dominance Hierarchies." *Animal Behaviour* 83: 925–34. https://doi.org/10.1016/j.anbehav.2012.01.011.

———. 2015. “The Network Motif Architecture of Dominance Hierarchies.” *Journal of the Royal Society Interface* 12: 20150080. https://doi.org/10.1098/rsif.2015.0080.

van Hooff, Jan A R A M, and Joep A B Wensing. 1987. “Dominance and Its Behavioural Measures in a Captive Wolf Pack.” In *Man and Wolf*, edited by Harry Frank, 219–52. Dordrecht: Junk.