tcR: a package for T-cell receptor repertoire data analysis

Vadim Nazarov
vdm.nazarov@gmail.com

Mikhail Pogorelyy
m.pogorely@gmail.com

August 2014

Abstract

Abstract High-throughput technologies has open new possibilities to analyse data of repertoires of immunological receptors (i.e., T-cell or B-cell receptors). Here we present a manual to an R package *tcR*. Paper is published in Journal of Something:

`Nazarov et al tcR: an R package for T-cell repertoire data analysis.`

Contents

1 Introduction

The *tcR* package is designed to help researchers in the immunology field to analyse TCR and BCR repertoires. In this vignette, we will cover main procedures for TCR repertoire analysis.

1.1 Package features

- Shared clones statistics *(number of shared clones, clonotypes, using V-segments or not; Jaccard index for number of shared clones; sequential intersection among the most abundant clones ("top-cross"))*
- V- and J-segments usage and it's analysis *(PCA, Shannon Entropy, Jensen-Shannon Divergence)*
- Diversity evaluation *(ecological diversity index, Gini index, inverse Simpson index, rarefaction analysis)*
- Artificial repertoire generation (beta chain only, for now)
- Spectratyping
- Various visualisation procedures

1.2 Data, provided along with the package

With the package few datasets are provided.

human.alphabets.rda - .rda file with character vectors for Variable and Joining segments names for human species. Few segments are merged together due to inefficiency of our sequencing technology. User can provide their own alphabets by assigning new character vectors to this alphabets. Variables stored at this file are:

```
> data(human.alphabets)
> V_ALPHA_ALPHABET
> J_ALPHA_ALPHABET
> V_BETA_ALPHABET
> J_BETA_ALPHABET
```

mouse.alphabets.rda - .rda file with character vector with names for V- and J-segments for mouse species. Variables stored at this file are:

```
> data(mouse.alphabets)
> V_BETA_ALPHABET
> J_BETA_ALPHABET
```

twa.rda, *twb.rda* - data frames with downsampled to the 10000 most abundant clonesets and 4 samples data of twins data (alpha and beta chains). Link: TCR data at Laboratory of Comparative and Functional Genomics. Variables:

```
> data(twa)
> head(twa[[1]])
> data(twb)
> head(twb[[1]])
```

1.3 Quick start (using examples pipelines with automatic report generation)

For exploratory analysis of a single repertoire, use the RMarkdown report file:

```
<path to the tcR package>/inst/library.report.Rmd
```

Analysis in the file included statistics and visualisation of number of clones, clonotypes, in- and out-of-frames, unique amino acid CDR3 sequences, V- and J-usage, most frequent k-mers, rarefaction analysis.

For analysis of a group of repertoires ("cross-analysis"), use the RMarkdown report fil:

```
<path to the tcR package>/inst/crossanalysis.report.Rmd
```

Analysis in the file included statistics and visualisation of number of shared clones and clonotypes, V-usage for individuals and groups, J-usage for individuals, Jensen-Shannon divergence among V-usages of repertoires and top-cross.

You will need the *knitr* package installed in order to generate reports from default pipelines. In RStudio you can run a pipeline file as follows:

```
Run RStudio -> load the pipeline .Rmd files -> press the knitr button
```

1.4 MiTCR: a tool for retrieving CDR3 sequences from NGS data

MiTCR is a tool for retrieving TCR CDR sequences from NGS data (link). Pipeline for processing files typically looks like follows:

```
NGS .fastq files -> run MiTCR -> tab-separated files with clonesets -> tcR parser
```

You can start MiTCR from an R session with `startmitcr` function. E.g., to run code above you need to do following:

```
> startmitcr('raw/TwA1_B.fastq.gz', 'mitcr/TwA1_B.txt', .file.path = '~/data/',
+             pset = 'flex', level = 1, 'debug', .mitcr.path = '~/programs/', .mem = '8g')
```

Run MiTCR on all files from the ' /data/raw/' directory:

```
> startmitcr(.file.path = '~/data/raw', pset = 'flex', level = 1, 'debug',
+             .mitcr.path = '~/programs/', .mem = '8g')
```

For parsing data *tcR* offers `parse.file`, `parse.file.list` and `parse.folder` functions.

```
> # Parse file in "~/data/twb1.txt".
> twb1 <- parse.file("~/data/twb1.txt")
> # Parse files "~/data/twb1.txt" and "~/data/immdat2.txt".
> twb12 <- parse.file.list(c("~/data/twb1.txt", "~/data/twb2.txt"))
> # Parse all files in "~/data/".
> twb <- parse.folder("~/data/")
```

1.5 Structure of a MiTCR data frame (clonesets representation)

The package basically operates with data frames with specific column names, which called MiTCR data frames. MiTCR data frame is an output file from the MiTCR tool. This files are tab-delimited files with columns stands for CDR3 nucleotide sequence, V-segment and oth.:

```
  Read.count  Percentage                     CDR3.nucleotide.sequence
1      81516 0.031979311 TGTGCCAGCAGCCAAGCTCTAGCGGGAGCAGATACGCAGTATTTT
2      46158 0.018108114 TGTGCCAGCAGCTTAGGCCCCAGGAACACCGGGGAGCTGTTTTTT
3      32476 0.012740568    TGTGCCAGCAGTTATGGAGGGGCGGCAGATACGCAGTATTTT
4      30356 0.011908876 TGCAGTGCTGGAGGGATTGAAACCTCCTACAATGAGCAGTTCTTC
5      27321 0.010718224          TGTGCCAGCTCACCCATCTTAGGGGAGCAGTTCTTC
6      23760 0.009321218       TGTGCCAGCAAAAAAGACAGGGACTATGGCTACACCTTC
  CDR3.amino.acid.sequence         V.segments J.segments   D.segments
1          CASSQALAGADTQYF            TRBV4-2    TRBJ2-3        TRBD2
2          CASSLGPRNTGELFF             TRBV13    TRBJ2-2 TRBD1, TRBD2
3           CASSYGGAADTQYF TRBV12-4, TRBV12-3    TRBJ2-3        TRBD2
4          CSAGGIETSYNEQFF           TRBV20-1    TRBJ2-1 TRBD1, TRBD2
5             CASSPILGEQFF             TRBV18    TRBJ2-1 TRBD1, TRBD2
6            CASKKDRDYGYTF            TRBV6-5    TRBJ1-2        TRBD1
  Last.V.nucleotide.position First.D.nucleotide.position
1                         15                         18
2                         16                         17
3                         12                         15
4                         12                         13
```

```
5                         13                          20
6                          9                          15
  Last.D.nucleotide.position First.J.nucleotide.position VD.insertions
1                         27                          28             2
2                         20                          23             0
3                         20                          25             2
4                         15                          23             0
5                         23                          24             6
6                         21                          22             5
  DJ.insertions Total.insertions Rank Diff Index
1             0                2    1    1     1
2             2                2    2    1     2
3             4                6    3    1     3
4             7                7    4    1     4
5             0                6    5    1     5
6             0                5    6    1     6
```

In our analysis only few columns are broadly used. Hence, to do almost all analysis you just need a data frames with following columns:

- *Read.count*
- *CDR3.amino.acid.sequence*
- *V.segments*

Additionally, for analysis of J-segments usage or nucleotide sequences intersection (see Subsection 3.1) you should provide:

- *J.segments*
- *CDR3.nucleotide.sequence*

Any data frame with this columns is suitable for processing with the package, hence user can generate their own table files and load them for the further analysis using `read.csv`, `read.table` and other base R functions.

2 Repertoire descriptive statistics

For exploratory analysis, a *tcR* provides functions for computing descriptive statistics.

2.1 Sequences summary

To get a general view of subject's repertoire (overall count of sequences, in- and out-of-frames numbers and percentage) use the `mitcr.stats` function. It returns a `summary` of counts of nucleotide sequences ('clones') and amino acid sequences ('clonotypes'), as well as summary of read counts:

```
> # Load the package.
> library(tcR)
> # Load additional packages for making this vignette.
> # Load the twins data, provided with the package.
> data(twb)
> # Load human alphabets of V-genes and J-genes, provided with the package.
> data(human.alphabets)
> mitcr.stats(twb)
```

```
       #Nucleotide clones #Aminoacid clonotypes %Aminoacid clonotypes
Subj.A             10000                  9850                0.9850
Subj.B             10000                  9838                0.9838
Subj.C             10000                  9775                0.9775
Subj.D             10000                  9872                0.9872
       #In-frames %In-frames #Out-of-frames %Out-of-frames Sum.Read.count
```

```
Subj.A       9654     0.9654            346         0.0346        1410263
Subj.B       9600     0.9600            400         0.0400        2251408
Subj.C       9808     0.9808            192         0.0192         969949
Subj.D       9288     0.9288            712         0.0712        1419130
       Min.Read.count 1st Qu.Read.count Median.Read.count Mean.Read.count
Subj.A             22                26                33          141.00
Subj.B             20                24                31          225.10
Subj.C             23                28                39           96.99
Subj.D             32                37                48          141.90
       3rd Qu.Read.count Max.Read.count
Subj.A                57          81520
Subj.B                55         171200
Subj.C                68         104600
Subj.D                83          33590
```

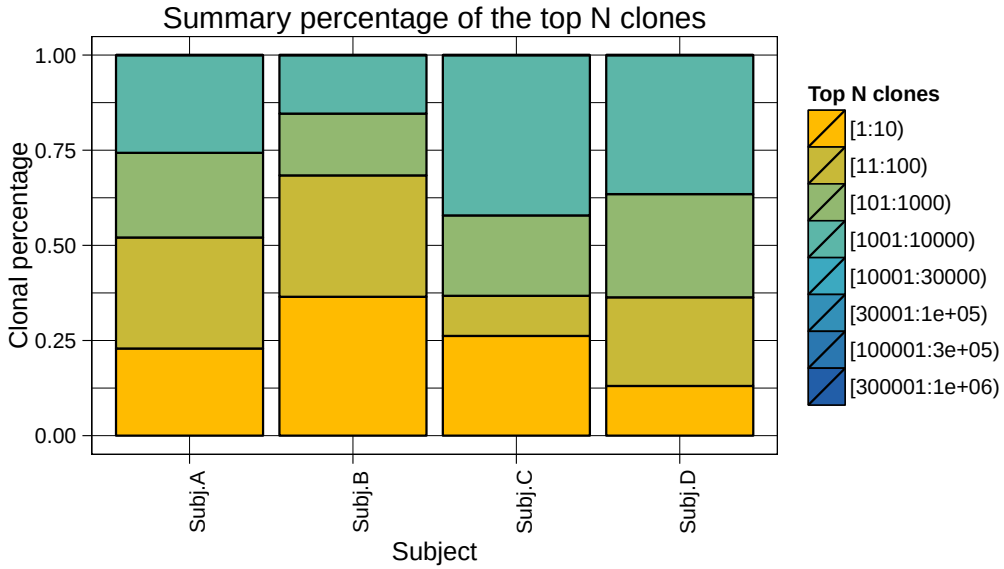
2.2 Percentage and counts of the most abundant clonotypes

Function `clonal.proportion` is used to get the number of most abundant by the count of reads clones. E.g., compute number of clones which fill up (approx.) the 25% from total repertoire's "Read.count":

```
>                             # How many clones fill up approximately
> clonal.proportion(twb, 25)  # the 25% of the sum of values in 'Read.count'?

      Subj.A  Subj.B  Subj.C  Subj.D
[1,] 12.0000  6.0000  7.0000 38.0000
[2,] 25.1000 26.5000 25.2000 25.2000
[3,]  0.0012  0.0006  0.0007  0.0038
```

To get a proportion of the most abundant clones' sum of reads to the overall overlall number of reads in a repertoire, use `top.proportion`, i.e. get
$(\sum$ `reads of top clones)/(`$\sum$ `reads for all clones)`. E.g., get a proportion of the top-10 clones' reads to the overall number of reads:

```
>                           # What accounts a proportion of the top-10 clones' reads
> top.proportion(twb, 10)   # to the overall number of reads?

   Subj.A    Subj.B    Subj.C    Subj.D
0.2289069 0.3648699 0.2620158 0.1305398

> vis.top.proportions(twb)  # Plot this proportions.
```



Function `tailbound.proportion` with two arguments `.col` and `.bound` gets subset of the given data frame with clones having column `.col` with value $\leq$ `.bound` and computes the ratio of sums of count reads of such subset to the overall data frame. E.g., get proportion of sum of reads of sequences which has "Read.count" <= 100 to the overall number of reads:

```
>                                  # What is a proportion of sequences which
>                                  # have 'Read.count' <= 100 to the
> tailbound.proportion(twb, 100)   # overall number of reads?
```

```
Subj.A Subj.B Subj.C Subj.D
0.8651 0.8641 0.8555 0.8020
```

2.3 In- and out-of-frame CDR3 sequences subsetting and statistics

Functions for performing subsetting and counting cardinality of in-frame and out-of-frame subsets are: `count.inframes`, `count.outframes`, `get.inframes`, `get.outframes`. Parameter `.head` for this functions is a parameter to the `head` function, that applied before subsetting. Functions accept both data frames and list of data frames as parameters. E.g., get data frame with only in-frame sequences and count out-of-frame sequences in the first 5000 rows for this data frame:

```
> imm.in <- get.inframes(twb) # Return all in-frame sequences from the 'twb'.
>                             # Count the number of out-of-frame sequences
> count.outframes(twb, 5000)  # from the first 5000 sequences.
```

```
Subj.A Subj.B Subj.C Subj.D
   172    212     73    326
```

```
> head(freq.Vb(imm.in)[,2] / freq.Vb(twb)[,2])       # Compare V-usage between in-frames and all seq.
```

```
[1] 1.0356591 0.7577993 0.9511155 1.0155492 1.0356591 0.9691997
```

General function with parameter stands for 'all' (all sequences), 'in' (only in-frame sequences) or 'out' (only out-of-frame sequences) is `count.frames`:

```
> imm.in <- get.frames(twb, 'in') # Similar to 'get.inframes(twb)'.
> count.frames(twb[[1]], 'all')   # Just return number of rows.
```

```
[1] 10000
```

```
> flag <- 'out'
> count.frames(twb, flag, 5000)   # Similar to 'count.outframes(twb, 5000)'.
```

```
Subj.A Subj.B Subj.C Subj.D
   172    212     73    326
```

2.4 V-, D-, J-segments statistics

To access V- and J-usage of a repertoire, *tcR* provides functions `freq.segments`, `freq.segments.2D` and a family of functions `freq.[VJ][ab]` for simplier use. Function `freq.segments`, depending on parameters, computes frequencies or counts of the given elements (e.g., V-segments) in the given column (e.g., "V.segments") of the input data frame(s). Function `freq.segments.2D` computes joint distributions or counts of the two given elements (e.g., V-segments and J-segments). For plotting V-usage and J-usage see section 6.4. V and J alphabets for humans are stored in the .rda file "human.alphabets.rda" (they are identical to those form IMGT: link to beta genes (red ones) and link to alpha genes (red ones)). All of the mentioned functions are accept data frames as well as list of data frames. Output for those functions are data frames with the first column stands for segment and the other for frequencies.

```
> # Equivalent to freq.Vb(twb[[1]]) by default.
> imm1.vs <- freq.segments(twb[[1]])
> head(imm1.vs)
```

```
       Segment        Freq
Other    Other 0.001691711
1     TRBV10-1 0.004080008
2     TRBV10-2 0.004876107
3     TRBV10-3 0.030749328
4     TRBV11-1 0.004378545
5     TRBV11-2 0.018608817
```

```
> imm.vs.all <- freq.segments(twb)  # Equivalent to freq.Vb(twb) by default.
> imm.vs.all[1:10, 1:4]
```

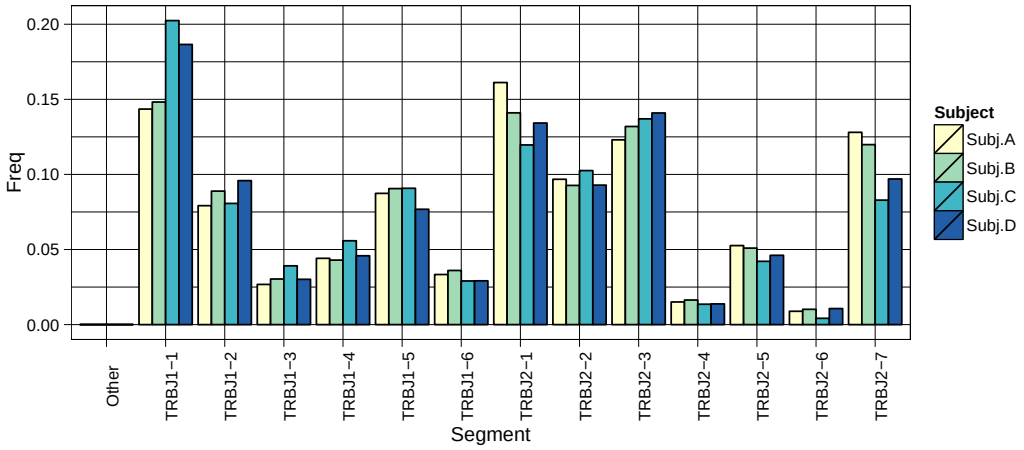
```
               Segment      Subj.A      Subj.B       Subj.C
1                Other 0.001691711 0.001492686 0.0022887850
2              TRBV10-1 0.004080008 0.003582446 0.0009951239
3              TRBV10-2 0.004876107 0.006567818 0.0022887850
4              TRBV10-3 0.030749328 0.030649816 0.0327395761
5              TRBV11-1 0.004378545 0.003482934 0.0033834212
6              TRBV11-2 0.018608817 0.022987362 0.0222907752
7              TRBV11-3 0.002089760 0.002388297 0.0027863469
8  TRBV12-4, TRBV12-3 0.050154244 0.049358145 0.0629913424
9              TRBV12-5 0.001592198 0.002288785 0.0037814708
10               TRBV13 0.006866355 0.003980496 0.0044780575
```

```
> imm1.vj <- freq.segments.2D(twb[[1]])
> imm1.vj[1:5, 1:5]
```

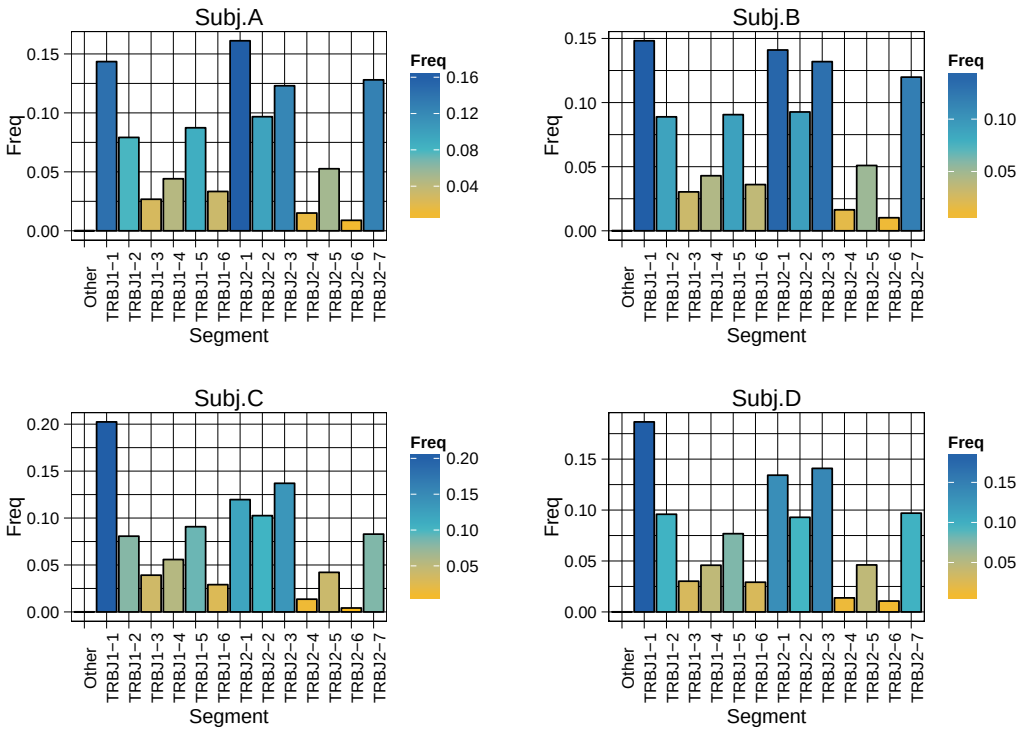
```
   Segment      TRBJ1-1      TRBJ1-2      TRBJ1-3      TRBJ1-4
1 TRBV10-1 0.0006598793 0.0001885370 9.426848e-05 1.885370e-04
2 TRBV10-2 0.0005656109 0.0005656109 1.885370e-04 1.885370e-04
3 TRBV10-3 0.0040535445 0.0023567119 1.131222e-03 6.598793e-04
4 TRBV11-1 0.0006598793 0.0002828054 9.426848e-05 9.426848e-05
5 TRBV11-2 0.0022624434 0.0011312217 4.713424e-04 1.036953e-03
```

You can also directly visualise segments usage with functions `vis.V.usage` and `vis.J.usage` with argument `.cast.freq` equal to TRUE:

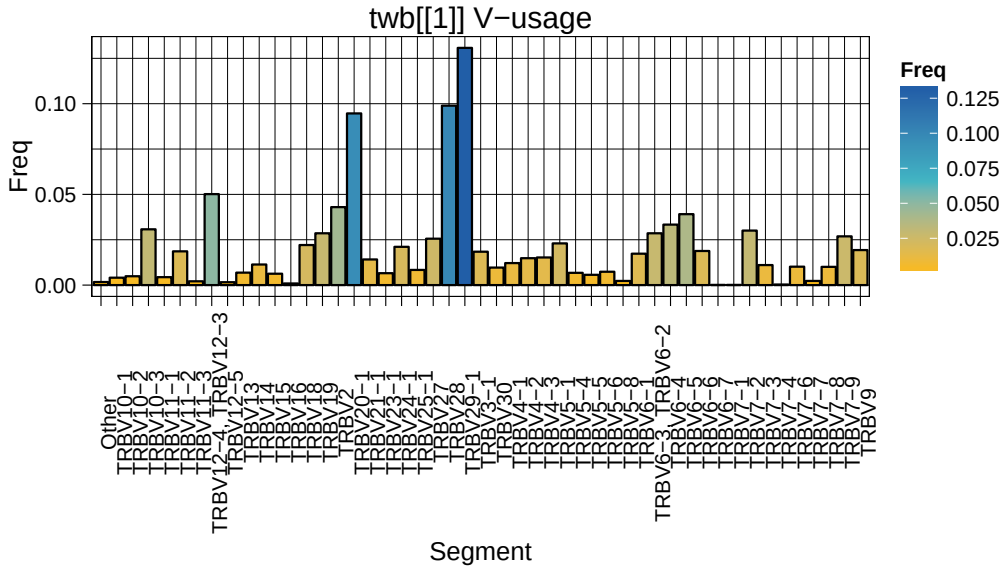
```
> # Put ".dodge = F" to get distinct plot for every data frame in the given list.
> vis.J.usage(twb, .cast.freq = T, .main = 'twb J-usage dodge', .dodge = T)
```




```
> vis.J.usage(twb, .cast.freq = T, .main = 'twb J-usage column', .dodge = F, .ncol = 2)
```




```
> vis.V.usage(imm1.vs, .cast.freq = F, .main = 'twb[[1]] V-usage', .coord.flip = F)
```



2.5 Search for a target CDR3 sequences

For exact or fuzzy search of sequences the package employed a function `find.clonotypes`. Input arguments for this function are data frame or list of data frames, targets (character vector or data frame having one column with sequences and additional columns with, e.g., V-segments), value of which column or columns to return, method to be used to compare sequences among each other (either "exact" for exact matching, "hamm" for matching sequences by Hamming distance (two sequences are matched if H <= 1) or "lev" for matching sequences by Levenshtein distance (two sequences are matched if L <= 1)), and name of column name from which sequences for matching are obtained. Sounds very complex, but in practice it's very easy, therefore let's go to examples. Suppose we want to search for some CDR3 sequences in a number of repertoires:

```
> cmv
```

```
  CDR3.amino.acid.sequence V.segments
1             CASSSANYGYTF    TRBV4-1
2             CSVGRAQNEQFF    TRBV4-1
3            CASSLTGNTEAFF    TRBV4-1
4         CASSALGGAGTGELFF    TRBV4-1
5         CASSLIGVSSYNEQFF    TRBV4-1
```

We will search for them using all methods of matching (exact, hamming or levenshtein) and with and without matching by V-segment. Also, for the first case (exact matching and without V-segment) we return "Total.insertions" column along with the "Read.count" column, and for the second case output will be a "Rank" - rank (generated by `set.rank`) of a clone or a clonotype in a data frame.

```
> twb <- set.rank(twb)
> # Case 1.
> cmv.imm.ex <-
+   find.clonotypes(.data = twb[1:2], .targets = cmv[,1], .method = 'exact',
+                   .col.name = c('Read.count', 'Total.insertions'),
+                   .verbose = F)
> head(cmv.imm.ex)
```

```
                   CDR3.amino.acid.sequence Read.count.Subj.A Read.count.Subj.B
CASSALGGAGTGELFF           CASSALGGAGTGELFF               153               319
CASSALGGAGTGELFF.1         CASSALGGAGTGELFF                NA                35
```

```
CASSLTGNTEAFF                 CASSLTGNTEAFF               35              263
CASSLTGNTEAFF.1               CASSLTGNTEAFF               35               35
CASSLTGNTEAFF.2               CASSLTGNTEAFF               NA               28
CASSSANYGYTF                   CASSSANYGYTF               NA            15320
                  Total.insertions.Subj.A Total.insertions.Subj.B
CASSALGGAGTGELFF                        9                      10
CASSALGGAGTGELFF.1                     NA                       9
CASSLTGNTEAFF                           2                       2
CASSLTGNTEAFF.1                         1                       0
CASSLTGNTEAFF.2                        NA                       1
CASSSANYGYTF                           NA                       1
```

```
> # Case 2.
> # Search for CDR3 sequences with hamming distance <= 1
> # to the one of the cmv$CDR3.amino.acid.sequence with
> # matching V-segments. Return ranks of found sequences.
> cmv.imm.hamm.v <-
+   find.clonotypes(twb[1:3], cmv, 'hamm', 'Rank',
+                   .target.col = c('CDR3.amino.acid.sequence', 'V.segments'),
+                   .verbose = F)
> head(cmv.imm.hamm.v)
```

```
                CDR3.amino.acid.sequence V.segments Rank.Subj.A Rank.Subj.B
CAQVLLIETQYF                  CAQVLLIETQYF    TRBV4-1          NA      8567.5
CASAGLDLFVTGELFF          CASAGLDLFVTGELFF    TRBV4-1          NA          NA
CASALQAYYNEQFF              CASALQAYYNEQFF    TRBV4-1        1403          NA
CASCDDYNSPLHF                CASCDDYNSPLHF    TRBV4-1          NA          NA
CASEDRGRTDTQYF              CASEDRGRTDTQYF    TRBV4-1          NA          NA
CASGGSLGQNTEAFF            CASGGSLGQNTEAFF    TRBV4-1          NA          NA
                Subj.C.Rank
CAQVLLIETQYF             NA
CASAGLDLFVTGELFF      7532.5
CASALQAYYNEQFF            NA
CASCDDYNSPLHF         7190.5
CASEDRGRTDTQYF        9729.5
CASGGSLGQNTEAFF        737.5
```

```
> # Case 3.
> # Similar to the previous example, except
> # using levenshtein distance and the "Read.count" column.
> cmv.imm.lev.v <-
+   find.clonotypes(twb[1:3], cmv, 'lev',
+                   .target.col = c('CDR3.amino.acid.sequence', 'V.segments'),
+                   .verbose = F)
> head(cmv.imm.lev.v)
```

```
                CDR3.amino.acid.sequence V.segments Read.count.Subj.A
CASSALGGAGTGELFF         CASSALGGAGTGELFF    TRBV4-1                NA
CASSLIGVSSYNEQFF         CASSLIGVSSYNEQFF    TRBV4-1                NA
CASSLTGNTEAFF               CASSLTGNTEAFF    TRBV4-1                NA
CASSSANYGYTF                 CASSSANYGYTF    TRBV4-1                NA
CSVGRAQNEQFF                 CSVGRAQNEQFF    TRBV4-1                NA
                Read.count.Subj.B Subj.C.Read.count
CASSALGGAGTGELFF                NA                NA
CASSLIGVSSYNEQFF                NA                NA
CASSLTGNTEAFF                   NA                NA
CASSSANYGYTF                    NA                NA
CSVGRAQNEQFF                    NA                NA
```

3 Cloneset analysis

Repertoires (both TCRs and BCRs) can be viewed as sets of elements, e.g. sets of CDR3 amino acid sequences or sets of tuples (CDR3 amino acid sequence, V-segment). *tcR* provides functions for evaluating similarity and diversity of such sets.

3.1 Intersections between sets of CDR3 sequences

A simplest way to evaluate similarity of two sets is compute the number of elements in their intersection set (i.e., number of shared elements). *tcR* overrides default function `intersect`, adding new parameters, thought `intersect(x,y)` works as the old function `base::intersect` if `x` and `y` both are not data frames. For data frames `base::intersect` isn't working, but `tcR::intersect` is: by default the function intersects the "CDR3.nucleotide.sequence" columns of the given data frames, but user can change target columns by using arguments `.type` or `.col`. As in the `find.clonotypes`, user can choose which method apply to the elements: exact match of elements, match by Hamming distance or match by Levenshtein distance.

```
> # Equivalent to intersect(twb[[1]]$CDR3.nucleotide.sequence,
> #                         twb[[2]]$CDR3.nucleotide.sequence)
> # or intersectCount(twb[[1]]$CDR3.nucleotide.sequence,
> #                   twb[[2]]$CDR3.nucleotide.sequence)
> # "n" stands for a "CDR3.nucleotide.sequence" column, "e" for exact match.
> intersect(twb[[1]], twb[[2]], 'n0e')
```

```
[1] 46
```

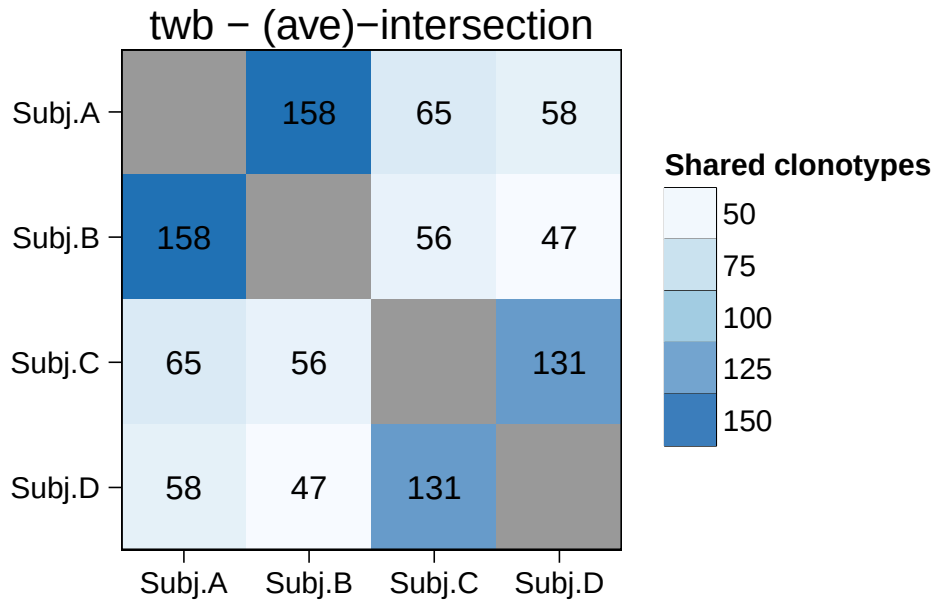
```
> # "a" stands for "CDR3.amino.acid.sequence" column.
> # "v" means that intersect should also use the "V.segments" column.
> intersect(twb[[1]], twb[[2]], 'ave')
```

```
[1] 158
```

```
> # Works also on lists, performs all possible pairwise intersections.
> intersect(twb, 'ave')
```

```
       Subj.A Subj.B Subj.C Subj.D
Subj.A     NA    158     65     58
Subj.B    158     NA     56     47
Subj.C     65     56     NA    131
Subj.D     58     47    131     NA
```

```
> # Plot a heatmap of number of shared clonotypes.
> vis.heatmap(intersect(twb, 'ave'), .title = 'twb - (ave)-intersection', .labs = '')
```



See the `vis.heatmap` function in the Section "Plots" for the visualisation of the intersection results.

Functions `intersectCount`, `intersectLogic` and `intersectIndices` are more flexible in terms of choosing which columns to match. They all have parameter `.col` that specifies names of columns which will used in computing intersection. Function `intersectCount` returns number of similar elements; `intersectIndices(x, y)` returns 2-column matrix with the first column stands for an index of an element in the given `x`, and the second column stands for an index of that element of `y` which is similar to a relative element in `x`; `intersec.logic(x, y)` returns logical vector of `length(x)` or `nrow(x)`, where TRUE at position `i` means that element with index i has been found in the `y`.

```
> # Get logic vector of shared elements, where
> # elements are tuples of CDR3 nucleotide sequence and corresponding V-segment
> imm.1.2 <- intersectLogic(twb[[1]], twb[[2]],
+                           .col = c('CDR3.amino.acid.sequence', 'V.segments'))
> # Get elements which are in both twb[[1]] and twb[[2]].
> head(twb[[1]][imm.1.2, c('CDR3.amino.acid.sequence', 'V.segments')])
```

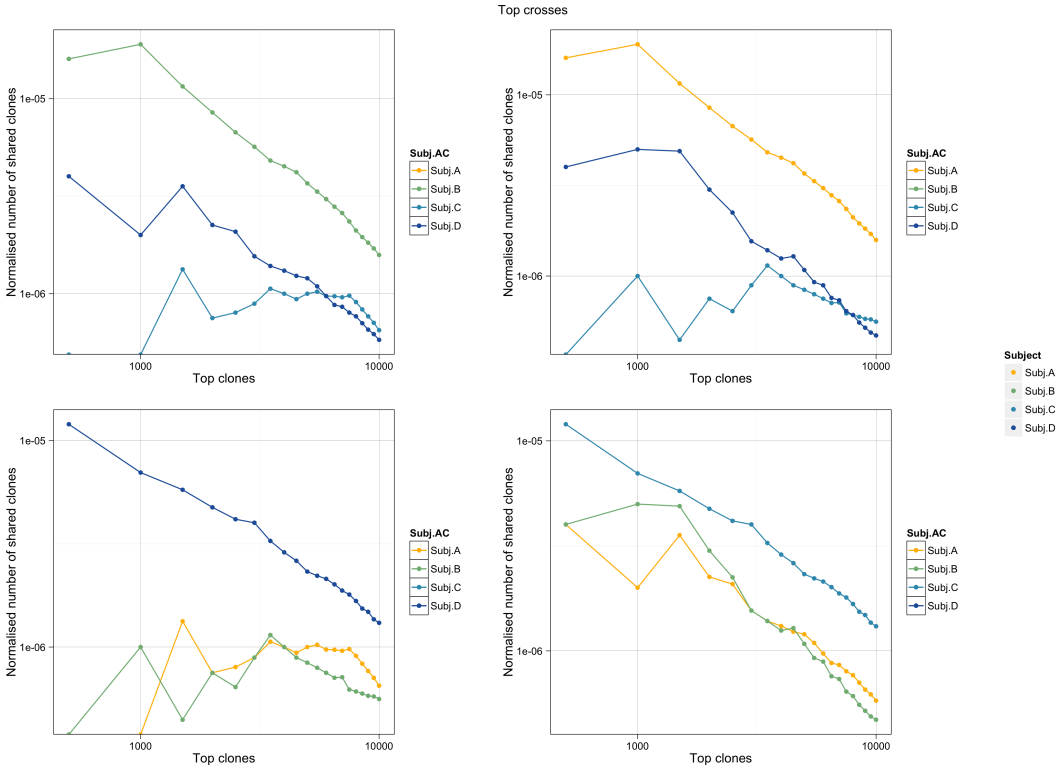
```
   CDR3.amino.acid.sequence V.segments
8             CASSLGLHYEQYF     TRBV28
14            CAWSRQTNTEAFF     TRBV30
17            CASSLGVGYEQYF     TRBV28
```

```
19            CASSLGLHYEQYF     TRBV28
30            CASSLGLNYEQYF     TRBV28
66            CASSLGVSYEQYF     TRBV28
```

3.2 Top cross

Number of shared clones among the most abundant clones may differ signigicantly from those with less count. To support research *tcR* offers the `top.cross` function. that will apply `intersect` to the first 1000 clones, 2000, 3000 and so on up to the first 100000 clones, if supplied `.n == seq(1000, 100000, 1000)`.

```
> twb.top <- top.cross(.data = twb, .n = seq(500, 10000, 500), .verbose = F, .norm = T)
> top.cross.plot(twb.top)
```



3.3 Diversity evaluation

For assessing the distribution of clones in the given repertoire, *tcR* provides functions for evaluating the diversity (functions `diversity` and `inverse.simpson`) and the skewness of the clonal distribution (function `gini`). Function `diversity` computes the ecological diversity index (with parameter `.q` for penalties for clones with large count). Function `inverse.simpson` computes the Inverse Simpson Index (i.e., inverse probability of choosing two similar clones). Function `gini` computes the Gini index of clonal distribution.

```
> # Evaluate the diversity of clones by the ecological diversity index.
> sapply(twb, function (x) diversity(x$Read.count))
```

```
  Subj.A   Subj.B   Subj.C   Subj.D
34.55417 23.97224 15.87257 98.03479
```

```
> # Compute the diversity as inverse probability of choosing two similar clones.
> sapply(twb, function (x) inverse.simpson(x$Read.count))
```

```
   Subj.A    Subj.B    Subj.C    Subj.D
117.63383  56.09537  55.31047 354.18601
```

```
> # Evaluate the skewness of clonal distribution.
> sapply(twb, function (x) gini(x$Read.count))
```

```
   Subj.A    Subj.B    Subj.C    Subj.D
0.7609971 0.8555769 0.6205305 0.6607465
```

See also the `entropy` function for accessing the repertoire diversity, which is described in Subsection 4.1.

3.4 More complicated repertoire similarity measures

tcR also provides more complex measures for evaluating the similarity of sets.

· Cosine similarity (function `cosine.similarity`) is a measure of similarity between two vectors of an inner product space that measures the cosine of the angle between them.

· Tversky index (function `tversky.index`) is an asymmetric similarity measure on sets that compares a variant to a prototype. If using default arguments, it's similar to Dice's coefficient.

· Overlap coefficient (function `overlap.coef`) is a similarity measure that measures the overlap between two sets, and is defined as the size of the intersection divided by the smaller of the size of the two sets.

· Morisita's overlap index (function `morisitas.index`) is a statistical measure of dispersion of individuals in a population and is used to compare overlap among samples. The formula is based on the assumption that increasing the size of the samples will increase the diversity because it will include different habitats (i.e. different faunas) (Morisita, 1959).

```
> cols <- c('CDR3.amino.acid.sequence', 'Read.count')
> # Apply the Morisitas overlap index to the each pair of repertoires.
> apply.symm(twb, function (x,y) morisitas.index(x[, cols], y[, cols]), .verbose = F)
```

```
             Subj.A       Subj.B       Subj.C       Subj.D
Subj.A           NA 0.0012404670 0.0000552547 0.0002881564
Subj.B 0.0012404670           NA 0.0001017043 0.0003148358
Subj.C 0.0000552547 0.0001017043           NA 0.0005150483
Subj.D 0.0002881564 0.0003148358 0.0005150483           NA
```

To visualise similarity among repertoires the `vis.heatmap` function is appropriate.

4 Analysis of segments usage

To evaluate V- and J-segments usage of repertoires, the package implements subroutines for two approaches to analysis: measures from the information theory and PCA (Principal Component Analysis).

4.1 Information measures

To assess the diversity of segments usage user can use the `entropy` function. Kullback-Leibler assymetric measure (function `kl.div`) and Jensen-Shannon symmetric measure (functions `js.div` for computing JS-divergence between the given distributions, `js.div.seg` for computing JS-divergence between segments distributions of two data frame with repertoires or a list with data frames) are provided to estimate distance among segments usage of different repertoires. To visualise distances *tcR* employed the `vis.radarlike` function, see Section "Plots" for more detailed information.

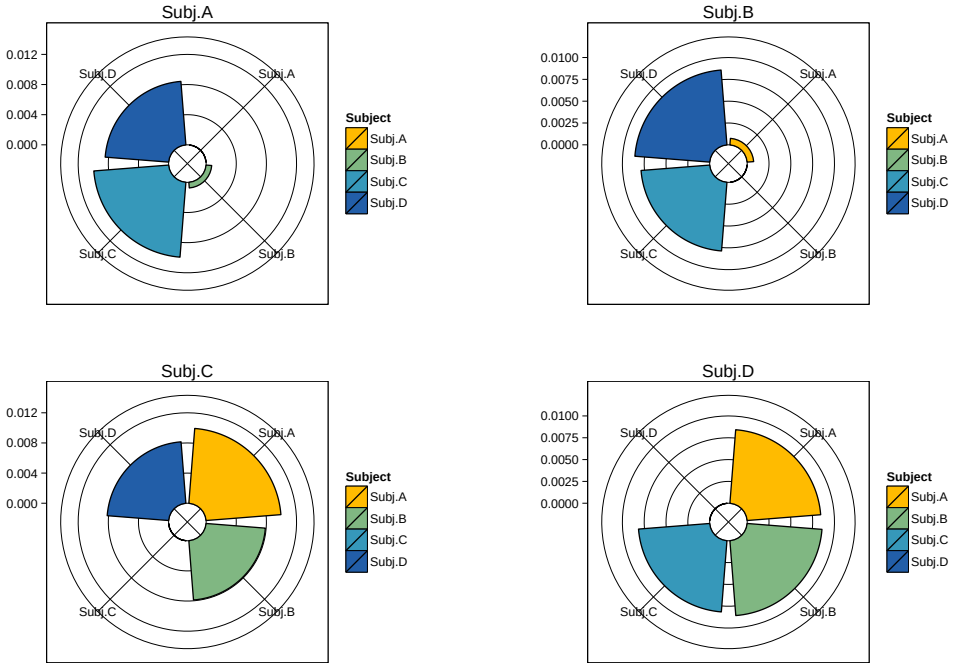
```
>                               # Transform "0:100" to distribution with Laplace correction
> entropy(0:100, .laplace = 1)  # (i.e., add "1" to every value before transformation).
```

```
[1] 6.386523
```

```
> entropy.seg(twb)  # Compute entropy of V-segment usage for each data frame. Same to
```

```
  Subj.A   Subj.B   Subj.C   Subj.D
4.807162 4.867361 4.718884 4.676153
```

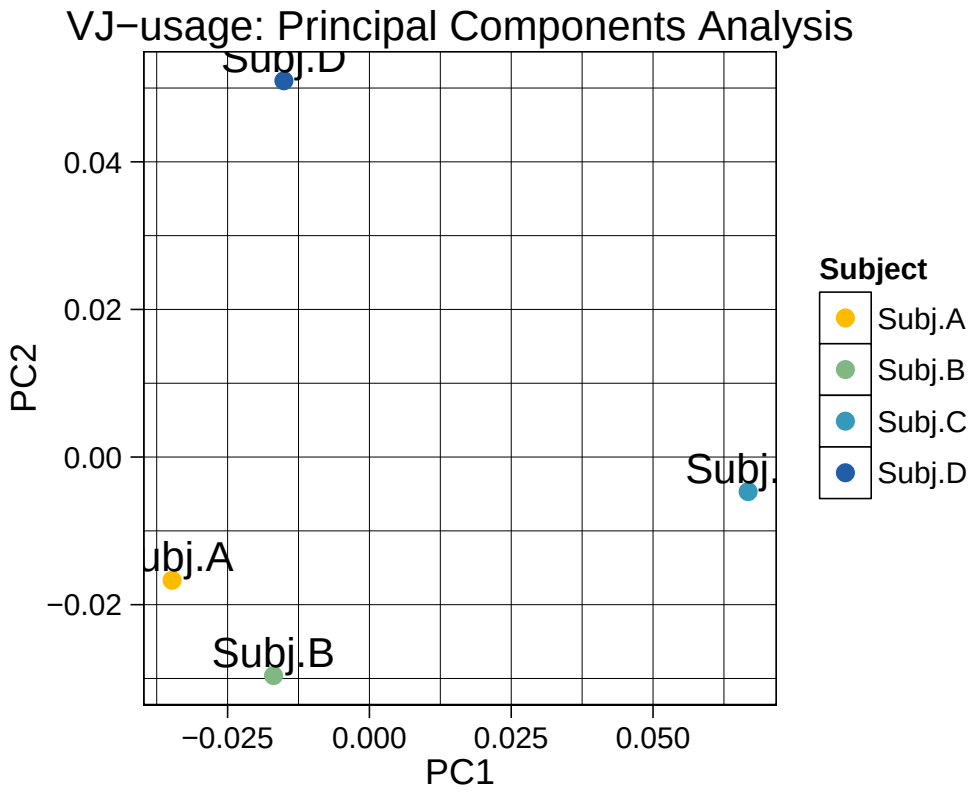
```
>                     # apply(freq.Vb(twb)[,-1], 2, entropy)
> # Next expression is equivalent to the expression
> # js.div(freq.Vb(twb[[1]])[,2], freq.Vb(twb[[2]])[,2], .norm.entropy = T)
> js.div.seg(twb[[1]], twb[[2]], .verbose = F)

[1] 0.0007516101

> # Also works when input arguments are list of data frames.
> imm.js <- js.div.seg(twb, .verbose = F)
> vis.radarlike(imm.js, .ncol = 2)
```



4.2 Principal Component Analysis (PCA)

Principal component analysis (PCA) is a statistical procedure for transforming a set of observations to a set of special values for analysis. In *tcR* implemented functions `pca.segments` for performing PCA on V- or J-usage, and `pca.segments.2D` for performing PCA on VJ-usage. For plotting the PCA results see the `vis.pca` function.

```
> pca.segments(twb)                       # Plot PCA results of V-segment usage.
> class(pca.segments(twb, .do.plot = F))  # Return object of class "prcomp"

[1] "prcomp"
```



5 Shared repertoire

To investigate a shared among a several repertoires clones (or so-called "shared repertoire") the package provided the `shared.repertoire` function along with functions for computing the shared repertoire statistics. The `shared.representation` function computes the number of shared clones for each repertoire for each degree of sharing (i.e., number of people, in which indicated amount of clones have been found). The function `shared.summary` is equivalent to `intersection` but on the shared repertoire. Measuring distances among repertoires using the cosine similarity on vector of counts of shared sequences is also possible with the `cosine.sharing` function.

```
> # Compute shared repertoire of amino acid CDR3 sequences and V-segments
> # which has been found in two or more people.
> imm.shared <- shared.repertoire(.data = twb, .type = 'avc', .min.ppl = 2, .verbose = F)
> head(imm.shared)
```

```
   CDR3.amino.acid.sequence V.segments People Subj.A Subj.B Subj.C Subj.D
1:            CASSDRDTGELFF    TRBV6-4      4    113    411    176   2398
2:          CASSDSSGGYNEQFF    TRBV6-4      4     68    357     31    115
3:           CASSFLSGTDTQYF     TRBV28      4     36    111     59    203
4:           CASSGQGNTEAFF       TRBV2      4    223    252     69    152
5:          CASSLGQGGQPQHF    TRBV7-9      4     34    139     31     84
6:           CASKGQLNTEAFF     TRBV19      3    125     NA     37     34
```

```
> shared.representation(imm.shared)  # Number of shared sequences.
```

```
  Subj.A Subj.B Subj.C Subj.D
1      0      0      0      0
2    219    205    192    170
```

```
3     22     19     20     23
4      5      5      5      5
```

```
> cosine.sharing(imm.shared)         # Compute cosing similarity on shared sequences.
```

```
             [,1]         [,2]         [,3]         [,4]
[1,]           NA 1.457794e-04 5.398229e-05 5.554715e-05
[2,] 1.457794e-04           NA 4.956112e-05 5.058172e-05
[3,] 5.398229e-05 4.956112e-05           NA 1.511286e-04
[4,] 5.554715e-05 5.058172e-05 1.511286e-04           NA
```

```
> # It seems like repetoires are clustering in three groups: (1,2), (3,4) and (5,6).
```

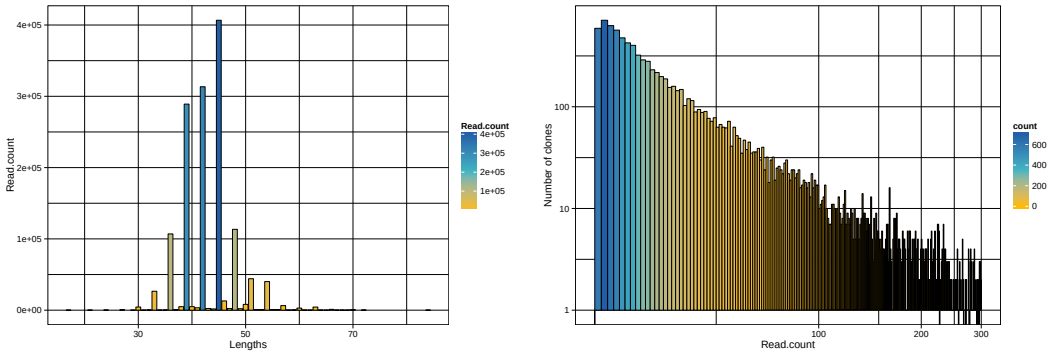
6 Plots

The package implements rich data visualisation procedures. All of them are described in this chapter, for detailed examples see related Sections.

6.1 CDR3 length and read count distributions

Plots of the distribution of CDR3 nucleotide sequences length (function `vis.count.len`) and the histogram of "Read.count" number (function `vis.number.count`). Input data is either a data frame or a list with data frames.

```
> p1 <- vis.count.len(twb[[1]])
> p2 <- vis.number.count(twb[[1]])
> grid.arrange(p1, p2, ncol = 2)
```



6.2 Head proportions plot

For visualisation of proportions of the most abundant clones in a repertoire *tcR* offers the `vis.top.proportions` function. As input it's receives either data frame or a list with data frames and an integer vector with number of clones for computing proportions of count for this clones. See Subsection 2.2 for examples.

6.3 Visualisation of distances: heatmap and radar-like plot

Pairwise distances can be represented as qudratic matrices or data frames, where every row and column represented a repertoire, and a value in every cell (i, j) is a distance between repertoires with indices i and j. For plotting quadratic matrices or data frames in *tcR* implemented functions `vis.heatmap` and `vis.radarlike`. See Subsection 3.1 and 3.4 for examples of set intersections procedures, and Subsection 4.1 for distance computing subroutines using methods from Information Theory.

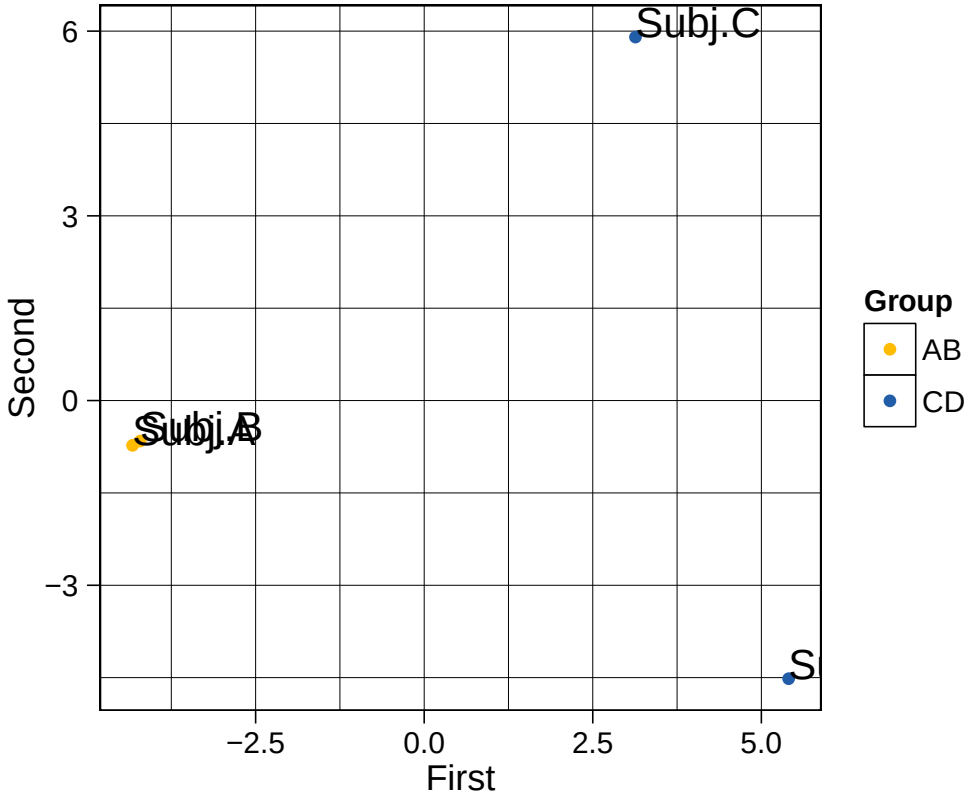
6.4 Segments usage

For visualisation of segments usage *tcR* employes subroutines for making classical histograms using functions `vis.V.usage` and `vis.J.usage`. Functions accept data frames as well as a list of data frames. Data frames could be a repertoire data or data from the `freq.segments` function. Using a parameter `.dodge`, user can change output between histograms for each data frame in the given list (`.dodge` == FALSE) or one histogram for all data, which is very useful for comparing distribution of segments (`.dodge` == TRUE). See Subsection 2.4 for examples.

6.5 PCA

For quick plotting of results from the `prcomp` function (i.e., objects of class `prcomp`), *tcR* provides the `vis.pca` function. Input argument for it is an object of class `prcomp` and a list of groups (vectors of indices) for colour points:

```
> imm.pca <- pca.segments(twb, scale. = T, .do.plot = F)
> vis.pca(imm.pca, list(AB = c(1,2), CD = c(3,4)))
```



7 Conclusion

Feel free to contact us for the package-related or immunoinformatics research-related questions.

8 Appendix A: Kmers retrieving

The *tcR* package implements functions for working with k-mers. Function `get.kmers` generates k-mers from the given chatacter vector or a data frame with columns for sequences and a count for each sequence.

```
> head(get.kmers(twb[[1]]$CDR3.amino.acid.sequence, 100, .meat = F, .verbose = F))
```

```
  Kmers Count
1 CASSL    20
2 CASSP    12
3 ASSLG    11
4 CASSY    11
5 NEQFF    11
6 YEQYF    11
```

```
> head(get.kmers(twb[[1]], .meat = T, .verbose = F))
```

```
  Kmers  Count
1 CASSL 283192
2 DTQYF 217783
3 NEQFF 179230
4 CASSQ 158877
5 ASSLG 154560
6 YEQYF 148602
```

9 Appendix B: Nucleotide and amino acid sequences manipulation

The *tcR* package also provides a several number of quick functions for performing classic bioinformatics tasks on strings. For more powerful subroutines see the Bioconductor's *Biostrings* package.

9.1 Nucleotide sequence manipulation

Functions for basic nucleotide sequences manipulations: reverse-complement, translation and GC-content computation. All functions are vectorised.

```
> revcomp(c('AAATTT', 'ACGTTTGGA'))
```

```
[1] "AAATTT"    "TCCAAACGT"
```

```
> cbind(bunch.translate(twb[[1]]$CDR3.nucleotide.sequence[1:10]), twb[[1]]$CDR3.amino.acid.sequence[1:10])
```

```
      [,1]              [,2]
 [1,] "CASSQALAGADTQYF" "CASSQALAGADTQYF"
 [2,] "CASSLGPRNTGELFF" "CASSLGPRNTGELFF"
 [3,] "CASSYGGAADTQYF"  "CASSYGGAADTQYF"
 [4,] "CSAGGIETSYNEQFF" "CSAGGIETSYNEQFF"
 [5,] "CASSPILGEQFF"    "CASSPILGEQFF"
 [6,] "CASKKDRDYGYTF"   "CASKKDRDYGYTF"
 [7,] "CASSQQGSGNTIYF"  "CASSQQGSGNTIYF"
 [8,] "CASSLGLHYEQYF"   "CASSLGLHYEQYF"
 [9,] "CASSRASSYNSPLHF" "CASSRASSYNSPLHF"
[10,] "CASSYLGPDDTEAFF" "CASSYLGPDDTEAFF"
```

```
> gc.content(twb[[1]]$CDR3.nucleotide.sequence[1:10])
```

```
 [1] 0.5333333 0.5777778 0.5238095 0.4888889 0.5555556 0.4871795 0.4523810
 [8] 0.4871795 0.5555556 0.5333333
```

9.2 Reverse translation subroutines

Function `codon.variants` returns a list of vectors of nucleotide codons for each letter for each input amino acid sequence. Function `translated.nucl.sequences` returns the number of nucleotide sequences, which, when translated, will result in the given amino acid sequence(s). Function `reverse.translation` return all nucleotide

sequences, which is translated to the given amino acid sequences. Optional argument `.nucseq` for each of this function provides restriction for nucleotides, which cannot be changed. All functions are vectorised.

```
> codon.variants('LQ')

[[1]]
[[1]][[1]]
[1] "CTA" "CTC" "CTG" "CTT" "TTA" "TTG"

[[1]][[2]]
[1] "CAA" "CAG"

> translated.nucl.sequences(c('LQ', 'CASSLQ'))

[1]   12 3456

> reverse.translation('LQ')

 [1] "CTACAA" "CTCCAA" "CTGCAA" "CTTCAA" "TTACAA" "TTGCAA" "CTACAG" "CTCCAG"
 [9] "CTGCAG" "CTTCAG" "TTACAG" "TTGCAG"

> translated.nucl.sequences('LQ', 'XXXXXG')

[1] 6

> codon.variants('LQ', 'XXXXXG')

[[1]]
[[1]][[1]]
[1] "CTA" "CTC" "CTG" "CTT" "TTA" "TTG"

[[1]][[2]]
[1] "CAG"

> reverse.translation('LQ', 'XXXXXG')

[1] "CTACAG" "CTCCAG" "CTGCAG" "CTTCAG" "TTACAG" "TTGCAG"
```