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 (e.g., T-cell or B-cell recetpors). 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

1.1 Overview and vignette structure

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

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

For analysis of a single repertoire, use the pipeline file:

```
<path to the tcR package>/inst/mono.pipeline.Rmd
```

For analysis of a group of repertoires ("cross-analysis"), use the pipeline file:

```
<path to the tcR package>/inst/batch.pipeline.Rmd
```

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
```

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

MiTCR is a tool for retrieving TCR CDR1-2-3 sequences from NGS data:

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

Shortly, to start MiTCR on a specific files you need to do:

```
java -Xmx8g -jar mitcr.jar -pset flex -level 2 ~/data/raw/TwA1_B.fastq.gz ~/data/mitcr/TwA1_B.txt
```

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/immdata1.txt".
> immdata1 <- parse.file("~/data/immdata1.txt")
> # Parse files "~/data/immdata1.txt" and "~/data/immdat2.txt".
> immdata12 <- parse.file.list(c("~/data/immdata1.txt", "~/data/immdata2.txt"))
> # Parse all files in "~/data/".
> immdata <- parse.folder("~/data/")
```

1.4 Structure of a MiTCR data frame (clonesets representation)

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
1             0                2
2             2                2
3             4                6
4             7                7
5             0                6
6             0                5
```

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 will be suitable for using with the package.

2 Repertoire descriptive statistics

To describe a repertoire, we need to compute a number of informative statistics.

2.1 Sequences summary

To get a general view of CDR3 sequences counts (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:

```
> library(tcR)
> mitcr.stats(immdata)
```

```
                           TwA1_B       TwA2_B      TwC1_B      TwC2_B
#Nucleotide clones      10000.0000   10000.0000  10000.0000 1.00000e+04
#Aminoacid clonotypes    9850.0000    9838.0000   9775.0000 9.87200e+03
%Aminoacid clonotypes       0.9850       0.9838      0.9775 9.87200e-01
#In-frames               9654.0000    9600.0000   9808.0000 9.28800e+03
%In-frames                  0.9654       0.9600      0.9808 9.28800e-01
#Out-of-frames            346.0000     400.0000    192.0000 7.12000e+02
%Out-of-frames              0.0346       0.0400      0.0192 7.12000e-02
Sum.Read.count        1410263.0000 2251408.0000 969949.0000 1.41913e+06
Min.Read.count             22.0000      20.0000     23.0000 3.20000e+01
1st Qu.Read.count          26.0000      24.0000     28.0000 3.70000e+01
Median.Read.count          33.0000      31.0000     39.0000 4.80000e+01
Mean.Read.count           141.0000     225.1000     96.9900 1.41900e+02
3rd Qu.Read.count          57.0000      55.0000     68.0000 8.30000e+01
Max.Read.count          81520.0000  171200.0000 104600.0000 3.35900e+04
                           TwD1_B       TwD2_B
#Nucleotide clones     10000.0000   10000.0000
#Aminoacid clonotypes   9815.0000    9783.0000
%Aminoacid clonotypes      0.9815       0.9783
#In-frames              9773.0000    9816.0000
%In-frames                 0.9773       0.9816
#Out-of-frames           227.0000     184.0000
%Out-of-frames             0.0227       0.0184
Sum.Read.count        802995.0000 1257855.0000
Min.Read.count            21.0000      20.0000
1st Qu.Read.count         24.0000      25.0000
Median.Read.count         30.0000      34.0000
Mean.Read.count           80.3000     125.8000
3rd Qu.Read.count         51.0000      63.0000
Max.Read.count         44710.0000  178200.0000
```

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 approximately the 25% of the sum of values in "Read.count":

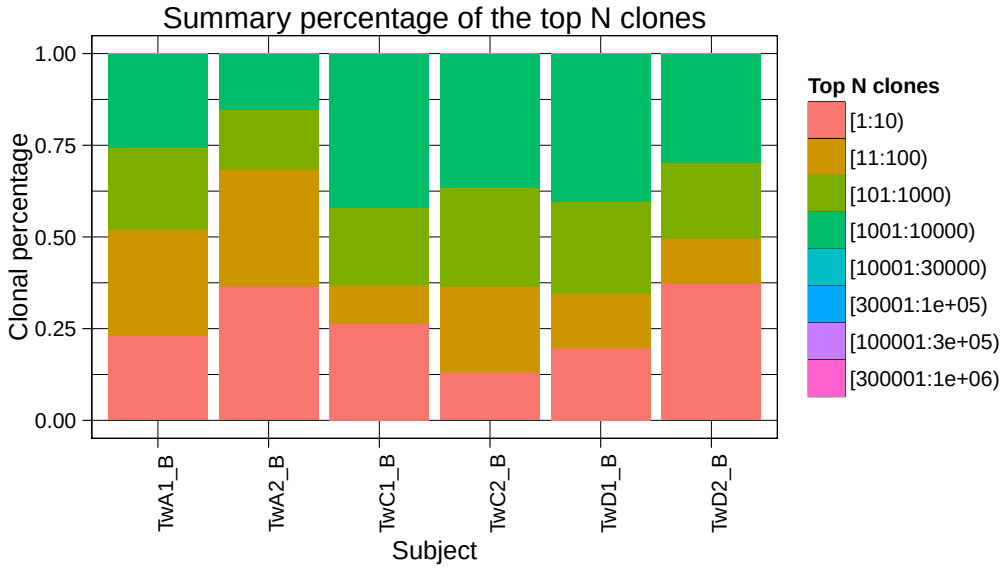
```
>                                # How many clones fill up approximately
> clonal.proportion(immdata, 25)  # the 25% of the sum of values in 'Read.count'?
```

```
              TwA1_B  TwA2_B  TwC1_B  TwC2_B  TwD1_B  TwD2_B
Clones       12.0000  6.0000  7.0000 38.0000 27.0000  3.0000
Percentage   25.1000 26.5000 25.2000 25.2000 25.0000 26.8000
Overall.prop  0.0012  0.0006  0.0007  0.0038  0.0027  0.0003
```

To get a proportion of sum of reads of top clones to the overall sum of reads, 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 proportion of the top-10 clones' reads
> top.proportion(immdata, 10)     # to the overall number of reads?

   TwA1_B    TwA2_B    TwC1_B    TwC2_B    TwD1_B    TwD2_B
0.2289069 0.3648699 0.2620158 0.1305398 0.1944109 0.3733085

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



Function `split.proporion` with two arguments `.col` and `.bound` gets subset of the given data frame with reads, which have column's `.col` 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 proportion of sequences which
>                                  # has 'Read.count' <= 100 to the
> tailbound.proportion(immdata, 100)  # overall number of reads?

TwA1_B TwA2_B TwC1_B TwC2_B TwD1_B TwD2_B
0.8651 0.8641 0.8555 0.8020 0.8900 0.8544
```

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.*frames`, `get.*frames`. 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(immdata) # Return all in-frame sequences from the 'immdata'.
>                                  # Count the number of out-of-frame sequences
> count.outframes(immdata, 5000)  # from the first 5000 sequences.

TwA1_B TwA2_B TwC1_B TwC2_B TwD1_B TwD2_B
   172    212     73    326    108     82
```

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

[1] 10000

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

TwA1_B TwA2_B TwC1_B TwC2_B TwD1_B TwD2_B
   172    212     73    326    108     82
```

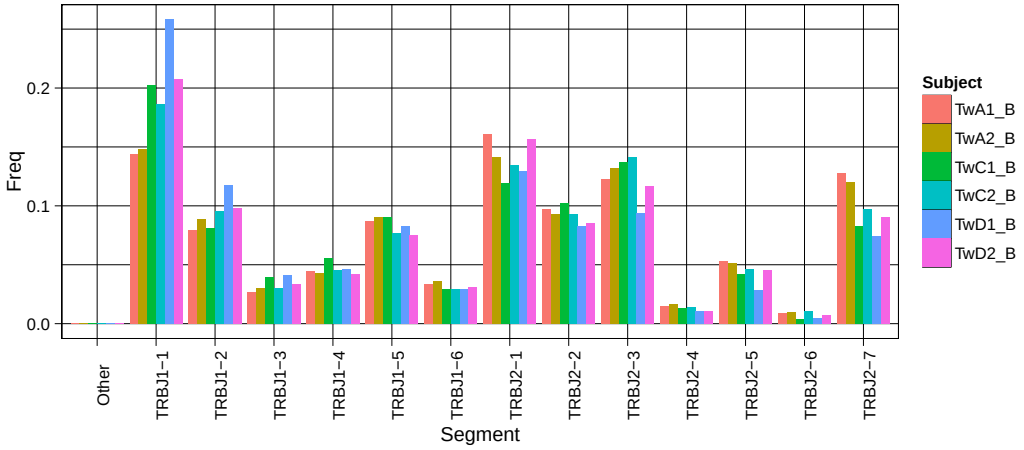
2.4 Segments statistics

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 about plots. V and J alphabets are store in the .rda file "human.alphabets.rda". All of mentioned functions are accepts data frames as well as list of data frames. Output for this functions are data frames with the first column stands for segment and other or others for frequencies.

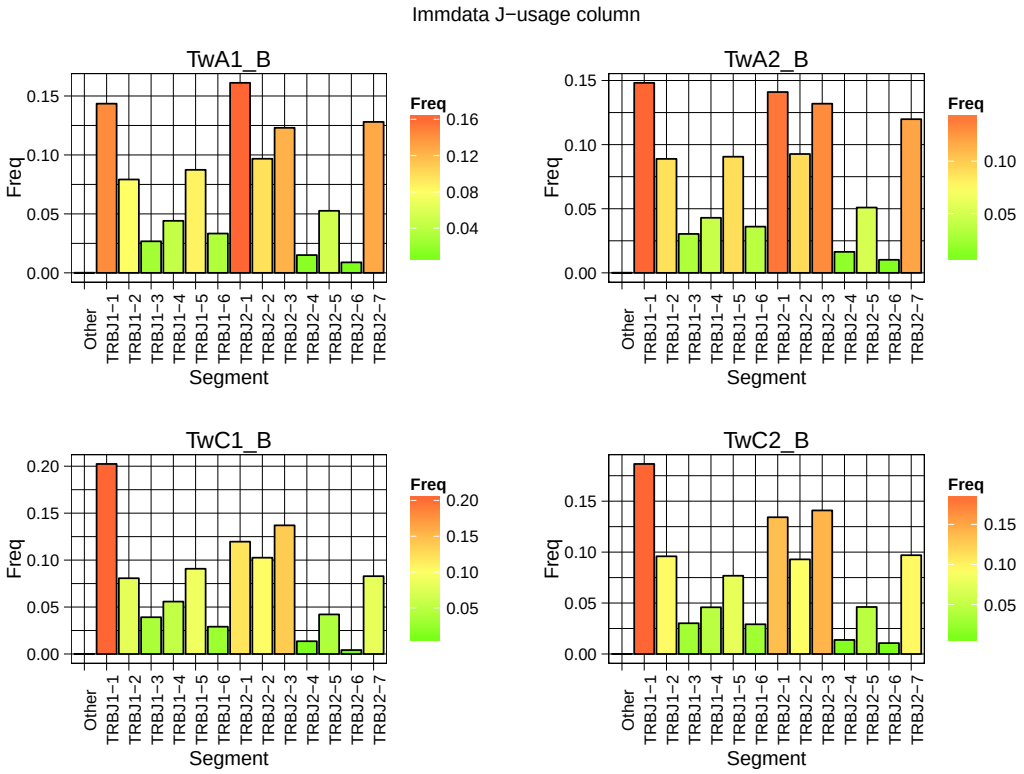
```
> data(human.alphabets)
> imm1.vs <- freq.segments(immdata[[1]])  # Equivalent to freq.Vb(immdata[[1]])
> head(imm1.vs)

       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(immdata)  # Equivalent to freq.Vb(immdata)
> imm.vs.all[1:10, 1:4]

              Segment      TwA1_B      TwA2_B       TwC1_B
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(immdata[[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 visualise segments usage with functions `vis.V.usage` and `vis.J.usage`:

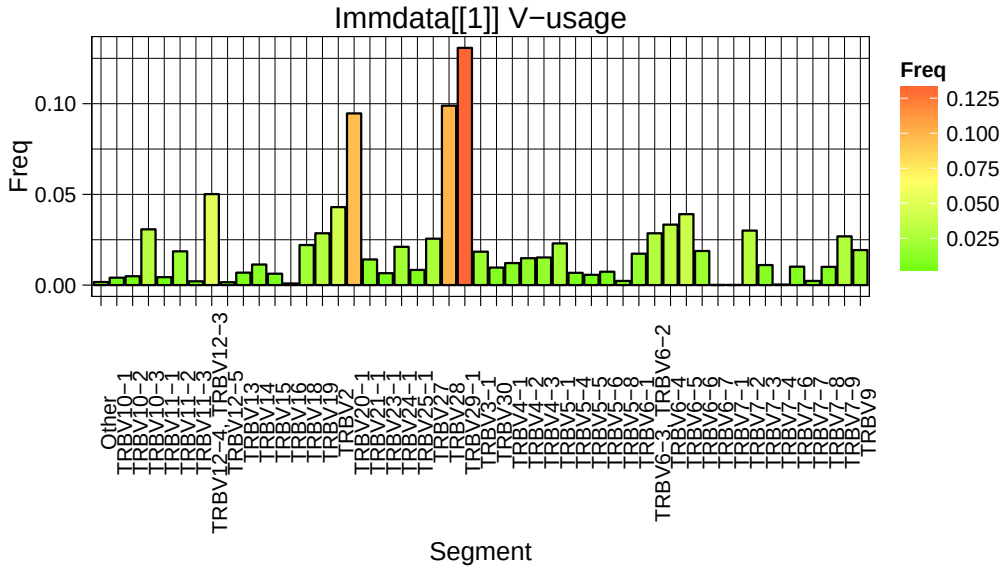
```
> # Put ".dodge = F" to get distinct plot for every data frame in the given list.
> library(gridExtra)
> library(ggplot2)
> library(reshape2)
> vis.J.usage(immdata, .cast.freq = T, .main = 'Immdata J-usage dodge', .dodge = T)
```




```
> vis.J.usage(immdata[1:4], .cast.freq = T, .main = 'Immdata J-usage column', .dodge = F, .ncol = 2)
```




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



2.5 Search for a target CDR3 sequences

For exact or fuzzy search of sequences the package employed the function `find.clonotypes`. Input arguments for this function are data frame or list of data frames, targets (character vector or data frame with column for sequences and additional columns like V-segments), value of which column or columns return, method which will be used in comparison of 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 columns name for getting sequences for matching from the given data. Sounds very complex, but in practice it's very easy, therefore let's go to examples. Suppose we want to search for a 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 using V-segments. 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.

```
> immdata <- set.rank(immdata)
> cmv.imm.ex <-
+   find.clonotypes(immdata[1:2], cmv[,1], 'exact',
+                   c('Read.count', 'Total.insertions'),
+                   .verbose = F)
> head(cmv.imm.ex)
```

```
                   CDR3.amino.acid.sequence Read.count.TwA1_B Read.count.TwA2_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.TwA1_B Total.insertions.TwA2_B
CASSALGGAGTGELFF                          9                      10
CASSALGGAGTGELFF.1                       NA                       9
CASSLTGNTEAFF                             2                       2
CASSLTGNTEAFF.1                           1                       0
CASSLTGNTEAFF.2                          NA                       1
CASSSANYGYTF                             NA                       1
```

```
> cmv.imm.hamm.v <-
+   find.clonotypes(immdata[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.TwA1_B Rank.TwA2_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
                TwC1_B.Rank
CAQVLLIETQYF             NA
CASAGLDLFVTGELFF      7532.5
CASALQAYYNEQFF            NA
CASCDDYNSPLHF         7190.5
CASEDRGRTDTQYF        9729.5
CASGGSLGQNTEAFF        737.5
```

```
> cmv.imm.lev.v <-
+   find.clonotypes(immdata[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.TwA1_B
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.TwA2_B TwC1_B.Read.count
CASSALGGAGTGELFF                NA                NA
CASSLIGVSSYNEQFF                NA                NA
CASSLTGNTEAFF                   NA                NA
CASSSANYGYTF                    NA                NA
CSVGRAQNEQFF                    NA                NA
```

3 Analysis of sets and distributions of receptors

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 theirs intersection set. *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(immdata[[1]]$CDR3.nucleotide.sequence,
> #                         immdata[[2]]$CDR3.nucleotide.sequence)
> # or intersectCount(immdata[[1]]$CDR3.nucleotide.sequence,
> #                   immdata[[2]]$CDR3.nucleotide.sequence)
> # First "n" stands for a "CDR3.nucleotide.sequence" column, "e" for exact match.
> intersect(immdata[[1]], immdata[[2]], 'n0e')
```

```
[1] 46
```

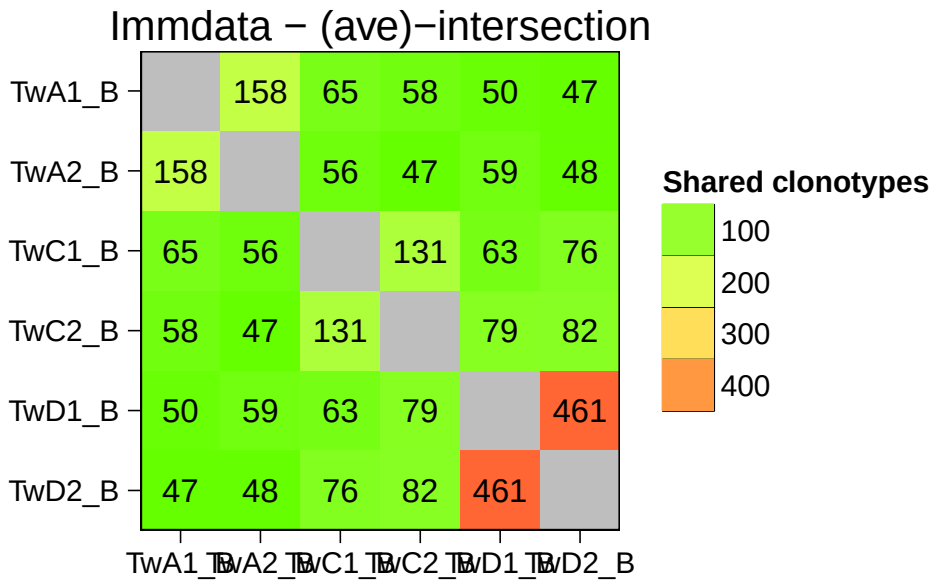
```
> # First "a" stands for "CDR3.amino.acid.sequence" column.
> # Second "v" means that intersect should also use the "V.segments" column.
> intersect(immdata[[1]], immdata[[2]], 'ave')
```

```
[1] 158
```

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

```
       TwA1_B TwA2_B TwC1_B TwC2_B TwD1_B TwD2_B
TwA1_B     NA    158     65     58     50     47
TwA2_B    158     NA     56     47     59     48
TwC1_B     65     56     NA    131     63     76
TwC2_B     58     47    131     NA     79     82
TwD1_B     50     59     63     79     NA    461
TwD2_B     47     48     76     82    461     NA
```

```
> # Plot results.
> vis.heatmap(intersect(immdata, 'ave'), .title = 'Immdata - (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 match. They all have parameter `.col` which specifies names of columns which will used in computing intersection. Function `intersectCount` returns number of similar elements; `intersectIndices(x, y)` returns 2-row 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 an 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 elements which are in both immdata[[1]] and immdata[[2]].
> # Elements are tuples of CDR3 nucleotide sequence and corresponding V-segment
> imm.1.2 <- intersectLogic(immdata[[1]], immdata[[2]],
+                           .col = c('CDR3.amino.acid.sequence', 'V.segments'))
> head(immdata[[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. This function works as follows: sequentially apply the `intersect` function to the top X clones, where X is a vector of integers, e.g. `seq(1000, 100000, 1000)`.

```
> immdata.top <- top.cross(immdata)
> top.cross.plot(immdata.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 inverse probability of choosing two similar clones. Function `gini` computes the Gini index of clonal distribution.

```
> sapply(immdata, function (x) diversity(x$Read.count))       # Evaluate the diversity of clones by the ecologi

  TwA1_B   TwA2_B   TwC1_B   TwC2_B   TwD1_B   TwD2_B
34.86013 24.10521 16.07719 98.89450 35.74310 11.47269

> sapply(immdata, function (x) inverse.simpson(x$Read.count)) # Compute the diversity as inverse probability of

   TwA1_B    TwA2_B    TwC1_B    TwC2_B    TwD1_B    TwD2_B
119.28802  56.59197  56.45036 359.01535 152.84909  32.15712

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

   TwA1_B    TwA2_B    TwC1_B    TwC2_B    TwD1_B    TwD2_B
0.7556390 0.8517935 0.6141982 0.6561231 0.6160872 0.7267838
```

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

3.4 More complicated set 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(immdata, function (x,y) morisitas.index(x[, cols], y[, cols]), .verbose = F)

             TwA1_B       TwA2_B       TwC1_B       TwC2_B       TwD1_B
TwA1_B           NA 1.240467e-03 5.525470e-05 0.0002881564 0.0003489930
TwA2_B 1.240467e-03           NA 1.017043e-04 0.0003148358 0.0001389210
TwC1_B 5.525470e-05 1.017043e-04           NA 0.0005150483 0.0001887098
TwC2_B 2.881564e-04 3.148358e-04 5.150483e-04           NA 0.0024776509
TwD1_B 3.489930e-04 1.389210e-04 1.887098e-04 0.0024776509           NA
```

```
TwD2_B 7.816552e-05 8.803425e-05 8.392898e-05 0.0002999521 0.0013253377
            TwD2_B
TwA1_B 7.816552e-05
TwA2_B 8.803425e-05
TwC1_B 8.392898e-05
TwC2_B 2.999521e-04
TwD1_B 1.325338e-03
TwD2_B           NA
```

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

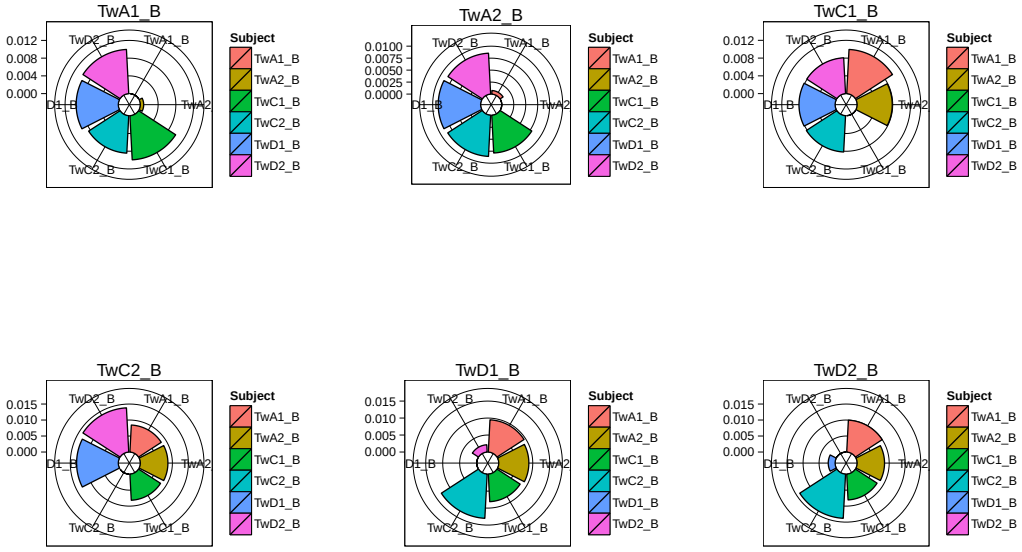
4 Analysis of segments usage

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) is applicable 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(immdata)  # Compute entropy of V-segment usage for each data frame. Same to

  TwA1_B   TwA2_B   TwC1_B   TwC2_B   TwD1_B   TwD2_B
4.807162 4.867361 4.718884 4.676153 4.712981 4.779432

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

[1] 0.0007516101

> imm.js <- js.div.seg(immdata, .verbose = F)  # Also works when input arguments are two data frames.
> vis.radarlike(imm.js)
```

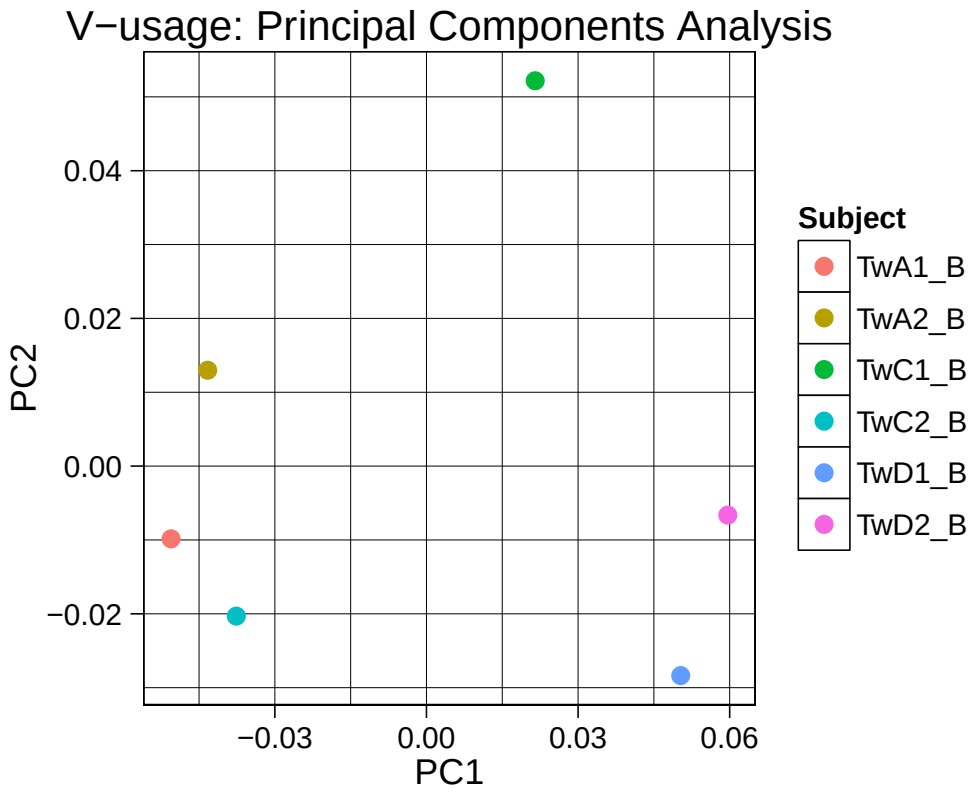


4.2 PCA

Principal component analysis (PCA) is a way to transform the data to less dimensional representation. In our package 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. For PCA of shared sequences see the next Section "Shared repertoire of sequences" (function `shared.seq.pca`).

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

```
[1] "sdev"     "rotation" "center"   "scale"    "x"
```



5 Shared repertoire of sequences

To investiage a shared repertoire of sequence the package provided the `shared.repertoire` function along with functions for computing the shared repertoire statistics. The `shared.representation` function computes number of shared clones for each repertoire for each degree of sharing (i.e., number of people, in which some clone has 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 shared amino acid CDR3 sequences and V-segments
> # which has been found in two or more people.
> imm.shared <- shared.repertoire(immdata, 'av', 2)
```

```
Aggregating sequences...

  |
  |                                                                      |   0%
  |
  |============                                                          |  17%
  |
  |========================                                              |  33%
  |
  |====================================                                  |  50%
  |
  |================================================                      |  67%
  |
```

```
  |=============================================================          |  83%
  |
  |======================================================================| 100%
Merging data tables...

  |
  |                                                                      |   0%
  |
  |==============                                                        |  20%
  |
  |============================                                          |  40%
  |
  |==========================================                            |  60%
  |
  |========================================================              |  80%
  |
  |======================================================================| 100%
```

```
> head(imm.shared)
```

```
   CDR3.amino.acid.sequence V.segments People TwA1_B TwA2_B TwC1_B TwC2_B
1:            CASSDRDTGELFF    TRBV6-4      5    113    411    176   2398
2:          CASSDSSGSTDTQYF    TRBV6-4      5    552    184     NA    128
3:         CASSDSTSGGADTQYF    TRBV6-4      5     69     NA    373   6257
4:           CASSERGGTDTQYF    TRBV6-4      5     44     80     NA     64
5:           CASSFLSGTDTQYF     TRBV28      5     36    111     59    203
6:            CASSGQGNTEAFF      TRBV2      5    223    252     69    152
   TwD1_B TwD2_B
1:    428     NA
2:    439   1184
3:   1256    729
4:    507    119
5:    161     NA
6:     NA     65
```

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

```
  TwA1_B TwA2_B TwC1_B TwC2_B TwD1_B TwD2_B
1      0      0      0      0      0      0
2    234    228    238    204    500    492
3     42     39     44     60     58     63
4      9     12     12     12     17     18
5      7      5      5      7      6      5
6      0      0      0      0      0      0
```

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

```
             [,1]         [,2]         [,3]         [,4]         [,5]
[1,]           NA 1.030843e-04 3.532319e-05 3.402201e-05 1.377858e-05
[2,] 1.030843e-04           NA 3.084526e-05 2.946663e-05 1.576509e-05
[3,] 3.532319e-05 3.084526e-05           NA 8.146968e-05 1.709315e-05
[4,] 3.402201e-05 2.946663e-05 8.146968e-05           NA 2.390578e-05
[5,] 1.377858e-05 1.576509e-05 1.709315e-05 2.390578e-05           NA
[6,] 1.369473e-05 1.347613e-05 2.073486e-05 2.366020e-05 6.642023e-05
             [,6]
[1,] 1.369473e-05
[2,] 1.347613e-05
[3,] 2.073486e-05
[4,] 2.366020e-05
```

```
[5,] 6.642023e-05
[6,]           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 is described in this chapter, for detailed examples see related Sections.

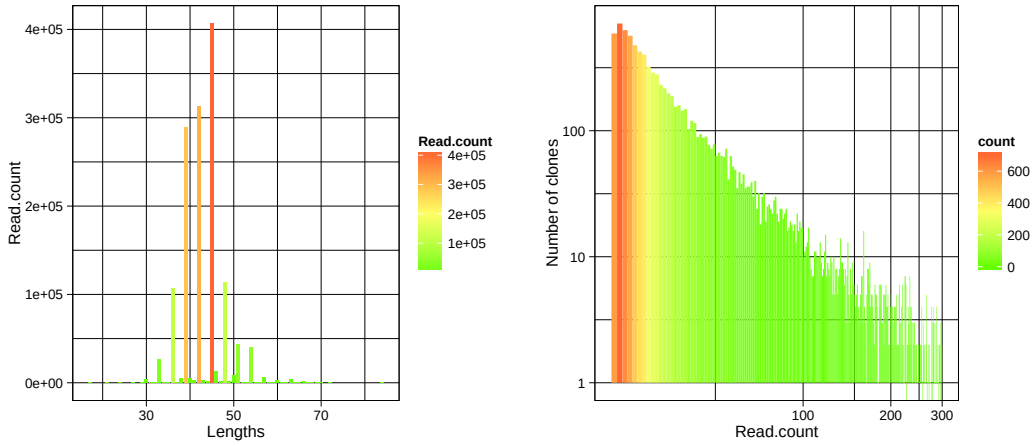
6.1 Length and read count distributions

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

```
> # library(ggplot2)
> # library(gridExtra)
> p1 <- vis.count.len(immdata[[1]])
> p2 <- vis.number.count(immdata[[1]])
```

```
Limits for x-axis set to (0,50). Transform y-axis to sqrt(y).
```

```
> 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 Grid plot and radar-like plot: visualisation of distances

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 visualising 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.

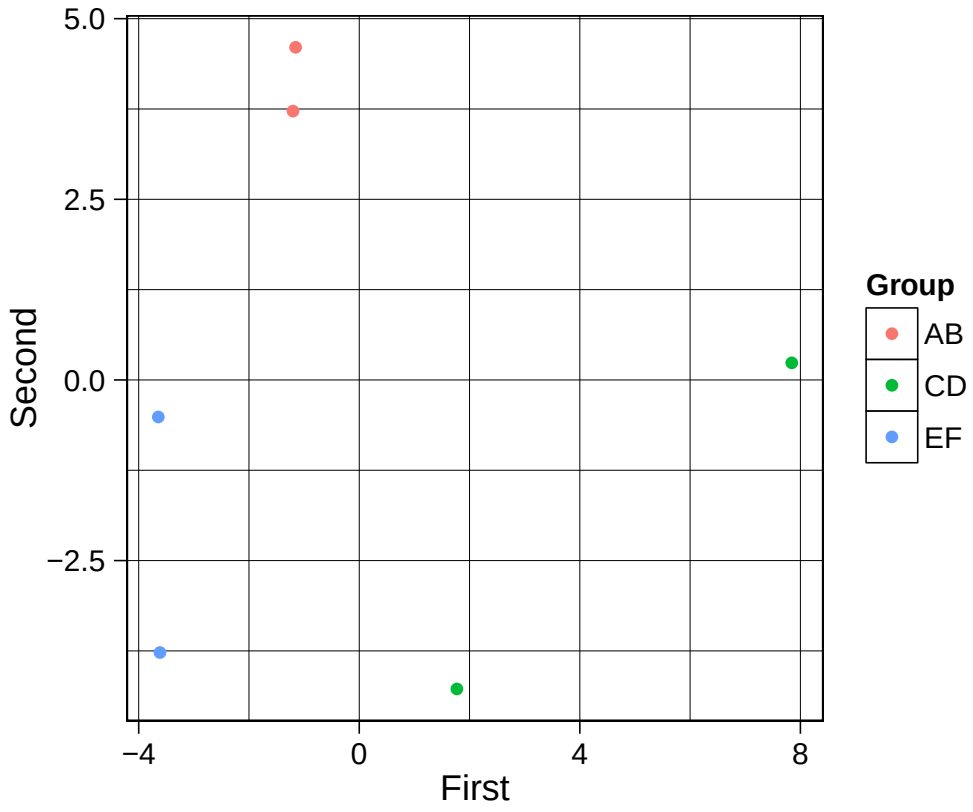
6.5 Spectratyping

One of the most popular visualisation approach is spectratyping. `spectratyping`

6.6 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(immdata, scale. = T, .do.plot = F)
> vis.pca(imm.pca, list(AB = c(1,2), CD = c(3,4), EF = c(5,6)))
```



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(immdata[[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(immdata[[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 few 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(immdata[[1]]$CDR3.nucleotide.sequence[1:10]), immdata[[1]]$CDR3.amino.acid.sequence[1:1
```

```
      [,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(immdata[[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 number of nucleotide sequences, which translated to 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('ACT')

[[1]]
[[1]][[1]]
[1] "GCA" "GCC" "GCG" "GCT"

[[1]][[2]]
[1] "TGC" "TGT"

[[1]][[3]]
[1] "ACA" "ACC" "ACG" "ACT"

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

[1]   32 3456

> reverse.translation('ACT')

 [1] "GCATGCACA" "GCCTGCACA" "GCGTGCACA" "GCTTGCACA" "GCATGTACA" "GCCTGTACA"
 [7] "GCGTGTACA" "GCTTGTACA" "GCATGCACC" "GCCTGCACC" "GCGTGCACC" "GCTTGCACC"
[13] "GCATGTACC" "GCCTGTACC" "GCGTGTACC" "GCTTGTACC" "GCATGCACG" "GCCTGCACG"
[19] "GCGTGCACG" "GCTTGCACG" "GCATGTACG" "GCCTGTACG" "GCGTGTACG" "GCTTGTACG"
[25] "GCATGCACT" "GCCTGCACT" "GCGTGCACT" "GCTTGCACT" "GCATGTACT" "GCCTGTACT"
[31] "GCGTGTACT" "GCTTGTACT"

> translated.nucl.sequences('ACT', 'XXXXXXXC')

[1] 8

> codon.variants('ACT', 'XXXXXXXC')

[[1]]
[[1]][[1]]
[1] "GCA" "GCC" "GCG" "GCT"

[[1]][[2]]
[1] "TGC" "TGT"

> reverse.translation('ACT', 'XXXXXXXC')

[1] "GCATGC" "GCCTGC" "GCGTGC" "GCTTGC" "GCATGT" "GCCTGT" "GCGTGT" "GCTTGT"
```