GrpString: An R Package for Analysis of Groups of Strings

by Hui Tang, Elizabeth L. Day, Molly B. Atkinson, and Norbert J. Pienta

Abstract The R package **GrpString** was developed as a comprehensive toolkit for quantitatively analyzing and comparing groups of strings. It offers functions for researchers and data analysts to prepare strings from event sequences, extract common patterns from strings, and compare patterns between string vectors. The package also finds transition matrices and complexity of strings, determines clusters in a string vector, and examines the statistical difference between two groups of strings.

Introduction

In domains such as psychology and social science, participants' actions can be listed as sequences of events. These event sequences can be recorded as strings in which a single character represents an event or a state. For example, in an eye-tracking study, the eye gazes on some regions in the order of "Question-Figure-Answer1-Answer2-Answer3-Answer4-Figure-Answer3" can be recorded as a string "QF1234F3" (also called a scanpath in eye-tracking studies). Another example is using the log files of an online learning system to generate event sequences from student actions. Processing and analyzing strings helps researchers explore the features of event sequences. In R, a string is a sequence of characters. It is generated using single or double quotes, and can contain zero or more characters. The R **base** package offers various functions for string operations including character count, case conversion, pattern detection, substring extraction and replacement, and string split and concatenation. These methods are also provided by packages **stringr** (Wickham, 2010, 2017), **stringb** (Meissner, 2016), and **stringi** (Gagolewski and Tartanus, 2017), which are built for the purpose of string manipulation. In addition, other packages, such as **gsubfn** (Grothendieck, 2014) and **uniqtag** (Jackman, 2015), focus on particular functionalities of handling strings. Despite a number of packages for string manipulation, there are few packages designed for the analysis of strings, especially groups of character strings. A common method for string analysis in the existing packages is string comparison by using distances between two strings. For example, the function `adist()` in the R **utils** package calculates a generalized Levenshtein distance between two strings, and the function `stringdist()` in package **stringdist** (van der Loo, 2014, 2016) provides options of computing ten different types of distances.

Broadly speaking, the text of natural language and DNA sequences are also strings. However, these two types of strings differ from simple character strings, which are continuous and generally short (at most several hundred characters in a string). Compared to simple character strings, DNA sequences usually are much longer while natural language text may be segmented by spaces or punctuation. There has been a relatively large collection of packages for processing and analyzing these two complex types of sequences (`https://cran.r-project.org/web/views/NaturalLanguageProcessing.html`; `https://www.bioconductor.org/`). Other packages for manipulating and analyzing event or text sequences include **TraMineR** (Gabadinho et al., 2011, 2017) and **informR** (Marcum and Butts, 2015). On the other hand, there is no package built specifically for analyzing groups of typical character strings from the quantitative perspective. Therefore, a comprehensive R package will fill this gap by providing functions to deal with one or multiple groups of simple character strings, including quantitatively describing and statistically examining differences between groups of strings ("string groups" or "string vectors" are also used in this paper).

Package **GrpString** (Tang and Pienta, 2017) is developed with this purpose, and it emphasizes quantifying the features of a string group as well as the differences between two string groups. First, the package provides functions for the users to convert raw event sequences to strings and then to "collapsed" strings. Next, there are functions that extract common patterns shared by the strings in a vector. The functions return the frequency of each pattern, as well as the number and starting position of each pattern in each string. When patterns from two string vectors are compared, featured patterns for each vector are listed to distinguish the two vectors of strings. The package also contains functions for finding the transition matrices of a single string or a group of strings and for calculating the complexity of strings based on transitions. A transition is a two-character sub-sequence within a string or group of strings. A transition matrix in this package is a two-dimensional matrix that lists the numbers of transitions. Cluster analysis is also included with both hierarchical and k-means methods. Lastly, users can employ a permutation test to statistically compare and visualize the difference between two string vectors. Figure 1 shows the main functions in the current version (v. 0.3.2) and a detailed description for each function follows.

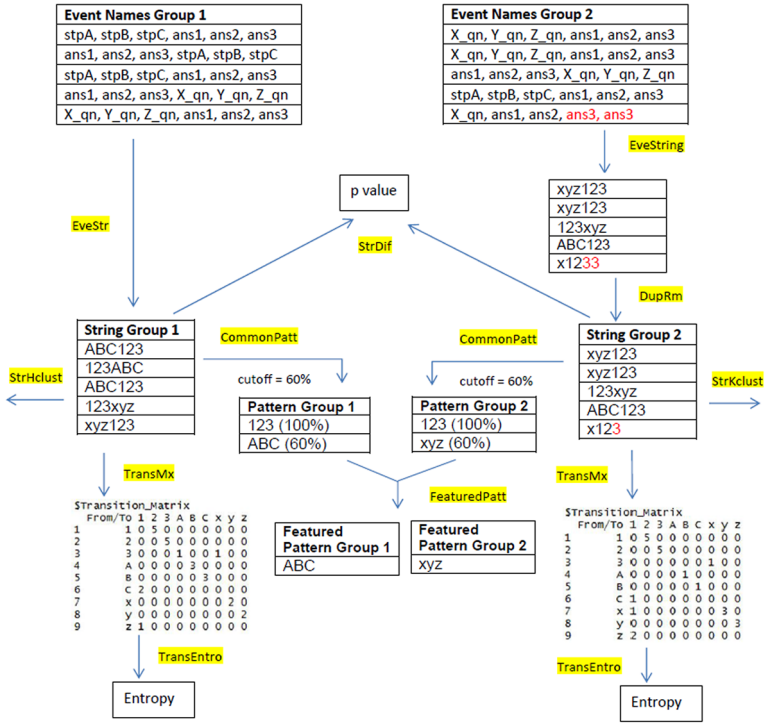

Figure 1: Flow chart of the main functions in R package **GrpString**. There are alternatives for some functions in the flow chart. For example, `EveStr()` can be replaced with `EveString()`, `CommonPatt()` can be replaced with `CommonPattern()`, and `StrHclust()` and `StrKclust()` are interchangeable. The cutoff 60% is chosen as an example showing how to obtain common patterns and then featured patterns. In practice, the users may select different cutoffs when using `CommonPatt()` or `CommonPattern()`.

Functions in GrpString

Converting event sequences to strings

Encoding each event name to its corresponding character is often a necessary step before string analysis. Automatic and simultaneous conversion of event names to string vectors can reduce the users' workload, especially when multiple sequences of event names need to be converted repeatedly. The **GrpString** package offers three functions to accomplish this task, based on a conversion key created by the user. The first function, `EveS()`, converts a sequence of event names to a single string. Here, the two input vectors, `eve.names` and `labels`, which must have the same length, form a conversion key. An element in `labels` can be a letter, digit, or a special character.

```
> event.vec <- c("aoi_1", "aoi_2", "aoi_3", "aoi_2", "aoi_1")
> eve.names <- c("aoi_1", "aoi_2", "aoi_3")
> labels <- c("a", "b", "c")
> EveS(event.vec, eve.names, labels)

[1] "abcba"
```

The second function, `EveStr()`, is applicable when event names are stored in a data frame in which each row has the same number of event names, and hence each converted string has the same length in the exported vector. In this example, `event.name.vec` and `label.vec` form a conversion key.

```
> event.df <- data.frame(c("aoi_1", "aoi_2"),
                         c("aoi_1", "aoi_3"),
                         c("aoi_3", "aoi_5"))
> event.name.vec <- c("aoi_1", "aoi_2", "aoi_3", "aoi_4", "aoi_5")
> label.vec <- c("a", "b", "c", "d", "e")
> EveStr(event.df, event.name.vec, label.vec)
```

```
[1] "aac" "bce"
```

The third function, `EveString()`, is a generalized version of `EveStr()`. It deals with the situation when the user stores event names in a file (e.g., .txt or .csv) in which different rows may have different numbers of elements. It is generally not convenient to read such a file into a data frame. Thus, a .txt or .csv file for event names is used directly in the function to save the user's effort. The following command converts an array of event names to a vector containing 45 strings with different lengths. The conversion key is stored in data frame `eventChar.df`, in which the first column contains event names and the second column contains the characters to be converted. The object event1d holds the directory of file eve1d.txt (located in the user's R library in this example), which has 45 rows, each with different numbers of event names. Note that it is common to use a file name (and its file path if the file is not in the current directory) directly instead of an object (like `event1d`) in the function, and the users should not forget the quote sign.

```
> data(eventChar.df)
> event1d <- paste(path.package("GrpString"), "/extdata/eve1d.txt", sep = "")
> EveString(event1d, eventChar.df$event, eventChar.df$char)

[1] "D02F0E20DEDC0C30BDC0E45G050A0B5050A06BG0BA5607BA"
[2] "A1ABC21EF0230E03G032C30CBABGBA5G5A7G"
[3] "B1G1GEG10CEF2BAC3DEBA404B5G6F6A"
. . .
[45] "DCB010A0Q1EF21F0FGF0G0GF0B0BC20D0D0303DF04030CF45050B0CBCB05607B0"
```

After the event-string conversion, a string may contain consecutive repeating characters, but sometimes the users are only interested in "collapsed" strings in which successively duplicated characters are removed. Function `DupRm()` allows users to obtain collapsed strings.

```
> dup1 <- "000<<<<<DDDFFF333333qqqqqKKKKK33FFF"
> dup3 <- "aaBB111^^~~~555667777!!!###$$$$$$&&&(((((***)))))@@@@@>>>>99"
> dup13 <- c(dup1, dup3)
> DupRm(dup13)

[1] "0<DF3qK3F"           "aB1^~5670!#$&(*)@>9"
```

Detecting patterns

In **GrpString**, a pattern (also called common pattern) is defined as a substring with a minimum length of three that occurs at least twice among a group of strings. For a single string of length m, the total number of substrings of length 3 or more is n = (m - 1) × (m - 2) / 2. Note that a string itself is also considered as a substring. By using exhaustive search of substrings in the string vector, this package provides two functions to detect and organize patterns. The simplified version, `CommonPatt()`, returns one data frame. Because there could be thousands of substrings in a string vector, the function utilizes a cutoff to display more frequent or important patterns. The cutoff is the minimum percentage of the occurrence of patterns or substrings and is selected by the user. If the number of strings in a group is num, and the cutoff is selected as c, then only patterns with the minimum number of occurrence $f_{min} = num \times c\%$ will be returned by function `CommonPatt()`. In the following example, $num = 6$, $c = 30$. Thus, $f_{min} = 6 \times 30\% = 1.8$. Patterns with frequency (i.e., number of occurrence) ≥ 2 are displayed in column `Freq_grp`.

```
> strs.vec <- c("ABCDdefABCDa", "def123DC", "123aABCD", "ACD13", "AC1ABC",
               "3123fe")
> CommonPatt(strs.vec, low = 30)

   Pattern Freq_grp Percent_grp Length Freq_str Percent_str
      ABCD        3      50.00%      4        2      33.33%
       ABC        4      66.67%      3        3      50.00%
       123        3      50.00%      3        3      50.00%
       BCD        3      50.00%      3        2      33.33%
       def        2      33.33%      3        2      33.33%
```

The exported data frame is sorted by the length of pattern and then by the frequency or percentage of pattern. Note that the result contains two sets of pattern frequencies and percentages; one is of the overall occurrence (`Freq_grp` and `Percent_grp`) in a string group and the other excludes duplicated occurrences in the same string (`Freq_str` and `Percent_str`). As a consequence, in some cases Percent_grp can be larger than 100%, while Percent_str will not exceed 100%. The full version of this

pair, `CommonPattern()`, offers more options. In addition to the lowest minimum cutoff as described in function `CommonPatt()`, the user can select the highest minimum cutoff and the interval between the two cutoffs. Furthermore, the user can choose using a conversion key to convert patterns back to sequences of event names, which makes it easier to interpret the patterns. This function exports a set of .txt files in the current directory instead of a data frame.This is because the number of files and the numbers of rows in some files could be large, which might be difficult for the user to view the results in R directly. The names of these .txt files consist of the name of the input string vector and the percentages resulted from the cutoffs and the interval in the function. Patterns that occur at least twice are exported in a separate .txt file with `"_f2up"` appended to the name of the input string vector. For example, the following command exports three files: `strs.vec_30up.txt`, `strs.vec_50up.txt`, and `strs.vec_f2up.txt`. Each file contains a table that has the same columns as shown in the above result from function `CommonPatt()`.

```
> strs.vec <- c("ABCDdefABCDa", "def123DC", "123aABCD", "ACD13", "AC1ABC",
                "3123fe")
> CommonPattern(strs.vec, low = 30, high = 50, interval = 20)
```

Pattern information and featured patterns

The first function in this pair, `PatternInfo()`, lists some basic information about patterns in each string. The information includes the length of each string and the starting position of each pattern in a string. If a pattern does not appear in a string, `"-1"` will be returned. If a pattern occurs at least twice in a string the default position is for the first occurrence, although there is an option for the users to choose the last occurrence of duplicated patterns in the same string.

```
> strs.vec <- c("ABCDdefABCDa", "def123DC", "123aABCD", "ACD13", "AC1ABC",
                "3123fe")
> patts <- c("ABC", "123")
> PatternInfo(patts, strs.vec)
```

```
             length ABC 123
ABCDdefABCDa     12   1  -1
def123DC          8  -1   4
123aABCD          8   5   1
ACD13             5  -1  -1
AC1ABC            6   4  -1
3123fe            6  -1   2
```

The second function in this pair, `FeaturedPatt()`, is developed to distinguish the pattern characteristics of two string groups. It compares two groups of strings and discovers featured patterns in each of the groups. It also lists basic information about the featured patterns in each string. In this function, "featured" means patterns in a resulting pattern group can only exist in one of the two input-pattern groups. Note that "featured patterns" shared by strings in both string groups are allowed. This is because, in practice, it is difficult to find a pattern that is exclusively present in all the strings within only one group. As a result, in this function, "featured patterns" are probably obtained from two pattern vectors, each of which contains patterns that are shared by a certain percentage of strings in a group. The following simple example uses two groups (or vectors) of strings, `s_grp1` and `s_grp2`, which are very similar to those in Figure 1. The two pattern vectors, `p1` and `p2`, are obtained from `s_grp1` and `s_grp2`, respectively by applying function `CommonPatt()` with cutoff = 60%. Thus, `p1` contains patterns that are shared by at least 60% of the strings in `s_grp1` and `p2` contains patterns that are shared by at least 60% of the strings in `s_grp2`. To apply `FeaturedPatt()`, the first two arguments are `p1` and `p2`, which serve as the "input-pattern groups", and the last two arguments are the original string groups `s_grp1` and `s_grp2`. The function exports five text files: uni_p1-p2.txt, p1-vs-p2_in_s_grp1.txt, p1-vs-p2_in_s_grp2.txt, p2-vs-p1_in_s_grp1.txt, and p2-vs-p1_in_s_grp2.txt. Figure 2 shows the content of the first three files. The resulting featured patterns are listed in File 1 (uni_p1_p2.txt). It can be seen that "ABC" (featured pattern group 1) does not appear in `p2` (input-pattern group 2), but can be found in `s_grp2` (20% of string group 2); "xyz" (featured pattern group 2) does not appear in `p1` (input-pattern group 1), but can be found in `s_grp1` (40% of string group 1). In File 2 (p1-vs-p2_in_s_grp1.txt) and File 3 (p1-vs-p2_in_s_grp2.txt), the four columns are (original) string group, length of string, number of featured patterns in string, and the starting position of featured pattern in string. If `p1` had n patterns instead of one, the number of columns in File 2 and File 3 should have $3 + n$ columns; starting from the 4th column, each column lists the starting position of a featured pattern.

```
> s_grp1 <- c("ABC123", "123ABC", "ABCx123", "123xyz", "xyz123")
> s_grp2 <- c("xyz123", "xyzA123", "123xyz", "ABC123", "x123")
```

```
> p1 <- c("123", "ABC")
> p2 <- c("123", "xyz")

> FeaturedPatt(p1, p2, s_grp1, s_grp2)
```

 # File 1. "uni_p1_p2.txt"

"onlyIn_s_grp1"	"onlyIn_s_grp2"
"1" "ABC"	"xyz"

 # File 2. "p1-vs-p2_in_s_grp1.txt"

"s_grp1"	"Length"	"numPattern"	"ABC"
"ABC123"	6	1	1
"123ABC"	6	1	4
"ABCx123"	7	1	1
"123xyz"	6	0	-1
"xyz123"	6	0	-1

 # File 3. "p1-vs-p2_in_s_grp2.txt"

"s_grp2"	"Length"	"numPattern"	"ABC"
"xyzA123"	7	0	-1
"xyz123"	6	0	-1
"123xyz"	7	0	-1
"ABC123"	6	1	1
"x123"	4	0	-1

Figure 2: Three of the five files exported using function `featuredPatt()`. File 1. "uni_p1_p2.txt": featured patterns in the original string groups. File 2. "p1-vs-p2_in_s_grp1.txt": information of featured pattern "ABC" (which is from original pattern group `p1`) in original string group `s_grp1`. File 3. "p1-vs-p2_in_s_grp2.txt": information of featured pattern "ABC" (which is from original pattern group `p1`) in original string group `s_grp2`.

Ideally, a featured pattern presents in 100% of the strings of one string group but in none of the strings of the other group. However, obtaining featured patterns based on cutoffs smaller than 100% as described above still has significance. If two groups of strings are different, then patterns in featured pattern vector 1 are likely to present more frequently in string group 1 than in string group 2. For example, although the featured pattern `"ABC"` exists in both string groups, it can be found in 60% of the strings in String Group 1, but in only 20% in String Group 2 (Figure 1 and Figure 2). Note that Figure 1 and Figure 2 only show a simplified example. In reality, each string may contain multiple featured patterns. Therefore, featured patterns, including their numbers and positions in a string (Figure 2), in turn can be used to categorize the string, i.e., to classify or predict to which group the string belongs.

Transition matrix and information

In addition to patterns, transitions are also often reported in string analysis. For example, researchers in eye-tracking studies may be interested in the gaze transitions within a specific location or between two different locations. A transition is a substring with length of 2. For a single string of length m, the total number of transitions n = m - 1. According to the definition of transition, transitions are not applied to strings with length < 2. The first function in this pair, `TransInfo()`, returns the numbers of two types of transitions — letter and digit by default — in a group of strings. Letters and digits are used as default because they are common components in strings and can represent two different types of events. The function also reports the number of transitions that do not belong to either of the two types. To be more flexible, the users can define any two types of transitions (see the following example).

```
> strs.vec <- c("ABCDdefABCDa", "def123DC", "123aABCD", "ACD13", "AC1ABC",
                "3123fe")
> TransInfo(strs.vec)

  transition_name transition_number
1           type1                24
```

```
2          type2                8
3          mixed                7
```

The second function in this pair, `TransMx()`, returns grand transition matrices in a group of strings. The first matrix contains the numbers of transitions between two characters in a string vector. Normalized transition numbers are stored in the second matrix. A data frame that contains transitions sorted by frequency is also returned. Moreover, this function provides an option to export a transition matrix for each individual string into the current directory. The following shows the usage and results of `TransMx()`; the matrix of normalized transitions $Transition_Normalized_Matrix and the data frame of the sorted transitions $Transition_Organized are not shown. As an example, the readers can understand transition matrix $Transition_Matrix by looking at the transition "12". In vector `strs.vec`, the transition "12" occurs three times (in the 2nd, 3rd and 6th strings). Thus, the value is 3 in the cell of row 1, column 2 of $Transition_Matrix, which is the total number of the transition "From 1 To 2".

```
> strs.vec <- c("ABCDdefABCDa", "def123DC", "123aABCD", "ACD13", "AC1ABC",
                "3123fe")
> TransMx(strs.vec)

$Transition_Matrix
   From/To 1 2 3 a A B C d D e f
1        1 0 3 1 0 1 0 0 0 0 0 0
2        2 0 0 3 0 0 0 0 0 0 0 0
3        3 1 0 0 1 0 0 0 0 1 0 1
4        a 0 0 0 0 1 0 0 0 0 0 0
5        A 0 0 0 0 0 4 2 0 0 0 0
6        B 0 0 0 0 0 0 4 0 0 0 0
7        C 1 0 0 0 0 0 0 0 4 0 0
8        d 0 0 0 0 0 0 0 0 0 2 0
9        D 1 0 0 1 0 0 1 1 0 0 0
10       e 0 0 0 0 0 0 0 0 0 0 2
11       f 1 0 0 0 1 0 0 0 0 1 0
```

Transition entropy

Transition entropy measures the diversity of the transitions in a string or a group of strings. It can be used as an estimate of string complexity. Larger entropy reflects more evenly distributed transitions and smaller entropy reflects more biased distribution of transitions. Entropy in the current version of package **GrpString** is calculated using the Shannon entropy formula (Shannon, 1948):

$$H = -\sum_{i=1}^{n} freqs_i \times log_2\left(freqs_i\right) \tag{1}$$

Here, $freqs_i$ is the ith frequency of a transition in a string or a string vector. These are normalized transition numbers, which can be obtained in the normalized transition matrix exported by function `TransMx()` in this package. The formula is equivalent to the function `entropy.empirical()` in the **entropy** package (Hausser and Strimmer, 2014) when setting $unit = "log_2"$. Strings with length < 2 are not counted for calculating entropies because transitions are not applied to those strings.

One function in this pair, `TransEntro()`, computes the overall transition entropy for a group of strings. The other function, `TransEntropy()`, returns the transition entropy for each of the strings in a group. Note that the third string (`"A"`) is skipped in the result because the length of this string is 1. This function provides another way to quantitatively and/or statistically compare two groups of strings. For example, with the entropy values obtained from `TransEntropy()`, the users can conduct a t-test to examine whether the difference in complexity of two string groups is statistically significant.

```
> stra.vec <- c("ABCDdefABCDa", "def123DC", "A", "123aABCD", "ACD13", "AC1ABC", "3123fe")
> TransEntro(stra.vec)

[1] 4.272331

> TransEntropy(stra.vec)

  String  Entropy
1      1 2.913977
2      2 2.807355
```

```
3      4 2.807355
4      5 2.000000
5      6 2.321928
6      7 2.321928
```

Statistical difference between two groups of strings and distribution of differences

In this pair of functions, `StrDif()` is the main function, and `HistDif()` is the auxiliary function that optimizes the histogram generated by the former. `StrDif()` employs a permutation test to statistically compare the difference between two groups of strings based on normalized Levenshtein distances (LDs). A Levenshtein distance between two strings is the minimum number of operations to transform one string into the other by inserting, deleting, or replacing characters. LDs in `StrDif()` are calculated directly using R's native function `adist()` in package **utils**. A normalized LD between two strings is the LD value divided by the length of the longer string. For two groups of strings, there are two types of LDs. One is within-group LD that is computed by comparing each string with all the others in the same group; the other is between-group LD that is computed by comparing each string in one group with all strings in the other group.

When comparing two groups of strings statistically, the null hypothesis is that the average normalized between-group LD is equal to the average normalized within-group LD. Under this null hypothesis, the difference (d^*) between the two average normalized LDs should be zero. We define $d^* = d_{between} - d_{within}$ as the difference for the original two groups of strings. In a permutation test, strings are re-allocated randomly between the two new groups, each with the same number of strings as the original groups. Every new combination of two new groups results in a new distance $d^{*'} = d'_{between} - d'_{within}$. If one group has n strings and the other group has m strings, the total number of all possible permutations (NP) is

$$NP = \frac{(n+m)!}{n!m!} \tag{2}$$

The p value is then calculated using equation 3.

$$p = \frac{\sum_{i=1}^{NP} \left(d_i^{*'} \geq d_i^* \right)}{NP} \tag{3}$$

In practice, NP can be very large, which will make the running time of computing all $d^{*'}$ values very long. Therefore, a smaller subset of all the possible combinations of two groups of strings (i.e., permutations) is usually selected using the Monte Carlo method to reduce the burden of computations. This function allows the users to choose the number of permutations. The default value is 1000, which has been proved to be a reasonable number (Tang et al., 2012).

`StrDif()` prints $d_{between}$, d_{within}, d^*, and p value. It also returns the $d^{*'}$ (including d^*) vector, in which the total number of elements is the sample size in the Monte Carlo simulation. The users should use a vector object to store the $d^{*'}$ values. Furthermore, the function generates a histogram demonstrating the distribution of $d^{*'}$ (Figure 3). The x-axis is the value of $d^{*'}$ and the y-axis is the absolute value of the frequency of $d^{*'}$. In the graph, d^* will be marked as "Observed Difference" with the p value labeled beside. Because the positions of the legend and text in the histogram may vary due to different distributions and p values resulted from different string groups, `StrDif()` provides options for the users to define these two positions in order to obtain the graph with optimal appearance. However, since the users usually do not know the ideal positions in advance, they may have to re-run the function at least once to adjust the positions. To avoid rerunning `StrDif()`, which can take a long time, `HistDif()` allows the users to adjust the positions by directly using the vector containing all $d^{*'}$ generated from `StrDif()`. This is another reason we suggest that the users save $d^{*'}$ values in a vector when performing `StrDif()`.

```
> strs1.vec <- c("ABCDdefABCDa", "def123DC", "123aABCD", "ACD13", "AC1ABC", "3123fe")
> strs2.vec <- c("xYZdkfAxDa", "ef1563xy", "BC9Dzy35X", "AkeC1fxz", "65CyAdC",
                 "Dfy3f69k")
> ld.dif.vec <- StrDif(strs1.vec, strs2.vec, num_perm = 500, p.x = 0.025)

For the initial two groups of strings,
   the average normalized between-group Levenshtein Distance is:  0.85056
   the average normalized within-group Levenshtein Distance is:  0.84306
```

```
  the difference in the average normalized Levenshtein Distance between
between-group and within-group is:  0.00751.
  The p value of the permutation test is:   0.41000
```

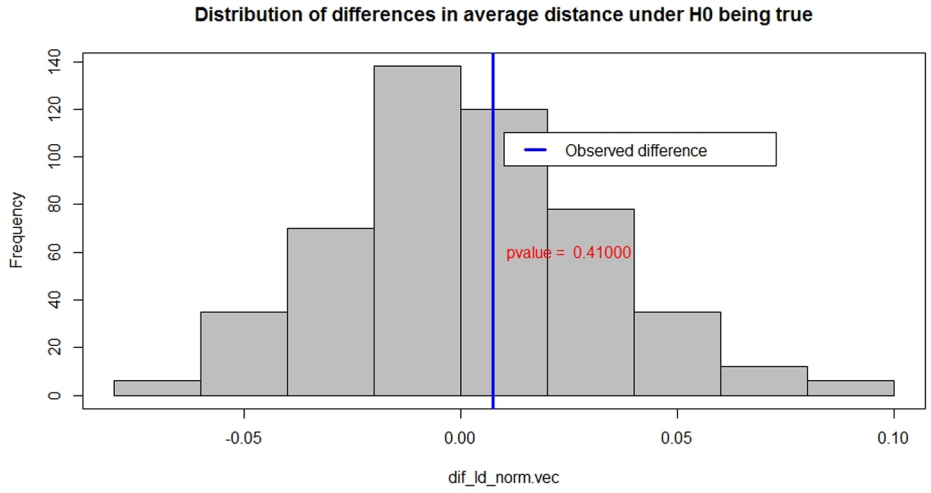

Figure 3: Histogram from function `StrDif()`: distribution of the differences in the average normalized Levenshtein Distance between between-group and within-group strings. The original difference ("Observed difference") d^* is marked in a blue line with the p value labeled beside. Because a subset of all the possible permutations are selected randomly, the actual p value may be slightly different each time the user runs the function.

Cluster analysis

In this pair, the functions perform string clustering based on Levenshtein distance matrices. Function `StrHclust()` utilizes hierarchical clustering and exports a hierarchical dendrogram (Figure 4), which may suggest a number of clusters. When the user selects a number of clusters, the function assigns each string to a corresponding cluster.

```
> strs3.vec <- c("ABCDdefABCDa", "AC3aABCD", "ACD1AB3", "xYZfgAxZY", "gf56xZYx",
                 "AkfxzYZg")
> StrHclust(strs3.vec)
```

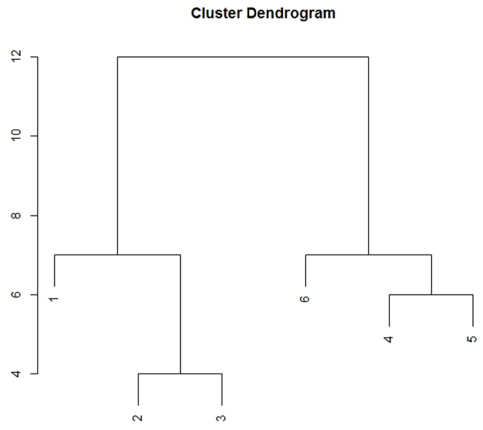

Figure 4: Dendrogram from function `StrHclust()`.

Function `StrKclust()` utilizes k-means clustering and assigns strings to their clusters based on the number of clusters the user chooses. In addition, a cluster plot is exported to visualize the clusters (Figure 5). In the plot, each labelled point represents a string in the input vector. Points are clustered by ellipses, which are also labelled. Note that by default, the ellipses are not shaded; but the users can shade the ellipses when setting "$shade = TRUE$" in the function.

```
> strs3.vec <- c("ABCDdefABCDa", "AC3aABCD", "ACD1AB3", "xYZfgAxZY", "gf56xZYx",
                 "AkfxzYZg")
> StrKclust(strs3.vec)

  Cluster      Strings
1       1 ABCDdefABCDa
2       1     AC3aABCD
3       1      ACD1AB3
4       2    xYZfgAxZY
5       2     gf56xZYx
6       2     AkfxzYZg
```

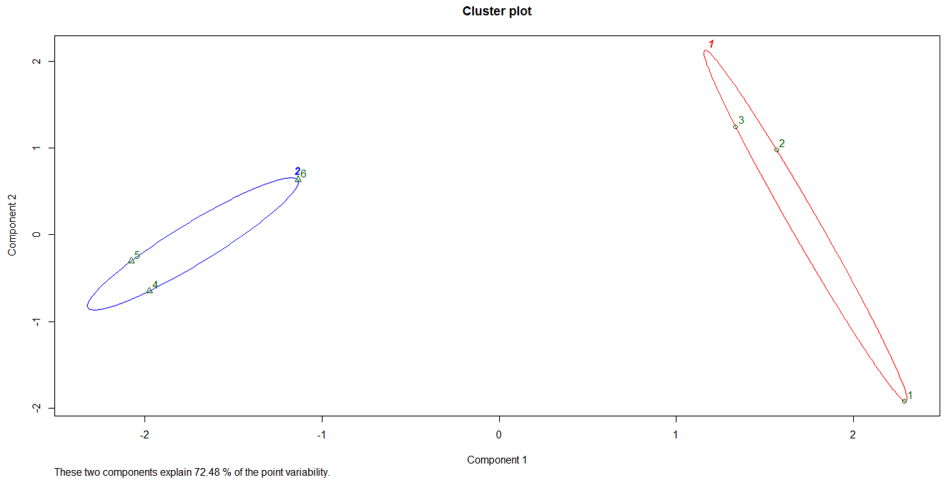

Figure 5: Cluster plot from function `StrKclust()`. All the points (1 to 6) and ellipses (1 and 2) are labelled.

Summary

This article describes the **GrpString** package for analyzing and comparing groups of strings. Most functions in package **GrpString** were initially developed for analyzing groups of scanpaths (i.e., sequences of eye gazes) in eye-tracking studies. Nevertheless, as described above, this R package can be applied in analysis of any type of character strings, and the strings do not have to be associated with any event or state data. It should be noted that the sole usage of one function may not be sufficient to draw conclusion when answering a research question. An example is `StrDif()`. There could be various factors, such as string lengths, that can affect the p value of a permutation test for string comparison. Therefore, when the users claim a statistically significant or non-significant difference between two string groups, the results from `CommonPattern()` and/or `TransEntropy()` may be used to support the conclusion. One limitation of the current version of **GrpString** is that it only provides basic and common options in some functions. For instance, only Levenshtein distance is available when distances between strings are computed (`StrDif()`, `StrHclust()` and `StrKclust()`); the updated versions should include an argument for different types of distance. Finally, there are many other advanced analytical methods that have not been included in this package, such as determining the centroid or median string in a string vector (de la Higuera and Casacuberta, 2000; Martínez-Hinarejos et al., 2000) and using Markov chains for string modeling and classification (Krejtz et al., 2014). We plan to build more functions into **GrpString** in the future and welcome feedback from the users to improve this package.

Acknowledgement

This work was supported by the Startup Grant from the Office of Research at the University of Georgia (No. 1026AR168004) and the Journal of Chemical Education Grants. We thank Dr. Rhonda DeCook in the Department of Statistics and Actuarial Science at the University of Iowa, who wrote the first version of function `StrDif()`.

Bibliography

C. de la Higuera and F. Casacuberta. Topology of strings: Median string is np-complete. *Theoretical computer science*, 230(1-2):39–48, 2000. URL https://doi.org/10.1016/S0304-3975(97)00240-5. [p9]

A. Gabadinho, R. Gilbert, M. Nicolas, and S. Matthias. Analyzing and visualizing state sequences in r with traminer. *Journal of Statistical Software*, 40:1–37, 2011. URL https://www.jstatsoft.org/article/view/v040i04. [p1]

A. Gabadinho, R. Gilbert, M. Nicolas, and S. Matthias. *TraMineR: Trajectory Miner: a Toolbox for Exploring and Rendering Sequences*, 2017. URL https://CRAN.R-project.org/package=TraMineR. R package version 2.0.7. [p1]

M. Gagolewski and B. Tartanus. *Stringi: Character String Processing Facilities*, 2017. URL https://CRAN.R-project.org/package=stringi. R package version 1.1.5. [p1]

G. Grothendieck. *Gsubfn: Utilities for Strings and Function Arguments*, 2014. URL https://CRAN.R-project.org/package=gsubfn. R package version 0.6-6. [p1]

J. Hausser and K. Strimmer. *Entropy: Estimation of Entropy, Mutual Information and Related Quantities*, 2014. URL https://CRAN.R-project.org/package=entropy. R package version 1.2.1. [p6]

S. Jackman. *Uniqtag: Abbreviate Strings to Short, Unique Identifiers*, 2015. URL https://CRAN.R-project.org/package=uniqtag. R package version 1.0. [p1]

K. Krejtz, T. Szmidt, A. T. Duchowski, and I. Krejtz. Entropy-based statistical analysis of eye movement transitions. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, pages 159–166. ACM, 2014. URL https://doi.org/10.1145/2578153.2578176. [p9]

C. S. Marcum and C. T. Butts. Constructing and modifying sequence statistics for relevent using informr in r. *Journal of Statistical Software*, 64:1–36, 2015. URL https://www.jstatsoft.org/article/view/v064i05. [p1]

C. D. Martínez-Hinarejos, A. Juan, and F. Casacuberta. Use of median string for classification. In *Pattern Recognition, 2000. Proceedings. 15th International Conference on*, volume 2, pages 903–906. IEEE, 2000. URL https://doi.org/10.1109/ICPR.2000.906220. [p9]

P. Meissner. *Stringb: Convenient Base R String Handling*, 2016. URL https://CRAN.R-project.org/package=stringb. R package version 0.1.13. [p1]

C. E. Shannon. A mathematical theory of communication. *Bell System Technical Journal*, 27:623–656, 1948. URL http://math.harvard.edu/~ctm/home/text/others/shannon/entropy/entropy.pdf. [p6]

H. Tang and N. J. Pienta. *GrpString: Patterns and Statistical Differences Between Two Groups of Strings*, 2017. URL https://CRAN.R-project.org/package=GrpString. R package version 0.3.2. [p1]

H. Tang, J. J. Topczewski, A. M. Topczewski, and N. J. Pienta. Permutation test for groups of scanpaths using normalized levenshtein distances and application in nmr questions. In *Proceedings of the Symposium on Eye Tracking Research and Applications*, pages 169–172. ACM, 2012. URL https://doi.org/10.1145/2168556.2168584. [p7]

M. van der Loo. The stringdist package for approximate string matching. *The R Journal*, 6:111–122, 2014. URL https://journal.r-project.org/archive/2014-1/loo.pdf. [p1]

M. van der Loo. *Stringdist: Approximate String Matching and String Distance Functions*, 2016. URL https://CRAN.R-project.org/package=stringdist. R package version 0.9.4.4. [p1]

H. Wickham. Stringr: Modern, consistent string processing. *The R Journal*, 2:38–40, 2010. URL https://journal.r-project.org/archive/2010/RJ-2010-012/RJ-2010-012.pdf. [p1]

H. Wickham. *Stringr: Simple, Consistent Wrappers for Common String Operations*, 2017. URL https://CRAN.R-project.org/package=stringr. R package version 1.2.0. [p1]

Hui Tang
Department of Chemistry, University of Georgia
140 Cedar Street, Athens, GA 30602-2556

United States
huitang@uga.edu

Elizabeth L. Day
Department of Chemistry, University of Georgia
140 Cedar Street, Athens, GA 30602-2556
United States
elday@uga.edu

Molly B. Atkinson
Department of Chemistry, University of Georgia
140 Cedar Street, Athens, GA 30602-2556
United States
matkin@uga.edu

Norbert J. Pienta
Department of Chemistry, University of Georgia
140 Cedar Street, Athens, GA 30602-2556
United States
npienta@uga.edu