Package ‘soundcorrs’

November 16, 2020

Title Semi-Automatic Analysis of Sound Correspondences

Version 0.4.0

Description A set of tools that can be used in computer-aided analysis of sound correspondences between languages, plus several helper functions. Analytic functions range from purely qualitative analysis, through statistic methods yielding qualitative results, to an entirely quantitative approach.

Depends R (>= 3.5.0)

License GPL-3

Encoding UTF-8

LazyData true

RoxygenNote 7.1.1

Suggests knitr (>= 1.21), rmarkdown (>= 1.11), shiny (>= 1.2.0), shinyjqui (>= 0.3.3)

VignetteBuilder knitr

NeedsCompilation no

Author Kamil Stachowski [aut, cre]

Maintainer Kamil Stachowski <kamil.stachowski@gmail.com>

Repository CRAN

Date/Publication 2020-11-16 12:30:03 UTC

R topics documented:

`addSeparators` *Intersperse a vector of strings with a character or string.*

Description

Primarily intended to insert separators into a column of words, to facilitate manual segmentation and aligning.

Usage

```
addSeparators(x, separator = "|")
```

Arguments

`x`	[character vector] The strings to be interspersed.
`separator`	[character] The string with which to intersperse. Defaults to `"|"`.

Details

Preparation of data for `soundcorrs` consists of segmentation and alignment. Segmentation can proceed on phoneme-by-phoneme, morpheme-by-morpheme, or any other basis; the only constraint is that each word in a pair/triple/... of words must contain the same number of segments. Segments are indicated by separators, by default the character `"|"`. The action of inserting separators, potentially between every two letters, in a large dataset, can become time consuming. `addSeparators` automates at least this part of the process.

Value

character vector A vector of interspersed strings.

Examples

```
addSeparators (c("word","mot","focal"), ".")
```

`allCooccs` *Generate all co-occurrence contingency tables for a dataset.*

Description

Generate all correspondence-to-correspondence or correspondence-to-metadata contingnecy tables for a dataset.

Usage

```
allCooccs(data, column, count, unit, bin)
```

Arguments

`data`	[soundcorrs] The dataset from which to draw frequencies. Only datasets with two languages are supported.
`column`	[character] Name of the column with metadata. If `NULL`, sound correspondences are cross-tabulated with themselves. Defaults to `NULL`.
`count`	[character] Report the absolute number of times or words, or relative to how many times or in how many words the given segments co-occur in L1 or L2. Accepted values are `"a(bs(olute))"` and `"r(el(ative))"`. Defaults to "a".
`unit`	[character] Count how many times a correspondence occurs or in how many words it occurs. Accepted values are `"o(cc(ur(ence(s))))"` and `"w(or(d(s)))"`. Defaults to `"w"`.
`bin`	[logical] Whether to bin tables before applying `fun` to them. Defaults to `TRUE`.

Details

A contingency table such as produced by `coocc` may be insightful but more often than not statistical tests cannot be applied directly to it, or at least they would not produce meaningful results. This function splits such a table into blocks such that each block only contains the correspondences of a single segment. The resulting slices, additionally binned or not (cf. `binTable`), can be then passed to `lapplyTest` for a near-automatic application of a test.

Value

list A list of tables.

See Also

`coocc`, `binTable`, `lapplyTest`

Examples

```
dataset <- loadSampleDataset ("data-abc")
allCooccs (dataset)
allCooccs (dataset, "DIALECT.L2", unit="o")
```

`allPairs` *Produce a list of all sound correspondences and all pairs in which they are attested.*

Description

Take all segment-to-segment correspondences in the dataset, and produce for each a section composed of a title, a contingency table of all renderings of the given segment, and subsections listing all word pairs in which the given rendering is attested, all nicely formatted.

Usage

```
allPairs(data, file, count, unit, direction, cols, formatter, ...)
```

Arguments

`data`	[character] The dataset. Only datasets with two languages are supported.
`file`	[character] Name of the file to write the formatted list to. If `NULL`, the output will be printed to the screen. Defaults to `NULL`.
`count`	[character] Report the absolute number of times or words, or relative to how many times or in how many words the given segments co-occur in L1 or L2. Accepted values are `"a(bs(olute))"` and `"r(el(ative))"`. Defaults to "a".
`unit`	[character] Count how many times a correspondence occurs or in how many words it occurs. Accepted values are `"o(cc(ur(ence(s))))"` and `"w(or(d(s)))"`. Defaults to `"w"`.
`direction`	[integer] If `1`, correspondences are in the order Language1 > Language2 ("x yields y"). If `2`, the order is Language2 < Language1 ("y originates from x"). Defaults to `1`.
`cols`	[character vector] Which columns of the dataset to print. Can be a vector of names, `"aligned"` (the two columns with segmented, aligned words), or `"all"` (all columns). Defaults to `"aligned"`.
`formatter`	[function] The function to which to pass unformatted data. Available formatters are: `formatter.none`, `formatter.html`, and `formatter.latex`. Defaults to `formatter.none`.
`...`	Additional arguments passed to `formatter`.

Details

`summary.soundcorrs` can produce a table of all segment-to-segment correspondences in a dataset, and `findExamples` and `findPairs` can find all the pairs of words which realize those correspondences, but combining their outputs is a time-consuming, and unnecessary manual labour. The same, or at least a very similar result can be produced automatically by this function. Its output is divided into sections, each comprised of the appropriate slice of the contingency table, and a list of all the examples which are relevant for the given correspondence. The output can be raw, or formatted as LaTeX or HTML, and it is not too difficult to write one's own, custom formatting function.

Value

character A formatted list of of all segment-to-segment correspondences and all pairs in which they are attested.

See Also

`findPairs`, `summary.soundcorrs`

Examples

```
dataset <- loadSampleDataset ("data-abc")
allPairs (dataset)
allPairs (dataset, formatter=formatter.latex, cols=c("ORTHOGRAPHY.L1", "ORTHOGRAPHY.L2"))
```

---

applyChanges *Apply a series of sound changes to a series of words.*

Description

Apply a list of soundchange's to a series of words, possibly with additional metadata, and possibly compare the results to a prediction.

Usage

```
applyChanges(data, changes, source, target, meta, highlight)
```

Arguments

`data`	[soundcorrs] A `soundcorrs` object.
`changes`	[soundchange] The list of `soundchange`'s to apply.
`source`	[character] Name of the column containing words to which to apply `soundchange`'s.
`target`	[character] Name of the column containing words to which to compare the results. Defaults to `NULL`.
`meta`	[character] Name of the column containing metadata to be passed to `soundchange` functions alongside words. Defaults to `NULL`.
`highlight`	[character] Highlight the differences between the intermediate forms in `$tree`, as well as between the results in `$end` and `target`? Can be `NULL` (do not highlight), `"console"` (highlight for the console), or `"HTML"` (highlight for a web browser). Defaults to `NULL`.

Details

Functions in `soundchange` objects are allowed to return more than one value, which makes manual application of a series of changes highly inconvenient and prone to errors. This function automates the process, while keeping track of all the intermediate forms. It returns the result in three formats: only the final shapes; their comparison to the shapes given under the `target` argument; and a tree with all the steps along the way. By default, only the final shapes are printed. All the three formats are accessible as elements of a named list: `$end`, `$match`, and `$tree`, respectively.

Note that the application of sound changes does not require the data to be segmented and aligned. If sound changes are the only goal of the project, these two time-consuming steps can be safely omitted.

Value

list.applyChanges A list with three fields: `$end`, a named list with the final results; `$match`, a named list with one of three values: `0` when none of the final results matches the `target`, `0.5` when at least one of the final results matches the `target`, or `1` when all the final results match the `target`; lastly `$tree`, a list tracing all the intermediate forms.

See Also

`print.list.applyChanges`, `print.tree.applyChanges`

Examples

```
# prepare sample data
dataset <- loadSampleDataset ("data-capitals")
changes <- list (loadSampleDataset("change-dl2l"), loadSampleDataset("change-rhotacism"))
# apply the changes
applyChanges (dataset, changes, "ORTHOGRAPHY.German")
applyChanges (dataset, changes, "ORTHOGRAPHY.German")$tree
applyChanges (dataset, changes, "ORTHOGRAPHY.German", "ORTHOGRAPHY.Polish", highlight="console")
```

`binTable` *Sum all rows and all columns in a table, except for the selected ones.*

Description

Useful for when the data are scarce and `chisq.test` returns a warning, or when a more specific analysis of the data is required.

Usage

```
binTable(x, row, col)
```

Arguments

`x`	[data.frame/matrix/table] Table to be binned.
`row`	[integer/vector] The rows to not be binned.
`col`	[integer/vector] The columns to not be binned.

Details

When working with sparse data, the absolute values in a table are sometimes too low to allow for the use of various statistical tests, or the features too numerous for the result of a statistical test to be clearly interpretable. In such cases, a solution may be found in binning, i.e. in combining all the rows or columns into one, with the exception of select few. For example, a 10x10 table may be thus reduced to a 2x2 or a 2x3 one. The values are magnified while the number of features is reduced.

Value

table Table with some of its data binned.

Examples

```
mtx <- matrix (1:16, nrow=4, dimnames=list(paste0("r",1:4),paste0("c",1:4)))
binTable (mtx, 1, 1)
binTable (mtx, 1, c(1,3))
```

---

cbind.soundcorrs *Attach one or more columns to a soundcorrs object.*

Description

Attach one or more columns to a soundcorrs object. Note that sound correspondences attached with this function will not be usable as such.

Usage

```
## S3 method for class 'soundcorrs'
cbind(data, ...)
```

Arguments

`data`	[soundcorrs] The soundcorrs object.
`...`	Objects to be attached.

Details

Once a data frame is enclosed in a soundcorrs object, it is recommended that it not be manually altered in any way. `cbind.soundcorrs` provides a safe way of adding a column to it.

Value

soundcorrs The original soundcorrs object with the columns attached.

Examples

```
dataset <- loadSampleDataset ("data-ie")
cbind (dataset, ID=1:nrow(dataset$data))
cbind (dataset, FAMILY="Indo-European")
```

char2value — *Information that the `char2value()` function is obsolete.*

Description

Since version 0.2.0 it is no longer available. If you need its functionality, please contact kamil.stachowski@gmail.com

Usage

```
char2value(...)
```

Arguments

`...`	Ignored, only for compatibility.

coocc — *Generate a contingency table of co-occurrences of sound correspondences with themselves, or with metadata.*

Description

Take all segment-to-segment correspondences in a dataset, and cross-tabulate them with themselves or with metadata taken from a separate column.

Usage

```
coocc(data, column, count, unit)
```

Arguments

`data`	[soundcorrs] The dataset from which to draw frequencies. Only datasets with two languages are supported.
`column`	[character] Name of the column with metadata. If `NULL`, sound correspondences are cross-tabulated with themselves. Defaults to `NULL`.
`count`	[character] Report the absolute number of times or words, or relative to how many times or in how many words the given segments co-occur in L1 or L2. Accepted values are `"a(bs(olute))"` and `"r(el(ative))"`. Defaults to "a".
`unit`	[character] Count how many times a correspondence occurs or in how many words it occurs. Accepted values are `"o(cc(ur(ence(s))))"` and `"w(or(d(s)))"`. Defaults to `"w"`.

Details

A set of segmented and aligned word pairs/triples/..., such as one held in a `soundcorrs` object, can be turned into a contingency table in more than one way. This function creates a table which details how often various sound correspondences co-occur in one word. Both rows and columns are named in the same way: L1 phoneme + underscore ("_") + L2 phoneme. The values in the table can be absolute or relative, and they can represent the number of times the given correspondence co-occurs, or the number of words in which it co-occurs. For example, in the pair German koala : French koala, the correspondence G a : Fr a (`"a_a"`) co-occurs twice: the correspondence of the first two a's co-occurs with the correspondence of the second two a's, and vice versa. When the numbers are relative, they add up to 1 in blocks where each block is an intersection of rows and columns whose names begin with the same segment, i.e. those which refer to the correspondences of the same segment. In the relative view, empty cells appear when the given correspondence never co-occurs, and therefore its relative frequency is 0 divided by 0.

Value

table The contingency table. The values represent how often the given correspondence co-occurs in the same word with the other correspondence or with the piece of metadata (cf. `summary`).

See Also

`summary.soundcorrs`, `allCooccs`

Examples

```
dataset <- loadSampleDataset ("data-abc")
coocc (dataset)
coocc (dataset, "DIALECT.L2")
round (coocc(dataset,"DIALECT.L2",count="r"), digits=3)
```

`expandMeta` *Expand custom metacharacters to regular expressions.*

Description

Turn characters defined in a `transcription` as metacharacters into the corresponding regular expression.

Usage

```
expandMeta(data, x)
```

Arguments

`data`	[transcription] The `transcription` to use.
`x`	[character] A single string that contains metacharacters.

Value

character The string with metacharacters expanded.

See Also

transcription

Examples

```
dataset <- loadSampleDataset ("data-abc")
expandMeta (dataset$trans[[1]], "aN")
expandMeta (dataset$trans[[1]], "[+vow,-high]")
```

---

findExamples *Find all pairs/triples/... with corresponding sequences of sounds.*

Description

Sift the dataset for word pairs/triples/... such that the first word in the first languages contains the first sequence, the one in the second language the second sequence, and so on.

Usage

```
findExamples(
  data,
  ...,
  distance.start,
  distance.end,
  na.value,
  zeros,
  cols,
  perl
)
```

Arguments

`data`	[soundcorrs] The dataset in which to look.
`...`	[character] Sequences for which to look. May be regular expressions as defined in R, or in the transcription. If an empty string, anything will be considered a match.
`distance.start`	[integer] The allowed distance between segments where the sound sequences begin. A negative value means alignment of the beginning of sequences will not be checked. Defaults to -1.
`distance.end`	[integer] The allowed distance between segments where the sound sequences end. A negative value means alignment of the end of sequences will not be checked. Defaults to -1.

na.value	[numeric] Treat NA's as matches (0) or non-matches (-1)? Note that an empty string query takes precedence over na.value, that is even whan na.value is set to -1, NA's will show up in the results when the query is an empty string. Defaults to 0.
zeros	[logical] Take linguistic zeros into account? Defaults to FALSE.
cols	[character vector] Which columns of the dataset to return as the result. Can be a vector of names, "aligned" (the two columns with segmented, aligned words), or "all" (all columns). Defaults to "aligned".
perl	[logical] Use Perl-compatible regular expressions? Defaults to FALSE.

Details

One of the more time-consuming tasks, when working with sound correspondences, is looking for specific examples which realize the given correspondence. findExamples can fully automate this process. It has several arguments that can help fine-tune the search, of which perhaps the most important are distance.start and distance.end. It should be noted that their default values (-1 for both) mean that findExamples will find every such pair/triple/... of words, that the first word contains the first query, the second word the second query, etc. – regardless of whether these segments do in fact correspond to each other in the alignment. This is intentional, and stems from the assumption that in this case, false positives are generally less harmful, and most of all easier to spot than false negatives.

findExamples accepts regular expressions in queries, both such as are available in pure R, and such as have been defined in the transcription, in both notations accepted by expandMeta. It is highly recommended that the user acquaints him or herself with the concept, as it is in it that the true power of findExamples lies.

Value

df.findExamples A list with two fields: $data, a data frame with found examples; and $which, a logical vector showing which rows of data are considered matches.

See Also

findPairs

Examples

```
# In the examples below, non-ASCII characters had to be escaped for technical reasons.
# In the actual usage, Unicode is supported under BSD, Linux, and macOS.

# prepare sample dataset
dataset <- loadSampleDataset ("data-capitals")
# find examples which have "a" in all three languages
findExamples (dataset, "a", "a", "a")
# find examples where German has schwa, and Polish and Spanish have a Vr sequence
findExamples (dataset, "\u0259", "Vr", "Vr")
# as above, but the schwa and the two vowels must be in the same segment
findExamples (dataset, "\u0259", "V(?=r)", "V(?=r)", distance.start=0, distance.end=0, perl=TRUE)
# find examples where German has a-umlaut, Polish has a or e, and Spanish has any sound at all
```

```
findExamples (dataset, "\u00E4", "[ae]", "")
# find examples where German has a linguistic zero while Polish and Spanish do not
findExamples (dataset, "-", "[^-]", "[^-]", zeros=TRUE)
# find examples where German has schwa, and Polish and Spanish have a
findExamples (dataset, "\u0259", "a", "a", distance.start=-1, distance.end=-1)
# as above, but the schwa and the two a's must be in the same segment
findExamples (dataset, "\u0259", "a", "a", distance.start=0, distance.end=0)
```

`findPairs` *A convenience wrapper around* `findExamples`.

Description

Sift the dataset for word pairs such that the first word contains `x` and the second word contains `y` in the corresponding segment or segments.

Usage

```
findPairs(data, x, y, exact, cols)
```

Arguments

`data`	[soundcorrs] The dataset in which to look. Only datasets with two languages are supported.
`x`	[character] The sequence to find in language1. May be a regular expression. If an empty string, anything will be considered a match.
`y`	[character] The sequence to find in language2. May be a regular expression. If an empty string, anything will be considered a match.
`exact`	[numeric] If 0 or `FALSE`, `distance.start`=`distance.end`=-1, `na.value`=0, and `zeros`=`FALSE`. If 0.5, `distance.start`=`distance.end`=1, `na.value`=0, and `zeros`=`FALSE`. If 1 or `TRUE`, `distance.start`=`distance.end`=0, `na.value`=-1, and `zeros`=`TRUE`. Defaults to 0.
`cols`	[character vector] Which columns of the dataset to return as the result. Can be a vector of names, `"aligned"` (the two columns with segmented, aligned words), or `"all"` (all columns). Defaults to `"aligned"`.

Details

Probably the most common usage of `findExamples` is with datasets containing pairs of words. This function is a simple wrapper around `findExamples` which hopes to facilitate its use in this most common case. Instead of the five arguments that `findExamples` requires, this function only takes two. It is, of course, at the cost of control but should a more fine-tuned search be required, `findExamples` can always still be used instead of `findPairs`.

The default is the inexact mode (`exact` set to `0` or `FALSE`). It corresponds to `distance.start` and `distance.end` being both set to `-1`, `na.value` being set to `0`, and `zeros` being set to `FALSE`, which are also the default settings in `findExamples()`. The risk here are false positives. In my experience, however, those are rare, and because they are displayed, the user has a chance to spot them.

The opposite is the exact mode (`exact` set to 1 or `TRUE`), which corresponds to `distance.start` and `distance.end` being both set to `0`, `na.value` being set to `-1`, and `zeros` to `TRUE`. The risk are false negatives, in my experience both much more common than false positives in the inexact mode, and effectively impossible to spot as they are simply not displayed.

A middle ground is the semi-exact mode (`exact` set to 0.5), where `distance.start` and `distance.end` are both set to `1`, `na.value` is set to `0`, and `zeros` to `FALSE`. It decreases the risk of false positives while increasing only a little the risk of false negatives.

Value

df.findExamples A subset of the dataset, containing only the pairs with corresponding sequences. Warning: pairs with multiple occurrences of such sequences are only included once.

See Also

`findExamples`, `allPairs`

Examples

```
# In the examples below, non-ASCII characters had to be escaped for technical reasons.
# In the actual usage, Unicode is supported under BSD, Linux, and macOS.

# prepare sample dataset
dataset <- loadSampleDataset ("data-ie")
# run findPairs
findPairs (dataset, "a", "a")
findPairs (dataset, "e", "f", exact=0)
findPairs (dataset, "e", "f", exact=0.5)
findPairs (dataset, "e", "f", exact=1)
```

`findSegments` *Information that the* `findSegments` *function is obsolete.*

Description

Since version 0.2.0 it is no longer available. If you need its functionality, please contact kamil.stachowski@gmail.com

Usage

```
findSegments(...)
```

Arguments

`...`	Ignored, only for compatibility.

fitTable — *Fit multiple models to multiple datasets.*

Description

Apply `multiFit` to all rows or all columns of a table.

Usage

```
fitTable(models, data, margin, conv = vec2df.id, ...)
```

Arguments

`models`	[list] A list of models to fit `data` to. Each element must be a list with at least two named fields: `formula` which contains the formula, and `start` which is a list of lists of starting estimates. Regarding the formula, the converter functions (`fun`, below) use "X" and "Y" for column names.
`data`	[matrix/table] The data to fit `models` to.
`margin`	[integer] As in `apply`: the subscripts which the fitting function (cf. `multiFit`) will be applied over. Accepted values are: `1` for rows, and `2` for columns.
`conv`	[function] Function that converts vectors into data frames to which `models` will be fitted. Available functions are: `vec2df.id`, `vec2df.hist`, and `vec2df.rank`. Defaults to `vec2df.id`.
`...`	Additional arguments passed to `multiFit`).

Details

Finding the right model and the right starting estimates for a model is often a time consuming process, very inconvenient to do manually. This function automates it as much as possible. It takes a list of models and starting estimates, as well as a list of datasets, and fits all the models to all the datasets. If any of the fits results in an error or a warning, the message is saved and can be inspected in the output, but it does not halt the process. `fitTable` is an extension of `multiFit` which fits multiple models to a single dataset.

Value

list.multiFit A list of results returned by the fitting function (cf. `multiFit`).

See Also

`multiFit`

Examples

```
dataset <- loadSampleDataset ("data-abc")
models <- list (
"model A" = list (
formula = "Y ~ a/X",
start = list (list(a=1))),
"model B" = list (
formula = "Y ~ a/(1+exp(1)^X)",
start = list (list(a=1)))
)
fitTable (models, summary(dataset), 1, vec2df.rank)
```

---

formatter.html — *A formatter for allPairs. This one formats to HTML.*

Description

A formatter for allPairs. This one formats to HTML.

Usage

```
formatter.html(what, x, direction = 1)
```

Arguments

what	[character] What type of data is x.
x	The object to be formatted.
direction	[integer] If 1, correspondences are in the order Language1 > Language2 ("x yields y"). If 2, the order is Language2 < Language1 ("y originates from x"). Defaults to 1.

Value

character Formatted x.

Examples

```
# prepare sample dataset
fTrans <- system.file ("extdata", "trans-common.tsv", package="soundcorrs")
fData <- system.file ("extdata", "data-capitals.tsv", package="soundcorrs")
tmp.ger <- read.soundcorrs (fData, "German", "ALIGNED.German", fTrans)
tmp.pol <- read.soundcorrs (fData, "Polish", "ALIGNED.Polish", fTrans)
dataset <- merge (tmp.ger, tmp.pol)
# run allPairs
allPairs (dataset, unit="o", formatter=formatter.html)
```

formatter.latex *A formatter for allPairs. This one formats to LaTeX.*

Description

A formatter for allPairs. This one formats to LaTeX.

Usage

```
formatter.latex(what, x, direction = 1)
```

Arguments

what	[character] What type of data is x.
x	The object to be formatted.
direction	[integer] If 1, correspondences are in the order Language1 > Language2 ("x yields y"). If 2, the order is Language2 < Language1 ("y originates from x"). Defaults to 1.

Value

character Formatted x.

Examples

```
# prepare sample dataset
fTrans <- system.file ("extdata", "trans-common.tsv", package="soundcorrs")
fData <- system.file ("extdata", "data-capitals.tsv", package="soundcorrs")
tmp.ger <- read.soundcorrs (fData, "German", "ALIGNED.German", fTrans)
tmp.pol <- read.soundcorrs (fData, "Polish", "ALIGNED.Polish", fTrans)
dataset <- merge (tmp.ger, tmp.pol)
# run allPairs
allPairs (dataset, unit="o", formatter=formatter.latex)
```

formatter.none *A formatter for allPairs. This one does practically no formatting at all.*

Description

A formatter for allPairs. This one does practically no formatting at all.

Usage

```
formatter.none(what, x, direction = 1)
```

Arguments

`what`	[character] What type of data is `x`.
`x`	The object to be formatted.
`direction`	[integer] If 1, correspondences are in the order Language1 > Language2 ("x yields y"). If 2, the order is Language2 < Language1 ("y originates from x"). Defaults to 1.

Value

character Formatted x.

Examples

```
# prepare sample dataset
fTrans <- system.file ("extdata", "trans-common.tsv", package="soundcorrs")
fData <- system.file ("extdata", "data-capitals.tsv", package="soundcorrs")
tmp.ger <- read.soundcorrs (fData, "German", "ALIGNED.German", fTrans)
tmp.pol <- read.soundcorrs (fData, "Polish", "ALIGNED.Polish", fTrans)
dataset <- merge (tmp.ger, tmp.pol)
# run allPairs
allPairs (dataset, unit="o", formatter=formatter.none)
```

`lapplyTest` *Apply a function to a list.*

Description

Takes a list and applies to each of its elements a function, returning a list of outputs. Primary intended for tests of independence on a list of contingency tables.

Usage

```
lapplyTest(x, fun = chisq.test, ...)
```

Arguments

`x`	[list] The list to which to apply `fun`.
`fun`	[function] The function which to apply to `data`. Must return an object containing an element named `p.value`. Defaults to `chisq.test`.
`...`	Additional arguments passed to `fun`.

Details

When applying a function to a list, any iteration that results in an error, breaks the whole loop. This is not always the most convenient behaviour, in particular when the function is a statistical test and the error is to do with sparse data in one of the tables in the list. `lapplyTest` is a wrapper around `base::lapply` which only differs from the original in its treatment of errors. It saves the message associated with the error or warning, but then continues to the next iteration rather than quitting the loop altogether.

Value

list.lapplyTest A list of outputs of `fun`.

See Also

`summary.list.lapplyTest`, `allCooccs`

Examples

```
dataset <- loadSampleDataset ("data-abc")
lapplyTest (allCooccs(dataset))
lapplyTest (allCooccs(dataset), fisher.test, simulate.p.value=TRUE)
```

`loadSampleDataset` *Load one of `soundcorrs`' sample datasets.*

Description

Retrieve and return one the sample datasets included in `soundcorrs`.

Usage

```
loadSampleDataset(x)
```

Arguments

`x`	[character] Name of the dataset to load. Available sets are: `soundchange`'s: `change-dl2l`, `change-palatalization`, `change-rhotacism`; `soundcorrs`'s: `data-abc`, `data-capitals`, `data-ie`; and `transcription`'s: `trans-common`, `trans-ipa`.

Details

R does not allow non-ASCII characters in preloaded datasets, and linguistic datasets can hardly fit within ASCII. Unicode is, however, allowed in raw data files. They cannot be automatically loaded when `soundcorrs` is attached because staged install makes it impossible to use `system.file` in this manner, and they cannot be included as a Unicode-escaped output of `dput` because Windows does not know how to convert this to its native encoding. This function makes the process of loading as painless as possible.

Value

soundchange/soundcorrs/transcription The selected sample dataset.

Examples

```
loadSampleDataset ("data-abc")
loadSampleDataset ("trans-ipa")
loadSampleDataset ("change-palatalization")
```

long2wide — *Convert from the long format (single entry per row) to the wide format (multiple entries per row).*

Description

Takes a data frame of word pairs/triples/..., each stored in multiple rows, and returns a data frame with the same words but each pair/triple/... stored in one row. WARNING: in the original data frame, entries from all languages must be in the same order.

Usage

```
long2wide(data, col.lang = "LANGUAGE", skip = NULL)
```

Arguments

`data`	[data.frame] The dataset to be converted.
`col.lang`	[character] Name of the column with language names. Defaults to `"LANGUAGE"`.
`skip`	[character vector] Names of columns to not convert. Defaults to `NULL`.

Details

Data for `soundcorrs` can be prepared in one of two formats: the 'long format' and the 'wide format'. In the 'long format', each row contains only a single word and metadata associated with it. In the 'wide format', each row contains the entire pair/triple/... of words, and all the metadata associated with them. The 'long format' is convenient for making sure that all the words in a pair/triple/... have the same number of segments, but it cannot be read directly by `soundcorrs`. `long2wide` and `wide2long` convert between the two formats.

Value

data.frame A data frame in the wide format (multiple entries per row).

See Also

`wide2long`

Examples

```
# path to sample data in the "long format"
fName <- system.file ("extdata", "data-abc.tsv", package="soundcorrs")
long <- read.table (fName, header=TRUE)
wide <- long2wide (long, skip=c("ID"))
```

merge.soundcorrs *Merge two or more soundcorrs objects.*

Description

Take multiple soundcorrs objects and combine them into one.

Usage

```
## S3 method for class 'soundcorrs'
merge(...)
```

Arguments

...	[soundcorrs] Objects to be merged.

Details

Data can be turned into a soundcorrs object using either read.soundcorrs or, the less preferred method, the raw soundcorrs constructor. However, both can only produce soundcorrs objects with only the data for a single language in them, whereas the typical usage of the soundcorrs package would require it to hold data for several languages simultaneously. This function can be used to safely combine multiple soundcorrs objects into one. The individual objects can all hold data for one or more languages, the only requirement being that the data from the different languages are compatible with each other, i.e. that they have the same number of words, and each word has the same number of segments as its counterparts in the pair/triple/.... An error will be also thrown if two or more of the datasets contain a column with the same name and different content, or when they contain two or more rows with identical content.

Value

soundcorrs The single, merged object.

Examples

```
# path to sample data in the "wide format"
fNameData <- system.file ("extdata", "data-capitals.tsv", package="soundcorrs")
# path to a sample transcription
fNameTrans <- system.file ("extdata", "trans-common.tsv", package="soundcorrs")
ger <- read.soundcorrs (fNameData, "German", "ALIGNED.German", fNameTrans)
pol <- read.soundcorrs (fNameData, "Polish", "ALIGNED.Polish", fNameTrans)
merge (ger, pol)
```

multiFit *Fit multiple models to one dataset.*

Description

Apply a fitting function, with multiple models and multiple starting estimates, to one dataset.

Usage

```
multiFit(models, data, fun = nls, ...)
```

Arguments

`models`	[list] A list of models to fit `data` to. Each element must be a list with at least two named fields: `formula` which contains the formula, and `start` which is a list of lists of starting estimates.
`data`	[numeric data.frame/list] A list of vectors to fit `models` to.
`fun`	[function] The function to use for fitting. Defaults to `nls`.
`...`	Additional arguments passed to `fun`.

Details

Finding the right model and the right starting estimates for a model is often a time consuming process, very inconvenient to do manually. This function automates it as much as possible. It takes a list of models and starting estimates, and fits them to data not stopping whenever an error occurs or a warning is issued. Error and warning messages are saved and can be inspected in the output, they just do not halt the process. `multiFit` has an extension in the form of `fitTable` which applies multiple models to multiple datasets.

Value

list.multiFit A list of results returned by `fun` or, if it ended with an error, `NULL`.

See Also

`fitTable`, `summary.list.multiFit`

Examples

```
set.seed (27)
dataset <- data.frame (X=1:10, Y=(1:10)^2+runif(10,-10,10))
models <- list (
"model A" = list (
formula = "Y ~ X^a",
start = list (list(a=100), list(a=1))),
"model B" = list (
formula = "Y ~ a*(X+b)",
start = list (list(a=1,b=1)))
)
multiFit (models, dataset)
```

`ngrams` *N-grams and their frequencies.*

Description

Find n-grams of specified length and return them as a list, or their counts as a table.

Usage

```
ngrams(x, n = 1, borders = c("", ""), rm = "", as.table = T)
```

Arguments

`x`	[character vector] Words to be cut into n-grams.
`n`	[integer] The length of n-grams to look for. Defaults to `1`.
`borders`	[character] Characters to prepend and append to every word. Must be a vector of exactly two character strings. Defaults to `c("","")`.
`rm`	[character] Characters to be removed from `x` before cutting into n-grams. May be a regular expression, f.ex. "[-|]" will capture the default symbol for linguistics zeros as well as the default segment separators. Empty string denotes nothing to replace. Defaults to empty string.
`as.table`	[logical] Return the result as a table? Defaults to `TRUE`.

Details

Data processed with `soundcorrs` are generally expected to be segmented and aligned, and both segmentation and alignment are recommended to be performed manually. This is a laborious process, but it is feasible when segments represent morphemes or phonemes. Should segments represent n-grams, however, the fully manual approach would have been very time consuming and prone to errors.

Value

table Table with counts of n-grams.

Examples

```
dataset <- loadSampleDataset ("data-capitals")
ngrams(dataset$data[,"ALIGNED.German"], n=2)
ngrams(dataset$data[,"ALIGNED.German"], n=3, as.table=FALSE)
ngrams(dataset$data[,"ALIGNED.German"], n=4, rm="[-\\|]", as.table=FALSE)
ngrams(dataset$data[,"ALIGNED.German"], n=5, borders=c(">","<"), rm="[-\\|]", as.table=FALSE)
```

`ngrams.scOne` *Information that the* `scOne` *class is obsolete.*

Description

Since version 0.2.0 it has been replaced with the `soundcorrs` class.

Usage

```
ngrams.scOne(...)
```

Arguments

`...`	Ignored, only for compatibilty.

`print.df.findExamples` *Pretty printing for the result of* `findExamples`.

Description

Pretty printing for the result of `findExamples`.

Usage

```
## S3 method for class 'df.findExamples'
print(x, ...)
```

Arguments

`x`	[df.findExamples] The output of `findExamples`.
`...`	Unused; only for consistency with `print`.

Details

The output of `findExamples` is a list, potentially a very long one, and difficult to read. To make it easier to digest, this function only prints the `$data` element, i.e. the found matches.

Value

df.findExamples The same object that was given as `x`.

See Also

`findExamples`

Examples

```
dataset <- loadSampleDataset ("data-capitals")
findExamples (dataset, "a", "a", "a", cols="all")
```

---

print.list.applyChanges

Pretty printing for the result of `applyChanges`.

Description

Pretty printing for the result of `applyChanges`.

Usage

```
## S3 method for class 'list.applyChanges'
print(x, ...)
```

Arguments

`x`	[list.applyChanges] The output of `applyChanges`.
`...`	Unused; only for consistency with `print`.

Details

The output of `applyChanges` is a list, potentially a very long one, and difficult to read. To make it easier to digest, this function only prints the `$end` element, i.e. the final shapes produced by the application of all of the sound changes.

Value

list.applyChanges The same object that was given as `x`.

See Also

`applyChanges`, `print.tree.applyChanges`

Examples

```
# prepare sample data
dataset <- loadSampleDataset ("data-capitals")
changes <- list (loadSampleDataset("change-dl2l"), loadSampleDataset("change-rhotacism"))
# apply the changes
applyChanges (dataset, changes, "ORTHOGRAPHY.German")
applyChanges (dataset, changes, "ORTHOGRAPHY.German", "ORTHOGRAPHY.Polish", highlight="console")
```

`print.scOne` *Information that the* `scOne` *class is obsolete.*

Description

Since version 0.2.0 it has been replaced with the `soundcorrs` class.

Usage

```
## S3 method for class 'scOne'
print(...)
```

Arguments

`...`	Ignored, only for compatibilty.

`print.soundchange` *A more reasonable display of a* `soundchange` *object.*

Description

A more reasonable display of a `soundchange` object.

Usage

```
## S3 method for class 'soundchange'
print(x, ...)
```

Arguments

`x`	[soundchange] The `soundchange` object.
`...`	Unused; only for consistency with `print`.

Details

The structure of a `soundchange` object is probably not too dificult to read for a human, but this does not mean it cannot be presented in a slightly more convenient form.

Value

soundchange The same object that was given as `x`

See Also

`soundchange`

Examples

```
# prepare sample transcription
trans <- loadSampleDataset ("trans-common")
# run print.soundchange
a2b <- soundchange ("a > b", "sample change", trans,
"This is a very simple sample sound change whereby \"a\" is turned into \"b\".")
a2b
```

print.soundcorrs *A more reasonable display of a soundcorrs object.*

Description

A more reasonable display of a soundcorrs object.

Usage

```
## S3 method for class 'soundcorrs'
print(x, ...)
```

Arguments

x	[soundcorrs] The soundcorrs object.
...	Unused; only for consistency with print.

Details

A soundcorrs may be quite large and therefore difficult to digest for a human. This function reduces it to a brief, easy to understand summary.

Value

soundcorrs The same object that was given as x.

See Also

soundcorrs

Examples

```
dataset <- loadSampleDataset ("data-abc")
dataset
```

print.transcription *A more reasonable display of a transcription object.*

Description

A more reasonable display of a transcription object.

Usage

```
## S3 method for class 'transcription'
print(x, ...)
```

Arguments

x	[transcription] The transcription object.
...	Unused; only for consistency with print.

Details

A transcription object may be quite large and therefore difficult to digest for a human. This function reduces it to a brief, easy to understand summary.

Value

transcription The same object that was given as x.

See Also

transcription

Examples

```
# path to a sample transcription
fName <- system.file ("extdata", "trans-common.tsv", package="soundcorrs")
read.transcription (fName)
```

`print.tree.applyChanges` *Pretty printing for part of the result of* `applyChanges`.

Description

Pretty printing for part of the result of `applyChanges`.

Usage

```
## S3 method for class 'tree.applyChanges'
print(x, ...)
```

Arguments

`x`	[tree.applyChanges] The `tree` element in the output of `applyChanges`.
`...`	Unused; only for consistency with `print`.

Details

One of the elements in the output of `applyChanges` is a tree. It is represented as a nested list, potentially a very deeply nested and a very long list which would have been all but impossible to digest for a human. This function prints it as a structure that more resembles a tree, very similar to the output of `str`.

Value

tree.applyChanges The same object that was given as `x`.

See Also

`applyChanges`, `print.list.applyChanges`

Examples

```
# prepare sample data
dataset <- loadSampleDataset ("data-capitals")
changes <- list (loadSampleDataset("change-dl2l"), loadSampleDataset("change-rhotacism"))
# apply the changes
applyChanges (dataset, changes, "ORTHOGRAPHY.German")$tree
```

`read.scOne` *Information that the* `scOne` *class is obsolete.*

Description

Since version 0.2.0 it has been replaced with the `soundcorrs` class.

Usage

```
read.scOne(...)
```

Arguments

`...`	Ignored, only for compatibilty.

`read.soundcorrs` *Read data for a single language from a tsv file.*

Description

Read the data for one language, from a file in the wide format, and combine it with metadata into a `soundcorrs` object. To obtain a `soundcorrs` object containing data for multiple languages, see `merge.soundcorrs`.

Usage

```
read.soundcorrs(file, name, col.aligned, transcription, separator = "\\|")
```

Arguments

`file`	[character] Path to the data file in the wide format.
`name`	[character] Name of the language.
`col.aligned`	[character] Name of the column with the aligned words.
`transcription`	[character] Path to the file with the transcription.
`separator`	[character] String used to separate segments in `col.aligned`. Defaults to `"|"`.

Details

The constructor for the `soundcorrs` class requires a data frame and a `transcription` object which means that the user would need to first read both from a file, and only then pass them to the constructor. This function saves these two steps. In addition, it attaches the name of the file to the object, which allows for easier identification later. It is recommended to use `read.soundcorrs` instead of the raw `soundcorrs` constructor whenever possible.

Value

soundcorrs	An object containing the data and metadata for one language.

See Also

soundcorrs

Examples

```
# path to sample data in the "wide format"
fNameData <- system.file ("extdata", "data-ie.tsv", package="soundcorrs")
# path to a sample transcription
fNameTrans <- system.file ("extdata", "trans-common.tsv", package="soundcorrs")
ger <- read.soundcorrs (fNameData, "Latin", "LATIN", fNameTrans)
```

---

read.transcription *Read transcription from a tsv file.*

Description

Read a table from file and create a `transcription` object out of it.

Usage

```
read.transcription(
  file,
  col.grapheme = "GRAPHEME",
  col.meta = "META",
  col.value = "VALUE"
)
```

Arguments

`file`	[character] Path to the data file.
`col.grapheme`	[character] Name of the column with graphemes. Defaults to `"GRAPHEME"`.
`col.meta`	[character] Name of the column with the coverage of metacharacters. If empty string or NA, the column will be generated automatically. Defaults to `"META"`.
`col.value`	[character] Name of the column with values of graphemes. Defaults to `"VALUE"`.

Details

The constructor for the `transcription` class requires a data frame which means that the user would need to first read it from a file, and only then pass it to the constructor. This function saves this one step. In addition, it attaches the name of the file to the object, which allows for easier identification later. It is recommended to use `read.transcription` instead of the raw `transcription` constructor whenever possible.

Value

transcription	A transcription object containing the read transcription.

See Also

`transcription`

Examples

```
# path to a sample transcription
fName <- system.file ("extdata", "trans-common.tsv", package="soundcorrs")
read.transcription (fName)
```

scOne — *Information that the* `scOne` *class is obsolete.*

Description

Since version 0.2.0 it has been replaced with the `soundcorrs` class.

Usage

```
scOne(...)
```

Arguments

`...`	Ignored, only for compatibilty.

soundchange — *Constructor function for the* `soundchange` *class.*

Description

Either takes a sound change, translates it to a function, and wraps that into a soundchange object, or takes a function and wraps it into a soundchange object.

Usage

```
soundchange(x, name, transcription, desc = NULL, perl = F)
```

Arguments

`x`	[character/function] Either a sound change, or a function that takes as arguments a character string named `x` and an object named `meta`, and returns a vector of character strings.
`name`	[character] Name of the sound change. Length limit is not enforced, but it is recommended to not exceed 30 characters.
`transcription`	[transcription] The `transcription` used in the notation of the sound change.
`desc`	[character] A description of the change. May be as short or as long as needed. Defaults to `NULL`.
`perl`	[logical] Use Perl-compatible regular expressions? This argument is only used when `x` is a character string. Defaults to `FALSE`.

Details

A `soundchange` object is basically a sound change function with some metadata: name, description, and associated transcription. The sound change function can be a simple substitution using `gsub`, or as complex as required. It must take two arguments: `x` which is a single character string to which the change is to be applied, and `meta` which can be any object that holds additional metadata; `meta` can also be `NULL`. The return value of a sound change function must be a vector of character strings. The ability to output multiple strings is primarily intended for reconstruction, and it is not recommended that it be used to fit multiple changes into a single function. Example: in language L, "*a" and "*e" merged to "a", and simultaneously *o, *u > o. Regardless of whether our functions depart from the current state and reconstruct proto-forms, or the opposite, two functions should be defined: one for the a-e merger, and one for the o-u merger. The difference between them will be that with a progressive change, each function will return only one value ("[ae] > a") whereas in a regressive one it will return two ("a > a, e").

When the sound change is given as a simple character string, rather than a complete function, the string must contain exactly one ">" or "<" sign. Spaces around it are ignored. The string may take full advantage of regular expressions, both the ones available in pure R, and custom ones defined by the user and accepted by `expandMeta`. Backreferences may be of particular usefulness. In short, any part of the 'find' string that is enclosed in brackets, is also available in the 'replace' string as `"\\1"`, `"\\2"`, etc. For example, if k's followed by e or i all change into s's, this process cannot be encoded as simply *`"k[ei] > s"` because this would result in the e/i being deleted. The correct notation would be `"k([ei]) > s\\1"`. However, see the caveat below.

Warning! When the META column is missing from the transcription and generated automatically, the alternatives are listed using round- rather than square-bracket notation, i.e. as `"(x|y)"` rather than `"[xy]"`. This is necessary because some graphemes may be longer than one character, but a side effect of this is that when a user-defined metacharacter ("wild-card") is used in the 'find' part of the sound change string (the part before `"<"`), it adds a set of round brackets to that part of the string. In turn, this means that backreferences in the 'replace' part of the sound change string must either be shifted accordingly, or round brackets around metacharacters in the 'find' part omitted. For example, if intervocalic s is to be replaced with r, the sound change string should be `"VsV > \\1r\\2"` rather than *`"(V)s(V) > \\1r\\2"` because `"V"` itself is already translated to `"(a|ä|e|. . . "`.

Note that sound changes and their application do not require the data to be segmented and aligned. If sound changes are the only goal of the project, these two time-consuming steps can be safely omitted.

Value

soundchange	An object containing the provided data.

Fields

`fun` [function] The sound change function.

`desc` [character] A description of the change. May be `NULL`.

`name` [character] The name of the change.

See Also

`print.soundchange`, `applyChanges`

Examples

```
# prepare sample transcription
trans <- loadSampleDataset ("trans-common")
# run soundchange
a2b <- soundchange ("a > b", "sample change", trans)
a2b <- soundchange ("b < a", "sample change", trans)
a2b <- soundchange (function(x,meta) gsub("a","b",x), "sample change", trans)
```

`soundcorrs` *Constructor function for the* `soundcorrs` *class.*

Description

Take a data frame and turn it into a `soundcorrs` object containing data for one language. To obtain a `soundcorrs` object containing data for multiple languages, see `merge.soundcorrs`. In the normal workflow, the user should have no need to call this constructor other than through `read.soundcorrs`.

Usage

```
soundcorrs(data, name, col.aligned, transcription, separator = "\\|")
```

Arguments

`data`	[data.frame] Data for one language.
`name`	[character] Name of the language.
`col.aligned`	[character] Name of the column with the aligned words.
`transcription`	[transcription] The `transcription` for the given language.
`separator`	[character] String used to separate segments in `col.aligned`. Defaults to `"|"`.

Details

`soundcorrs` is the fundamental class of the entire soundcorrs package, and it is required for most tasks that the package promises to make easier and faster than manual labour. A `soundcorrs` object is a list containing the original data frame, some metadata (names of languages, names of columns, transcriptions), as well as transformations of the original data for faster processing in `findExamples` and other functions (words exploded into individual segments, with segment separators removed, etc.). The basic unit in `soundcorrs` is a pair/triple/... of words, each of which is assigned to a specific language.

This constructor function is not really intended for the end user. Whenever possible, `read.soundcorrs` should be used instead. Regardless of the function used, two pieces of information are required for each word: the language it comes from, and its segmented and aligned form. Segmentation means that the word is cut into parts which can represent phonemes, morphemes, or anything else (the default separator is a vertical bar, `"|"`). A word with no separators in it is considered one big segment, and in fact, for `soundchange`'s this is enough. Alignment means that each word in a pair/triple/... has the same number of segments, and that those segments are in the corresponding places. Often, one of the words in a pair/triple/... will naturally have fewer segments than the others; in such cases, a filler character, 'linguistic zero' needs to be used (`"-"` is a good choice); for example, to align the Spanish and Swedish names for 'Stockholm', a total of three such 'empty' segments is required: e|s|t|o|k|-|o|l|m|o : -|s|t|o|k|k|o|l|m|-. Linguistic zero must be defined in the `transcription`.

Typically, a `soundcorrs` object will be used to hold an entire list of pairs/triples/... of words from various languages. However, both this constructor function and `read.soundcorrs` can only read data from one language at a time. This is because each language requires relatively many pieces of metadata (name, column names, transcription), and if all of this information for multiple languages were to be passed as arguments to one function, the call would very quickly become illegible. Multiple `soundcorrs` objects can be merged into one using `merge.soundcorrs`.

Three sample datasets are available: `data-abc`, `data-capitals`, and `data-ie`; they can be loaded with the help of `loadSampleDataset`.

Value

soundcorrs An object containing the provided data and metadata for one language.

Fields

`cols` [character list] Names of important columns.

`data` [data.frame] The original data.

`names` [character] Name of the language.

`segms` [character list] Words exploded into segments. With linguistic zeros preserved (`$z`) or removed (`$nz`).

`segpos` [integer list] A lookup list to check which character belongs to which segment. Counted with linguistic zeros preserved (`$z`) and removed (`$nz`).

`separators` [character] The strings used as segment separator in `cols$aligned`.

`trans` [transcription] The transcription.

`words` [character list] Words obtained by removing separators from the `cols$aligned` columns. With linguistic zeros (`$z`) or without them (`$nz`).

Examples

```
# prepare sample transcription
trans <- loadSampleDataset ("trans-common")
# read sample data in the "wide format"
fNameData <- system.file ("extdata", "data-capitals.tsv", package="soundcorrs")
readData <- read.table (fNameData, header=TRUE)
# make out of them a soundcorrs object
ger <- soundcorrs (readData, "German", "ALIGNED.German", trans)
pol <- soundcorrs (readData, "Polish", "ALIGNED.Polish", trans)
spa <- soundcorrs (readData, "Spanish", "ALIGNED.Spanish", trans)
dataset <- merge (ger, pol, spa)
```

`soundcorrsGUI` *GUI for* `soundcorrs`*.*

Description

A graphic user interface for the `soundcorrs` package.

Usage

```
soundcorrsGUI()
```

Details

The console version of `soundcorrs` is flexible and offers a wide range of possibilities but it also requires a degree of experience with R and programming in general. This object provides an interface to `soundcorrs` which removes a considerable part of this requirement, though it is at the cost of flexibility and power.

When running the app from the console, press ctrl-c to quit the server.

Value

shiny.appobj A `shiny` app object. To run the app, print it or pass it to `shiny::runApp`.

`subset.soundcorrs` *Return a subset of sound correspondences data which meets a condition.*

Description

Reduce a `soundcorrs` object to just those word pairs/triples/... which meet a certain condition.

Usage

```
## S3 method for class 'soundcorrs'
subset(x, condition, ...)
```

Arguments

`x`	[soundcorrs] The dataset to be subsetted.
`condition`	[logical] The condition the subsetted data must meet.
`...`	Unused; only for consistency with `subset`.

Details

Once a data frame is enclosed in a `soundcorrs` object, it is recommended that it not be manually altered in any way. `subset.soundcorrs` provides a safe way of subsetting it.

Value

soundcorrs A soundcorrs object containing the subsetted dataset.

Examples

```
# In the examples below, non-ASCII characters had to be escaped for technical reasons.
# In actual usage, all soundcorrs functions accept characters from beyond ASCII.

dataset <- loadSampleDataset ("data-capitals")
subset (dataset, OFFICIAL.LANGUAGE=="German")
subset (dataset, grepl("German",OFFICIAL.LANGUAGE))
subset (dataset, findExamples(dataset, "\u00E4", "e", "")$which)  # a-diaeresis
```

`summary.list.lapplyTest` *A quick summary of the result of* `lapplyTest`.

Description

Take the output of `lapplyTest`, and extract from it only the noteworthy results.

Usage

```
## S3 method for class 'list.lapplyTest'
summary(object, p.value = 0.05, ...)
```

Arguments

`object`	[list.lapplyTest] The output of `lapplyTest`.
`p.value`	[double] Results above this value will not be reported. Defaults to 0.05.
`...`	Unused; only for consistency with `summary`.

Details

The output of `lapplyTest` may be difficult to digest for a human. This function selects from it only the results that are of particular interest, and presents them in an easy to read form.

Value

A more human-friendly digest.

See Also

lapplyTest

Examples

```
dataset <- loadSampleDataset ("data-abc")
lapplyTest (allCooccs(dataset))
```

---

summary.list.multiFit *A comparison of the results produced by fitTable or multiFit.*

Description

Take the output of fitTable or multiFit, extract a specific metric from the fits, and present them in the form of a table.

Usage

```
## S3 method for class 'list.multiFit'
summary(object, metric = "rss", ...)
```

Arguments

object	[list.multiFit] The output of fitTable or multiFit.
metric	[character] The metric to extract from object. Available metrics are: "aic", "bic", "rss", and "sigma". Defaults to "rss".
...	Unused; only for consistency with summary.

Details

The output of multiFit may be difficult to digest for a human. This function selects from it only the results that are of particular interest, and presents them in an easy to read form.

Value

A more human-friendly digest.

See Also

multiFit

Examples

```
set.seed (27)
dataset <- data.frame (X=1:10, Y=(1:10)^2+runif(10,-10,10))
models <- list (
"model A" = list (
formula = "Y ~ X^a",
start = list (list(a=100), list(a=1))),
"model B" = list (
formula = "Y ~ a*(X+b)",
start = list (list(a=1,b=1)))
)
summary (multiFit(models,dataset))
summary (fitTable(models,as.matrix(dataset),1,vec2df.rank), "sigma")
```

`summary.soundcorrs` *Generate a segment-to-segment contingency table for two languages.*

Description

Produce a contingency table detailing all segment-to-segment correspondences in a dataset.

Usage

```
## S3 method for class 'soundcorrs'
summary(object, count = "a", unit = "w", ...)
```

Arguments

`object`	[soundcorrs] The dataset from which to draw frequencies. Only datasets with two languages are supported.
`count`	[character] Report either the absolute or the relative numbers?. Accepted values are `"a(bs(olute))"` and `"r(el(ative))"`. Defaults to "a".
`unit`	[character] Count how many times a correspondence occurs or in how many words it occurs? Accepted values are `"o(cc(ur(ence(s))))"` and `"w(or(d(s)))"`. Defaults to `"w"`.
`...`	Unused; only for consistency with `base::summary`.

Details

A set of segmented and aligned word pairs/triples/..., such as one held in a `soundcorrs` object, can be turned into a contingency table in more than one way. Perhaps the simplest option is to see how often various segments from one language correspond to various segments from another language, which is the kind of table this function produces. Correspondences can be reported in absolute or relative numbers, and can represent the number of times the given correspondence occurs, or in how many words it occurs (the same correspondence can occur more than once in a single pair/triple/... of words, e.g. in German koala : French koala, the correspondence G a : Fr a occurs twice). When the numbers are relative, each row in the table adds up to 1. In theory, `summary.soundcorrs` can support a `soundcorrs` objects with any number of languages in it, but the legibility of the output drops very quickly when that number exceeds two.

Value

table	The contingency table. The values represent how often the given segments correspond to each other, not how often they co-occur in the same word (cf. `coocc`).

See Also

`coocc`

Examples

```
dataset <- loadSampleDataset ("data-ie")
summary (dataset)
round (summary(dataset,count="r"), digits=3)
summary (dataset, unit="o")
```

`transcription` *Constructor function for the* `transcription` *class.*

Description

Take a data frame containing transcription and turn it into a `transcription` object, as required by the `soundcorrs` constructor function. In the normal workflow, the user should have no need to call this function other than through `read.transcription`.

Usage

```
transcription(
  data,
  col.grapheme = "GRAPHEME",
  col.meta = "META",
  col.value = "VALUE"
)
```

Arguments

`data`	[data.frame] Data frame containing the transcription and its meaning.
`col.grapheme`	[character] Name of the column with graphemes. Defaults to `"GRAPHEME"`.
`col.meta`	[character] Name of the column with the coverage of metacharacters. If empty string or `NA`, the column will be generated automatically. Defaults to `"META"`.
`col.value`	[character] Name of the column with values of graphemes. Defaults to `"VALUE"`.

Details

The primary reason why transcription needs to be defined, are regular expressions. R has a powerful system of regular expressions but they are general, not designed specifically for use in linguistics. Linguistics has its own convention of regular expressions, or rather two conventions, and to emulate them, it is necessary for `soundcorrs` to know the linguistic value of individual graphemes. One convention is the traditional, 'European' one where typically single characters represent entire classes of sounds, e.g. "C" stands for 'any consonant', "A" for 'any back vowel', etc. The other convention is the 'binary', 'American' notation where instead of using single characters, one lists all the distinctive features, e.g. "[+cons]" or "[+vowel,+back]". Having the values of graphemes encoded in a `transcription` object, `expandMeta` is able to translate these two notations into regular expressions that R can understand.

This constructor function is not really intended for the end user. Whenever possible, `read.transcription` should be used instead. Regardless of the function used, a data frame with two columns is required in order to create a `transcription` object: one column for the graphemes, and one for their values. It is probably not necessary, but nevertheless recommended, just to be on the safe side, that graphemes be single characters. (This also excludes combining diacritical marks.) Values must be separated by commas, without spaces. Typically, they will be phonetic features, but in principle they can be anything. A transcription may also have a third column that holds the string that the given grapheme is going to be turned into by `expandMeta`. Regular graphemes should be simply repeated in this column, whereas metacharacters (such as "C" or "A" mentioned above) should be expanded into all the graphemes they represent, separated by a bar ("|"), and enclosed in brackets, e.g. "(a|o|u)". If the third column is missing, this function will generate it automatically. Note, however, that the generation is based on the value column, and any grapheme whose value is a subset of the value of another grapheme, will be considered a metacharacter. For example, if "p" is defined as "cons,stop,blab", and "b" as "cons,stop,blab,voiced", "p" will be considered a metacharacter for both "p" and "b", and translated into "(p|b)" by `expandMeta`.

Graphemes cannot contain in them characters reserved for regular expressions: . + * ^ \ $? | ( ) [ ] { }, and they also cannot contain in them characters defined as metacharacters in the transcription. For example, if "A" is defined as "vowel,back", and therefore represents all the back vowels in the transcription, a regular grapheme "A:" is forbidden. A metacharacter "A:", on the other hand, is permitted (e.g. for 'any long back vowel'), though it is recommended that such overlapping metacharacters be avoided as much as possible.

Lastly, a transcription must contain so-called linguistic zero. This is a character which signifies an empty segment in a word, a segment which has been only added in order to align the segments in all the words in a pair/triple/.... For example, English passport has two phonemes fewer than Spanish pasaporte id., so in order for the two words to be aligned, the English one needs two filler segments:p|a|s|-|p|o|r|t|- : p|a|s|a|p|o|r|t|e. To designate a character as linguistic zero in a transcription, its value must be `"NULL"`.

Two sample transcriptions are available: `trans-common`, `trans-ipa`; they can be loaded with the help of `loadSampleDataset`.

Value

transcription	An object containing the provided data.

Fields

`data` [data.frame] The original data frame.

`cols` [character list] Names of the important columns in the data frame.

`meta` [character] A vector of character strings which act as metacharacters in regular expressions. Mostly useful to speed up `expandMeta`.

`values` [character] A named list with values of individual graphemes exploded into vectors.

`zero` [character] A regular expression to catch linguistic zeros.

See Also

`link{expandMeta}`, `print.transcription`, `read.transcription`

Examples

```
# path to a sample transcription
fName <- system.file ("extdata", "trans-common.tsv", package="soundcorrs")
fut <- transcription (read.table(fName,header=TRUE))
```

`vec2df.hist` *A vector to data frame converter for* `fitTable`*. This one makes a histogram, and returns a data frame with midpoints and counts.*

Description

A vector to data frame converter for `fitTable`. This one makes a histogram, and returns a data frame with midpoints and counts.

Usage

```
vec2df.hist(data)
```

Arguments

`data`	[numeric vector] The data to be converted.

Value

data.frame Converted `data`.

Examples

```
# prepare sample dataset
fTrans <- system.file ("extdata", "trans-common.tsv", package="soundcorrs")
fData <- system.file ("extdata", "data-capitals.tsv", package="soundcorrs")
tmp.ger <- read.soundcorrs (fData, "German", "ALIGNED.German", fTrans)
tmp.pol <- read.soundcorrs (fData, "Polish", "ALIGNED.Polish", fTrans)
dataset <- merge (tmp.ger, tmp.pol)
# prepare and run fitTable
models <- list (
"model A" = list (
formula = "Y ~ a/X",
start = list (list(a=1))),
"model B" = list (
formula = "Y ~ a/(1+exp(1)^X)",
start = list (list(a=1)))
)
fitTable (models, summary(dataset), 1, vec2df.hist)
```

vec2df.id

A vector to data frame converter for `fitTable`. This one only does the necessary minimum.

Description

A vector to data frame converter for `fitTable`. This one only does the necessary minimum.

Usage

```
vec2df.id(data)
```

Arguments

`data`	[numeric vector] The data to be converted.

Value

data.frame Converted `data`.

Examples

```
# prepare sample dataset
fTrans <- system.file ("extdata", "trans-common.tsv", package="soundcorrs")
fData <- system.file ("extdata", "data-capitals.tsv", package="soundcorrs")
tmp.ger <- read.soundcorrs (fData, "German", "ALIGNED.German", fTrans)
tmp.pol <- read.soundcorrs (fData, "Polish", "ALIGNED.Polish", fTrans)
dataset <- merge (tmp.ger, tmp.pol)
# prepare and run fitTable
models <- list (
"model A" = list (
formula = "Y ~ a/X",
start = list (list(a=1))),
"model B" = list (
formula = "Y ~ a/(1+exp(1)^X)",
start = list (list(a=1)))
)
fitTable (models, summary(dataset), 1, vec2df.id)
```

vec2df.rank *A vector to data frame converter for `fitTable`. This one orders data by rank.*

Description

A vector to data frame converter for `fitTable`. This one orders data by rank.

Usage

```
vec2df.rank(data)
```

Arguments

`data`	[numeric vector] The data to be converted.

Value

data.frame Converted `data`.

Examples

```
# prepare sample dataset
fTrans <- system.file ("extdata", "trans-common.tsv", package="soundcorrs")
fData <- system.file ("extdata", "data-capitals.tsv", package="soundcorrs")
tmp.ger <- read.soundcorrs (fData, "German", "ALIGNED.German", fTrans)
tmp.pol <- read.soundcorrs (fData, "Polish", "ALIGNED.Polish", fTrans)
dataset <- merge (tmp.ger, tmp.pol)
# prepare and run fitTable
models <- list (
"model A" = list (
formula = "Y ~ a/X",
start = list (list(a=1))),
"model B" = list (
formula = "Y ~ a/(1+exp(1)^X)",
start = list (list(a=1)))
)
fitTable (models, summary(dataset), 1, vec2df.rank)
```

wide2long — *Convert from the wide format (multiple entries per row) to the long format (single entry per row).*

Description

Takes a data frame of word pairs/triples/..., each stored in a single row, and returns a data frame with the same pairs/triples/... but with each word stored in its own row.

Usage

```
wide2long(data, suffixes, col.lang = "LANGUAGE", strip = 0)
```

Arguments

`data`	[data.frame] The dataset to be converted.
`suffixes`	[character vector] Suffixes used to differentiate column names; in the output, those will be used as language names.
`col.lang`	[character] Name of the column in which language names are to be stored. Defaults to `"LANGUAGE"`.
`strip`	[integer] The number of characters to strip from the beginning of suffixes when they are turned into language names. Defaults to 0.

Details

Data for `soundcorrs` can be prepared in one of two formats: the 'long format' and the 'wide format'. In the 'long format', each row contains only a single word and metadata associated with it. In the 'wide format', each row contains the entire pair/triple/... of words, and all the metadata associated with them. The 'long format' is convenient for making sure that all the words in a pair/triple/... have the same number of segments, but it cannot be read directly by `soundcorrs`. `long2wide` and `wide2long` convert between the two formats.

Value

data.frame A data frame in the long format (single entry per row).

See Also

`long2wide`

Examples

```
# path to sample data in the "wide format"
fName <- system.file ("extdata", "data-capitals.tsv", package="soundcorrs")
wide <- read.table (fName, header=TRUE)
long <- wide2long (wide, c(".German",".Polish",".Spanish"), strip=1)
```

%hasPrefix% *Check if a string starts with another string.*

Description

Within soundcorrs, primarily intended to extract rows and columns from contingency tables. Other than that, of general applicability.

Usage

```
x %hasPrefix% prefix
```

Arguments

x	[character] The string or strings in which to look.
prefix	[character] The string to look for. May be a regular expression.

Details

Contingency tables, such as produced by coocc and summary, can get quite sizeable and therefore difficult to read with larger datasets. Since both their column and row names are composed from individual segments connected by an underscore ("_"), %hasPrefix% offers an easy way to select the interesting bit of the table by the first segment.

Value

logical TRUE iff x begins with prefix.

See Also

%hasSuffix%

Examples

```
"loans.tsv" %hasPrefix% "loans"
c("abc","bbc","cbc") %hasPrefix% "[bc]"
```

%hasSuffix% *Check if a string ends in another string.*

Description

Within soundcorrs, primarily intended to extract rows and columns from contingency tables. Other than that, of general applicability.

Usage

```
x %hasSuffix% suffix
```

Arguments

x	[character] The string or strings in which to look.
suffix	[character] The string to look for. May be a regular expression.

Details

Contingency tables, such as produced by coocc and summary, can get quite sizeable and therefore difficult to read with larger datasets. Since both their column and row names are composed from individual segments connected by an underscore ("_"), %hasSuffix% offers an easy way to select the interesting bit of the table by the last segment.

Value

logical TRUE iff x ends with suffix.

See Also

%hasPrefix%

Examples

```
"loans.tsv" %hasSuffix% ".tsv"
c("aba","abb","abc") %hasSuffix% "[bc]"
```

Index