Package 'acss'

July 22, 2025

Type Package

Title Algorithmic Complexity for Short Strings

Version 0.3-2

Imports zoo

Depends R (>= 2.15.0), acss.data

Suggests effects, lattice, knitr, rmarkdown

LazyData true

Description Main functionality is to provide the algorithmic complexity for short strings, an approximation of the Kolmogorov Complexity of a short string using the coding theorem method (see ?acss). The database containing the complexity is provided in the data only package acss.data, this package provides functions accessing the data such as prob_random returning the posterior probability that a given string was produced by a random process. In addition, two traditional (but problematic) measures of complexity are also provided: entropy and change complexity.

URL https://complexity-calculator.com/methodology.html

License GPL (>= 2)

RoxygenNote 7.3.2

Encoding UTF-8

VignetteBuilder knitr

NeedsCompilation no

Author Nicolas Gauvrit [aut],
Henrik Singmann [aut, cre],
Fernando Soler Toscano [ctb],
Hector Zenil [ctb]

Maintainer Henrik Singmann <singmann+acss@gmail.com>

Repository CRAN

Date/Publication 2025-05-21 14:40:01 UTC

Contents

`acss` *ACSS complexity*

Description

Functions to obtain the algorithmic complexity for short strings, an approximation of the Kolmogorov Complexity of a short string using the coding theorem method.

Usage

```
acss(string, alphabet = 9)

local_complexity(string, alphabet = 9, span = 5)

likelihood_d(string, alphabet = 9)

likelihood_ratio(string, alphabet = 9)

prob_random(string, alphabet = 9, prior= 0.5)
```

Arguments

`string`	`character` vector containing the to be analyzed strings (can contain multiple strings).
`alphabet`	`numeric`, the number of possible symbols (not necessarily actually appearing in str). Must be one of `c(2, 4, 5, 6, 9)` (can also be `NULL` or contain multiple values for `acss()`). Default is 9.
`prior`	`numeric`, the prior probability that the underlying process is random.
`span`	size of substrings to be created from `string`.

Details

The algorithmic complexity is computed using the coding theorem method: For a given alphabet size (number of different symbols in a string), all possible or a large number of random samples of Turing machines (TM) with a given number of states (e.g., 5) and number of symbols corresponding to the alphabet size were simulated until they reached a halting state or failed to end. The outputs of the TMs at the halting states produces a distribution of strings known as the algorithmic probability of the strings. The algorithmic coding theorem (Levin, 1974) establishes the connection between the complexity of a string $K(s)$ and its algorithmic probability $D(s)$ as:

$$K(s) \approx -\log_2(D(s))$$

This package accesses a database containing data on 4.5 million strings from length 1 to 12 simulated on TMs with 2, 4, 5, 6, and 9 symbols.

For a more detailed discussion see Gauvrit, Singmann, Soler-Toscano, and Zenil (2014), https://complexity-calculator.com/methodology.html, or references below.

Value

"acss" A matrix in which the rows correspond to the strings entered and the columns to the algorithmic complexity K and the algorithmic probability D of the string (see https://complexity-calculator.com/methodology.html).

"local_complexity" A list with elements corresponding to the strings. Each list containes a named vector of algorithmic complexities (K) of all substrings in each string with length span.

"likelihood_d" A named vector with the likelihoods for `string` given a detreministic process.

"likelihood_ratio" A named vector with the likelihood ratios (or Bayes factors) for `string` given a random rather than detreministic process.

"prob_random" A named vector with the posterior probabilities that for a random process given the strings and the provided prior for being produced by a random process (default is 0.5, which correspond to a prior of 1 - 0.5 = 0.5 for a detereministic process).

Note

The first time per session one of the functions described here is used, a relatively large dataset is loaded into memory which can take a considerable amount of time (> 10 seconds).

References

Delahaye, J.-P., & Zenil, H. (2012). Numerical evaluation of algorithmic complexity for short strings: A glance into the innermost structure of randomness. *Applied Mathematics and Computation*, 219(1), 63-77. doi:10.1016/j.amc.2011.10.006

Gauvrit, N., Singmann, H., Soler-Toscano, F., & Zenil, H. (2014). Algorithmic complexity for psychology: A user-friendly implementation of the coding theorem method. arXiv:1409.4080 [cs, stat]. http://arxiv.org/abs/1409.4080.

Gauvrit, N., Zenil, H., Delahaye, J.-P., & Soler-Toscano, F. (2014). Algorithmic complexity for short binary strings applied to psychology: a primer. *Behavior Research Methods*, 46(3), 732-744. doi:10.3758/s13428-013-0416-0

Levin, L. A. (1974). Laws of information conservation (nongrowth) and aspects of the foundation of probability theory. *Problemy Peredachi Informatsii*, 10(3), 30-35.

Soler-Toscano, F., Zenil, H., Delahaye, J.-P., & Gauvrit, N. (2012). Calculating Kolmogorov Complexity from the Output Frequency Distributions of Small Turing Machines. *PLoS ONE*, 9(5): e96223.

Examples

```
# WARNING: The first call to one of the functions
# discussed on this page loads a large data set
# and usually takes > 10 seconds. Stay patient.

acss(c("HEHHEE", "GHHGGHGHH", "HSHSHHSHSS"))
##                 K.9          D.9
## HEHHEE     23.38852 9.106564e-08
## GHHGGHGHH  33.50168 8.222205e-11
## HSHSHHSHSS 35.15241 2.618613e-11

acss(c("HEHHEE", "GHHGGHGHH", "HSHSHHSHSS"))[,"K.9"]
## [1] 23.38852 33.50168 35.15241

acss(c("HEHHEE", "GHHGGHGHH", "HSHSHHSHSS"), alphabet = 2)
##                 K.2          D.2
## HEHHEE     14.96921 3.117581e-05
## GHHGGHGHH  25.60208 1.963387e-08
## HSHSHHSHSS 26.90906 7.935321e-09

acss(c("HEHHEE", "GHHGGHGHUE", "HSHSHHSHSS"), NULL)
##                 K.2      K.4      K.5      K.6      K.9
## HEHHEE     14.96921 18.55227 19.70361 20.75762 23.38852
## GHHGGHGHUE       NA 31.75832 33.00795 34.27457 37.78935
## HSHSHHSHSS 26.90906 29.37852 30.52566 31.76229 35.15241
##                      D.2          D.4          D.5          D.6
## HEHHEE     3.117581e-05 2.601421e-06 1.171176e-06 5.640722e-07
## GHHGGHGHUE           NA 2.752909e-10 1.157755e-10 4.812021e-11
## HSHSHHSHSS 7.935321e-09 1.432793e-09 6.469341e-10 2.745360e-10
##                      D.9
## HEHHEE     9.106564e-08
## GHHGGHGHUE 4.209915e-12
## HSHSHHSHSS 2.618613e-11

## Not run:
likelihood_d(c("HTHTHTHT", "HTHHTHTT"), alphabet = 2)
##    HTHTHTHT    HTHHTHTT
## 0.010366951 0.003102718

likelihood_ratio(c("HTHTHTHT", "HTHHTHTT"), alphabet = 2)
##  HTHTHTHT  HTHHTHTT
## 0.3767983 1.2589769

prob_random(c("HTHTHTHT", "HTHHTHTT"), alphabet = 2)
##  HTHTHTHT  HTHHTHTT
```

```
## 0.2736772 0.5573217

## End(Not run)

local_complexity(c("01011010111" ,"GHHGGHGHUE"), alphabet = 5, span=5)
## $`01011010111`
##    01011    10110    01101    11010    10101    01011    10111
## 16.22469 16.24766 16.24766 16.22469 16.24322 16.22469 15.93927
##
## $GHHGGHGHUE
##    GHHGG    HHGGH    HGGHG    GGHGH    GHGHU    HGHUE
## 16.44639 16.44639 16.24766 16.22469 16.58986 16.86449


local_complexity(c("01011010111" ,"GHHGGHGHUE"), span=7)
## $`01011010111`
##  0101101  1011010  0110101  1101011  1010111
## 26.52068 26.52068 26.47782 26.62371 26.29186
##
## $GHHGGHGHUE
##  GHHGGHG  HHGGHGH  HGGHGHU  GGHGHUE
## 27.04623 26.86992 27.30871 27.84322
```

---

bdm *Block Decomposition Method*

Description

Obtain Complexity of Longer Strings than allowed by ACSS via the Block Decomposition Method.

Usage

```
bdm(
  string,
  blocksize = 10,
  alphabet = 9,
  delta = blocksize,
  print_blocks = FALSE
)
```

Arguments

`string`	`character` vector containing the to be analyzed strings (can contain multiple strings).
`blocksize`	size of blocks/substrings/windows to be created from `string`. Default is 10. Values larger than 10 will elicit a warning as not all strings with length > 10 are represented for all alphabets.

alphabet	numeric, the number of possible symbols (not necessarily actually appearing in string). Must be one of c(2, 4, 5, 6, 9) (must be a scalar value in the current implementation). Default is 9.
delta	distance between two blocks. Default is delta = blocksize (the maximum) in which the decomposition is disjoint, hence a partition (i.e., there is no overlap between blocks). In the default setting the last block is not of length blocksize if the length of the string is not divisible by blocksize. If delta = 1 the blocks are all of length blocksize and maximally overlap.
print_blocks	logical. Should blocks be printed to the console? Default is FALSE. Mainly for debugging purposes.

Examples

```
acss("SSOOXFXFOXXOXOXFFXXXSOTTOFFFXX")  # too long

bdm("SSOOXFXFOXXOXOXFFXXXSOTTOFFFXX") # default blocksize is 10
bdm("SSOOXFXFOXXOXOXFFXXXSOTTOFFFXX", blocksize = 5)
bdm("SSOOXFXFOXXOXOXFFXXXSOTTOFFFXX", blocksize = 7) # gives warning
bdm("SSOOXFXFOXXOXOXFFXXXSOTTOFFFXX", blocksize = 7, delta = 1)

multi <- c(
  "SSOOXFXFOXXOXOXFFXXXSOTTOFFFXX",
  "SSODXFXDOXXOXOXFFXRRSORTOXDOXX",
  "DXXXXRRXXXSSOOOXOFFFOOOOORFODD"
)

bdm(multi)
bdm(multi, delta = 1)
bdm(multi, blocksize = 5)


# binary bdm should give 57.5664 in this case
bdm("0101010101010101", alphabet = 2, blocksize = 12, delta = 1)

# show all blocks:
bdm("0101010101010101", alphabet = 2, blocksize = 12, delta = 1, print_blocks = TRUE)

# uses a block of size less than 12:
bdm("0101010101010101", alphabet = 2, blocksize = 6)
```

---

entropy *Standard measures of complexity for strings*

Description

Functions to compute different measures of complexity for strings: Entropy, Second-Order Entropy, and Change Complexity

Usage

```
entropy(string)

entropy2(string)

change_complexity(string)
```

Arguments

`string`	`character` vector containing the to be analyzed strings (can contain multiple strings for the entropy measures).

Details

For users who need advanced functions, a comprehensive package computing various versions of entropy estimators is available **entropy**. For users who just need first and second-order entropy and which to apply them to short string, the **acss** package provides two functions: `entropy` (first-order entropy) and `entropy2` second-order entropy.

Change complexity (`change_complexity`) assesses cognitive complexity or the subjective perception of complexity of a binary string. It has been comprehensively defined by Aksentijevic and Gibson (2012). Although the algorithm will work with any number of symbols up to 10, the rationale of Change Complexity only applies to binary strings.

Value

`numeric`, the complexity of the string. For `entropy` and `entropy2` of the same length as `string`. `change_complexity` currently only works with inputs of length 1.

References

Aksentijevic & Gibson (2012). Complexity equals change. *Cognitive Systems Research*, 15-17, 1-16.

Examples

```
strings1 <- c("010011010001", "0010203928837", "0000000000")
strings2 <- c("001011", "01213", "010101010101")

entropy(strings1)
entropy("XYXXYYXYXXXY") # "same" string as strings1[1]
entropy(c("HUHHEGGTE", "EGGHHU"))

entropy2(strings1)
entropy2("XYXXYYXYXXXY")

entropy2(strings2)

change_complexity(strings1)
change_complexity("XYXXYYXYXXXY")
```

exp1 *Data from Experiment 1 in Gauvrit, Singmann, Soler-Toscano & Zenil*

Description

34 participants were asked to produce at their own pace a series of 10 symbols among "A", "B", "C", and "D" that would "look as random as possible, so that if someone else sees the sequence, she will believe it is a truly random one".

Usage

```
exp1
```

Format

A data.frame with 34 rows and 2 variables.

Source

Gauvrit, Singmann, Soler-Toscano & Zenil (submitted). Complexity for psychology. A user-friendly implementation of the coding theorem method.

Examples

```
# load data
data(exp1)

# summary statistics
nrow(exp1)
summary(exp1$age)
mean(exp1$age)
sd(exp1$age)

## Not run:
# this uses code from likelihood_d() to calculate the mean complexity K
# for all strings of length 10 with alphabet = 4:
tmp <- acss_data[nchar(rownames(acss_data)) == 10, "K.4", drop = FALSE]
tmp <- tmp[!is.na(tmp[,"K.4"]),,drop = FALSE]
tmp$count <- count_class(rownames(tmp), alphabet = 4)
(mean_K <- with(tmp, sum(K.4*count)/sum(count)))

t.test(acss(exp1$string, 4)[,"K.4"], mu = mean_K)

## End(Not run)
```

exp2 *Data from Experiment 2 in Gauvrit, Singmann, Soler-Toscano & Zenil*

Description

Responses of one participant (42 years old) to 200 randomly generated strings of length 10 from an alphabet of 6 symbols. For each string, the participant was asked to indicate whther or not the string appears random or not.

Usage

```
exp2
```

Format

A data.frame with 200 rows and 2 variables.

Source

Gauvrit, Singmann, Soler-Toscano & Zenil (submitted). Complexity for psychology. A user-friendly implementation of the coding theorem method.

Examples

```
# load data
data(exp2)

exp2$K <- acss(exp2$string, 6)[,"K.6"]

m_log <- glm(response ~ K, exp2, family = binomial)
summary(m_log)

# odds ratio of K:
exp(coef(m_log)[2])

# calculate threshold of 0.5
(threshold <- -coef(m_log)[1]/coef(m_log)[2])

require(effects)
require(lattice)
plot(Effect("K", m_log), rescale.axis = FALSE, ylim = c(0, 1))
trellis.focus("panel", 1, 1)
panel.lines(rep(threshold, 2), c(0, 0.5), col = "black", lwd = 2.5, lty = 3)
panel.lines(c(33,threshold), c(0.5, 0.5), col = "black", lwd = 2.5, lty = 3)
trellis.unfocus()
```

matthews2013 *Data from Experiment 1 in Matthews (2013)*

Description

Mean responses on a 6-point scale ("definitely random" to "definitely not random") of participants to 216 strings of length 21.

Usage

```
matthews2013
```

Format

A data.frame with 216 rows and 3 variables.

Source

Matthews, W. (2013). Relatively random: Context effects on perceived randomness and predicted outcomes. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 39(5), 1642-1648.

Examples

```
## Not run:
data(matthews2013)

spans <- 3:11
# note, the next loop takes more than 5 minutes.
for (i in spans) {
  matthews2013[,paste0("K2_span", i)] <-
    sapply(local_complexity(matthews2013$string, alphabet=2, span = i), mean)
}

lm_list <- vector("list", 8)
for (i in seq_along(spans)) {
  lm_list[[i]] <- lm(as.formula(paste0("mean ~ K2_span", spans[i])), matthews2013)
}

plot(spans, sapply(lm_list, function(x) summary(x)$r.squared), type = "o")

# do more predictors increase fit?
require(MASS)
m_initial <- lm(mean ~ 1, matthews2013)
m_step <- stepAIC(m_initial,
                  scope = as.formula(paste("~", paste(paste0("K2_span", spans),
                  collapse = "+"))))
summary(m_step)
```

```
m_initial2 <- lm(as.formula(paste("mean ~", paste(paste0("K2_span", spans),
                  collapse = "+"))), matthews2013)
m_step2 <- stepAIC(m_initial2)
summary(m_step2)


## End(Not run)
```

---

normalize_string *Helper functions for calculating cognitive complexity.*

Description

normalize_string takes a character vector and normalizes its input using the symbols 0, 1, 2...9. count_class takes a character vector and an integer alphabet (with the restriction that the number of different symbols in the character vector doesn't exceed alphabet) and returns the total number of strings that are equivalent to the input when normalized and considering alphabet. alternations returns the number of alternations of symbols in a string.

Usage

```
normalize_string(string)

count_class(string, alphabet)

alternations(string, proportion = FALSE)
```

Arguments

string	character vector containing the to be analyzed strings (can contain multiple strings).
alphabet	numeric, the number of possible symbols (not necessarily actually appearing in string).
proportion	boolean, indicating if the result from alternation should be given as a proportion (between 0 and 1) or the raw number of alternations (default is FALSE correpsonding to raw values).

Details

nothing yet.

Value

normalize_string A normalized vector of strings of the same length as string.

count_class A vector of the same length as string with the number of possible equivalent strings when string is normalized and considering alphabet.

alternations A vector with the number (or proprtion) of alternations of the same length as string

Examples

```
#normalize_string:
normalize_string(c("HUHHEGGTE", "EGGHHU"))

normalize_string("293948837163536")

# count_class
count_class("010011",2)

count_class("332120",4)

count_class(c("HUHHEGGTE", "EGGHHU"), 5)
count_class(c("HUHHEGGTE", "EGGHHU"), 6)

# alternations:
alternations("0010233")
alternations("0010233", proportion = TRUE)

alternations(c("HUHHEGGTE", "EGGHHU"))
alternations(c("HUHHEGGTE", "EGGHHU"), proportion = TRUE)
```

Index