Package: eddington (via r-universe)

September 16, 2024

Title Compute a Cyclist's Eddington Number

Version 4.2.0

Description Compute a cyclist's Eddington number, including efficiently computing cumulative E over a vector. A cyclist's Eddington number <https://en.wikipedia.org/wiki/Arthur_Eddington#Eddington_number_for_cycling> is the maximum number satisfying the condition such that a cyclist has ridden E miles or greater on E distinct days. The algorithm in this package is an improvement over the conventional approach because both summary statistics and cumulative statistics can be computed in linear time, since it does not require initial sorting of the data. These functions may also be used for computing h-indices for authors, a metric described by Hirsch (2005) <doi:10.1073/pnas.0507655102>. Both are specific applications of computing the side length of a Durfee square <https://en.wikipedia.org/wiki/Durfee_square>.

License GPL (>= 2)

Encoding UTF-8

LazyData true

Depends R (>= 4.2.0)

LinkingTo Rcpp

Imports Rcpp, R6, methods, xml2

Suggests testthat, knitr, rmarkdown, stats, dplyr

SystemRequirements C++17

VignetteBuilder knitr

RoxygenNote 7.2.3

URL https://github.com/pegeler/eddington2

BugReports https://github.com/pegeler/eddington2/issues

NeedsCompilation yes

Author Paul Egeler [aut, cre], Tashi Reigle [ctb]

Maintainer Paul Egeler <paulegeler@gmail.com>

Date/Publication 2024-08-16 20:10:02 UTC

Repository https://pegeler.r-universe.dev

RemoteUrl https://github.com/cran/eddington

RemoteRef HEAD

RemoteSha 4439063bee807bd9ded8f3385ed9e6e97b8fd096

Contents

Eddington *An R6 Class for Tracking Eddington Numbers for Cycling*

Description

The class will maintain the state of the algorithm, allowing for efficient updates as new rides come in.

Warnings

The implementation uses an experimental base R feature utils::hashtab.

Cloning of Eddington objects is disabled. Additionally, Eddington objects cannot be serialized; they cannot be carried between sessions using base::saveRDS or base::save and then loaded later using base::readRDS or base::load.

Active bindings

current The current Eddington number.

cumulative A vector of cumulative Eddington numbers.

number_to_next The number of rides needed to get to the next Eddington number.

n The number of rides in the data.

hashmap The hash map of rides above the current Eddington number.

Methods

Public methods:

- `Eddington$new()`
- `Eddington$print()`
- `Eddington$update()`
- `Eddington$getNumberToTarget()`
- `Eddington$isSatisfied()`

Method `new()`**:** Create a new Eddington object.

Usage:

```
Eddington$new(rides, store.cumulative = FALSE)
```

Arguments:

`rides` A vector of rides

`store.cumulative` logical, indicating whether to keep a vector of cumulative Eddington numbers

Returns: A new `Eddington` object

Method `print()`**:** Print the current Eddington number.

Usage:

```
Eddington$print()
```

Method `update()`**:** Add new rides to the existing `Eddington` object.

Usage:

```
Eddington$update(rides)
```

Arguments:

`rides` A vector of rides

Method `getNumberToTarget()`**:** Get the number of rides of a specified length to get to a target Eddington number.

Usage:

```
Eddington$getNumberToTarget(target)
```

Arguments:

`target` Target Eddington number

Returns: An integer representing the number of rides of target length needed to achieve the target number.

Method `isSatisfied()`**:** Test if an Eddington number is satisfied.

Usage:

```
Eddington$isSatisfied(target)
```

Arguments:

`target` Target Eddington number

Returns: Logical

Examples

```
# Randomly generate a set of 15 rides
rides <- rgamma(15, shape = 2, scale = 10)

# View the rides sorted in decreasing order
stats::setNames(sort(rides, decreasing = TRUE), seq_along(rides))

# Create the Eddington object
e <- Eddington$new(rides, store.cumulative = TRUE)

# Get the Eddington number
e$current

# Update with new data
e$update(rep(25, 10))

# See the new data
e$cumulative
```

`EddingtonModule` *An Rcpp Module for Tracking Eddington Numbers for Cycling*

Description

A stateful C++ object for computing Eddington numbers.

Arguments

`rides` An optional vector of values used to initialize the class.

`store_cumulative` Whether to store a vector of the cumulative Eddington number, as accessed from the `cumulative` property.

Fields

`new` Constructor. Parameter list may either be empty, `store_cumulative`, or `rides` and `store_cumulative`

`current` The current Eddington number.

`cumulative` A vector of Eddington numbers or `NULL` if `store_cumulative` is `FALSE`.

`hashmap` A `data.frame` containing the distances and counts above the current Eddington number.

`update` Update the class state with new data.

`getNumberToNext` Get the number of additional distances required to reach the next Eddington number.

`getNumberToTarget` Get the number of additional distances required to reach a target Eddington number.

Warning

EddingtonModule objects cannot be serialized at this time; they cannot be carried between sessions using base::saveRDS or base::save and then loaded later using base::readRDS or base::load.

Examples

```
# Create a class instance with some initial data
e <- EddingtonModule$new(c(3, 3, 2), store_cumulative = TRUE)
e$current

# Update with new data and look at the vector of cumulative Eddington numbers.
e$update(c(3, 3, 5))
e$cumulative

# Get the number of rides required to reach the next Eddington number and
# an Eddington number of 4.
e$getNumberToNext()
e$getNumberToTarget(4)
```

E_cum — *Calculate the cumulative Eddington number*

Description

This function is much like E_num except it provides a cumulative Eddington number over the vector rather than a single summary number.

Usage

```
E_cum(rides)
```

Arguments

rides	A vector of mileage, where each element represents a single day.

Value

An integer vector the same length as rides.

See Also

E_next, E_num, E_req, E_sat

`E_next` *Get the number of rides required to increment to the next Eddington number*

Description

Get the number of rides required to increment to the next Eddington number.

Usage

```
E_next(rides)
```

Arguments

`rides`	A vector of mileage, where each element represents a single day.

Value

A named list with the current Eddington number (`E`) and the number of rides required to increment by one (`req`).

See Also

`E_cum`, `E_num`, `E_req`, `E_sat`

`E_num` *Get the Eddington number for cycling*

Description

Gets the Eddington number for cycling. The Eddington Number for cycling, *E*, is the maximum number where a cyclist has ridden *E* miles on *E* distinct days.

Usage

```
E_num(rides)
```

Arguments

`rides`	A vector of mileage, where each element represents a single day.

Details

The Eddington Number for cycling is related to computing the rank of an integer partition, which is the same as computing the side length of its Durfee square. Another relevant application of this metric is computing the Hirsch index (doi:10.1073/pnas.0507655102) for publications.

This is not to be confused with the Eddington Number in astrophysics, N_{Edd}, which represents the number of protons in the observable universe.

Value

An integer which is the Eddington cycling number for the data provided.

See Also

E_cum, E_next, E_req, E_sat

Examples

```
# Randomly generate a set of 15 rides
rides <- rgamma(15, shape = 2, scale = 10)

# View the rides sorted in decreasing order
stats::setNames(sort(rides, decreasing = TRUE), seq_along(rides))

# Get the Eddington number
E_num(rides)
```

E_req *Determine the number of additional rides required to achieve a specified Eddington number*

Description

Determine the number of additional rides required to achieve a specified Eddington number.

Usage

```
E_req(rides, candidate)
```

Arguments

rides	A vector of mileage, where each element represents a single day.
candidate	The Eddington number to test for.

Value

An integer vector of length 1. Returns 0L if *E* is already achieved.

See Also

E_cum, E_next, E_num, E_sat

E_sat *Determine if a dataset satisfies a specified Eddington number*

Description

Indicates whether a certain Eddington number is satisfied, given the data.

Usage

```
E_sat(rides, candidate)
```

Arguments

`rides`	A vector of mileage, where each element represents a single day.
`candidate`	The Eddington number to test for.

Value

A logical vector of length 1.

See Also

`E_cum`, `E_next`, `E_num`, `E_req`

get_haversine_distance *Compute the distance between two points using the Haversine formula*

Description

Uses the Haversine great-circle distance formula to compute the distance between two latitude/longitude points.

Usage

```
get_haversine_distance(
  lat_1,
  lon_1,
  lat_2,
  lon_2,
  units = c("miles", "kilometers")
)
```

Arguments

`lat_1`, `lon_1`, `lat_2`, `lon_2`
: The coordinates used to compute the distance.

`units`
: The units of the output distance.

Value

The distance between two points in the requested units.

References

https://en.wikipedia.org/wiki/Haversine_formula

Examples

```
# In NYC, 20 blocks == 1 mile. Thus, computing the distance between two
# points along 7th Ave from W 39 St to W 59 St should return ~1 mile.
w39_coords <- list(lat=40.75406905512651, lon=-73.98830604245481)
w59_coords <- list(lat=40.76684156255418, lon=-73.97908243833855)

get_haversine_distance(
  w39_coords$lat,
  w39_coords$lon,
  w59_coords$lat,
  w59_coords$lon,
  "miles"
)

# The total distance along a sequence of points can be computed. Consider the
# following sequence of points along Park Ave in the form of a list of points
# where each point is a list containing a `lat` and `lon` tag.
park_ave_coords <- list(
  list(lat=40.735337983655434, lon=-73.98973648773142),  # E 15 St
  list(lat=40.74772623378332, lon=-73.98066078090876),   # E 35 St
  list(lat=40.76026319186414, lon=-73.97149360922498),   # E 55 St
  list(lat=40.77301604875587, lon=-73.96217737679450)    # E 75 St
)

# We can create a function to compute the total distance as follows:
compute_total_distance <- function(coords) {
  sum(
    sapply(
      seq_along(coords)[-1],
      \(i) get_haversine_distance(
        coords[[i]]$lat,
        coords[[i]]$lon,
        coords[[i - 1]]$lat,
        coords[[i - 1]]$lon,
        "miles"
      )
    )
  )
}

# Then applying the function to our sequence results in a total distance.
compute_total_distance(park_ave_coords)
```

read_gpx *Read a GPX file into a data frame containing dates and distances*

Description

Reads in a GPS Exchange Format XML document and outputs a `data.frame` containing distances. The corresponding dates for each track segment (`trkseg`) will be included if present in the source file, else the `date` column will be populated with `NA`s.

Usage

```
read_gpx(file, units = c("miles", "kilometers"))
```

Arguments

`file`	The input file to be parsed.
`units`	The units desired for the distance metric.

Details

Distances are computed using the Haversine formula and do not account for elevation changes.

This function treats the first timestamp of each `trkseg` as the date of record. Thus overnight track segments will all count toward the day in which the journey began.

Value

A data frame containing up to two columns:

date The date of the ride. See description and details.

distance The distance of the track segment in the requested units.

Examples

```
## Not run:
# Get a list of all GPX export files in a directory tree
gpx_export_files <- list.files(
  "/path/to/gpx/exports/",
  pattern = "\\.gpx$",
  full.names = TRUE,
  recursive = TRUE
)
```

```
# Read in all files and combine them into a single data frame
rides <- do.call(rbind, lapply(gpx_export_files, read_gpx))

## End(Not run)
```

`rides` *A year of simulated bicycle ride mileages*

Description

Simulated dates and distances of rides occurring in 2009.

Usage

```
rides
```

Format

A data frame with 250 rows and 2 variables:

ride_date date the ride occurred

ride_length the length in miles

Details

The dataset contains a total of 3,419 miles spread across 178 unique days. The Eddington number for the year was 29.

Index