Package ‘mpathsenser’

October 13, 2022

Title Process and Analyse Data from m-Path Sense

Version 1.1.0

Description Overcomes one of the major challenges in mobile (passive) sensing, namely being able to pre-process the raw data that comes from a mobile sensing app, specifically "m-Path Sense" <https://m-path.io>. The main task of `mpathsenser` is therefore to read "m-Path Sense" JSON files into a database and provide several convenience functions to aid in data processing.

Depends R (>= 4.0.0)

License GPL (>= 3)


BugReports https://gitlab.kuleuven.be/ppw-okpiv/researchers/u0134047/mpathsenser/-/issues/

Encoding UTF-8

RoxygenNote 7.2.1

Imports DBI, dbplyr, dplyr, furrr, future, jsonlite, lubridate, magrittr, purrr, rjson, rlang, RSQLite, stats, tibble, tidyr

Suggests curl, dbx, ggplot2, httr, knitr, lifecycle, lme4, lmerTest, progressr, rmarkdown, rvest, sodium, testthat (>= 3.0.0), vroom

Config/testthat/edition 3

Config/Needs/website tidyverse/tidytemplate

VignetteBuilder knitr

NeedsCompilation no

Author Koen Niemeijer [aut, cre] (<https://orcid.org/0000-0002-0816-534X>), Kristof Meers [ctb] (<https://orcid.org/0000-0002-9610-7712>), KU Leuven [cph, fnd]

Maintainer Koen Niemeijer <koen.niemeijer@kuleuven.be>

Repository CRAN

Date/Publication 2022-10-06 12:30:02 UTC
**R topics documented:**

- activity_duration .......................... 3
- add_gaps .................................... 4
- app_category .................................. 5
- bin_data ...................................... 6
- copy .......................................... 8
- close_db ..................................... 9
- copy_db ..................................... 10
- coverage .................................... 10
- create_db ................................... 12
- decrypt_gps ................................ 13
- device_info ................................... 13
- first_date .................................. 14
- fix_jsons ................................... 14
- freq .......................................... 15
- geocode_rev .................................. 16
- get_app_usage ................................ 17
- get_data ..................................... 18
- get_installed_apps ......................... 19
- get_nrows ................................... 19
- get_participants ............................. 20
- get_processed_files ......................... 21
- get_studies .................................. 21
- haversine .................................... 22
- identify_gaps ................................ 22
- import ...................................... 24
- index_db ..................................... 25
- last_date ................................... 26
- link .......................................... 26
- link_db ...................................... 28
- link_gaps ................................... 30
- moving_average ............................. 31
- n_screen_on ................................ 32
- n_screen_unlocks ........................... 33
- open_db ..................................... 34
- screen_duration ............................ 34
- sensors ...................................... 35
- step_count .................................. 36
- test_jsons ................................... 36
- unzip_data .................................. 37
activity_duration  

Get a summary of physical activity (recognition)

Description

[Experimental]

Usage

activity_duration(
  data = NULL,
  db = NULL,
  participant_id = NULL,
  confidence = 70,
  direction = "forward",
  start_date = NULL,
  end_date = NULL,
  by = c("Total", "Day", "Hour")
)

Arguments

data  A data frame containing the activity data. See get_data for retrieving activity data from an mpathsenser database.

db  A database connection to an m-Path Sense database.

participant_id  A character string identifying a single participant. Use get_participants to retrieve all participants from the database. Leave empty to get data for all participants.

confidence  The minimum confidence (0-100) that should be assigned to an observation by Activity Recognition.

direction  The directionality of the duration calculation, i.e. $t - t_{t-1}$ or $t_{t+1} - t$.

start_date  Optional search window specifying date where to begin search. Must be convertible to date using as.Date. Use first_date to find the date of the first entry for a participant.

end_date  Optional search window specifying date where to end search. Must be convertible to date using as.Date. Use last_date to find the date of the last entry for a participant.

by  Either 'Total', 'Hour', or 'Day' indicating how to summarise the results.

Value

A tibble containing a column 'activity' and a column 'duration' for the hourly activity duration.
add_gaps

Add gap periods to sensor data

Description

[Experimental]

Since there may be many gaps in mobile sensing data, it is pivotal to pay attention in the analysis to them. This function adds known gaps to data as "measurements", thereby allowing easier calculations for, for example, finding the duration. For instance, consider a participant spent 30 minutes walking. However, if it is known there is a gap of 15 minutes in this interval, we should somehow account for it. add_gaps accounts for this by adding the gap data to sensors data by splitting intervals where gaps occur.

Usage

add_gaps(data, gaps, by = NULL, fill = NULL)

Arguments

data A data frame containing the activity data. See get_data for retrieving activity data from an mpathsenser database.
gaps A data frame (extension) containing the gap data. See identify_gaps for retrieving gap data from an mpathsenser database. It should at least contain the columns from and to (both in a date-time format), as well as any specified columns in by.
by A character vector indicating the variable(s) to match by, typically the participant IDs. If NULL, the default, *_join() will perform a natural join, using all variables in common across x and y.
fill A named list of the columns to fill with default values for the extra measurements that are added because of the gaps.

Details

In the example of 30 minutes walking where a 15 minute gap occurred (say after 5 minutes), add_gaps adds two rows: one after 5 minutes of the start of the interval indicating the start of the gap(if needed containing values from fill), and one after 20 minutes of the start of the interval signalling the walking activity. Then, when calculating time differences between subsequent measurements, the gap period is appropriately accounted for. Note that if multiple measurements occurred before the gap, they will both be continued after the gap.

Value

A tibble containing the data and the added gaps.
See Also

identify_gaps for finding gaps in the sampling; link_gaps for finding which gaps occur in the data;

Examples

# Define some data
dat <- data.frame(
  participant_id = "12345",
  time = as.POSIXct(c("2022-05-10 10:00:00", "2022-05-10 10:30:00", "2022-05-10 11:30:00")),
  type = c("WALKING", "STILL", "RUNNING"),
  confidence = c(80, 100, 20)
)

# Get the gaps from identify_gaps, but in this example define them ourselves
gaps <- data.frame(
  participant_id = "12345",
  from = as.POSIXct(c("2022-05-10 10:05:00", "2022-05-10 10:50:00")),
  to = as.POSIXct(c("2022-05-10 10:20:00", "2022-05-10 10:10:00"))
)

# Now add the gaps to the data
add_gaps(data = dat,
gaps = gaps,
by = "participant_id")

# You can use fill if you want to get rid of those pesky NA's
add_gaps(data = dat,
gaps = gaps,
by = "participant_id",
fill = list(type = "GAP", confidence = 100))

---

**app_category**

*Find the category of an app on the Google Play Store*

**Description**

[Stable]

This function scrapes the Google Play Store by using name as the search term. From there it selects the first result in the list and its corresponding category and package name.

**Usage**

app_category(name, num = 1, rate_limit = 5, exact = TRUE)
Arguments

name: The name of the app to search for.
num: Which result should be selected in the list of search results. Defaults to one.
rate_limit: The time interval to keep between queries, in seconds. If the rate limit is too low, the Google Play Store may reject further requests or even ban your entirely.

exact: In m-Path Sense, the app names of the AppUsage sensor are the last part of the app’s package names. When exact is TRUE, the function guarantees that name is exactly equal to the last part of the selected package from the search results. Note that when exact is TRUE, it interacts with num in the sense that it no longer selects the top search result but instead the top search result that matches the last part of the package name.

Value

A list containing the following fields:

- package: the package name that was selected from the Google Play search
- genre: the corresponding genre of this package

Warning

Do not abuse this function or you will be banned by the Google Play Store. The minimum delay between requests seems to be around 5 seconds, but this is untested. Also make sure not to do batch lookups, as many subsequent requests will get you blocked as well.

Examples

```r
app_category("whatsapp")

# Example of a generic app name where we can't find a specific app
app_category("weather") # Weather forecast channel

# Get OnePlus weather
app_category("net.oneplus.weather")
```

bin_data

Create bins in variable time series

Description

[Experimental]

In time series with variable measurements, an often recurring task is calculating the total time spent (i.e. the duration) in fixed bins, for example per hour or day. However, this may be difficult when two subsequent measurements are in different bins or span over multiple bins.
**Usage**

```r
bin_data(
  data, 
  start_time, 
  end_time, 
  by = c("sec", "min", "hour", "day"), 
  fixed = TRUE
)
```

**Arguments**

- **data** A data frame or tibble containing the time series.
- **start_time** The column name of the start time of the interval, a POSIXct.
- **end_time** The column name of the end time of the interval, a POSIXct.
- **by** A binning specification.
- **fixed** Whether to create fixed bins. If TRUE, bins will be rounded to, for example, whole hours or days (depending on by). If FALSE, bins will be created based on the first timestamp.

**Value**

A tibble containing the group columns (if any), date, hour (if by = "hour"), and the duration in seconds.

**See Also**

- `link_gaps` for linking gaps to data.

**Examples**

```r
data <- tibble::tibble(
  participant_id = 1, 
  datetime = c("2022-06-21 15:00:00", "2022-06-21 15:55:00", 
  "2022-06-21 17:05:00", "2022-06-21 17:10:00"), 
  confidence = 100, 
  type = "WALKING"
)

# get bins per hour, even if the interval is longer than one hour
data %>%
dplyr::mutate(datetime = as.POSIXct(datetime)) %>%
dplyr::mutate(lead = dplyr::lead(datetime)) %>%
bin_data(
  start_time = datetime, 
  end_time = lead, 
  by = "hour"
)
```
# Alternatively, you can give an integer value to by to create custom-sized bins, but only if fixed = FALSE. Not that these bins are not rounded to, as in this example 30 minutes, but rather depends on the earliest time in the group.
data %>%
dplyr::mutate(datetime = as.POSIXct(datetime)) %>%
dplyr::mutate(lead = dplyr::lead(datetime)) %>%
bin_data(
  start_time = datetime,
  end_time = lead,
  by = 1800L,
  fixed = FALSE
)

# More complicated data for showcasing grouping:
data <- tibble::tibble(
  participant_id = 1,
  datetime = c(
    "2022-06-21 15:00:00", "2022-06-21 15:55:00",
    "2022-06-21 17:05:00", "2022-06-21 17:10:00"
  ),
  confidence = 100,
  type = c("STILL", "WALKING", "STILL", "WALKING")
)

# binned_intervals also takes into account the prior grouping structure
out <- data %>%
dplyr::mutate(datetime = as.POSIXct(datetime)) %>%
dplyr::mutate(lead = dplyr::lead(datetime)) %>%
dplyr::group_by(participant_id, type) %>%
bin_data(
  start_time = datetime,
  end_time = lead,
  by = "hour"
)
print(out)

# To get the duration for each bin (note to change the variable names in sum):
purrr::map_dbl(out$bin_data, ~ sum(.x$lead - .x$datetime, na.rm = TRUE))

# Or:
out %>%
tidyr::unnest(bin_data) %>%
dplyr::mutate(duration = .data$lead - .data$datetime) %>%
dplyr::group_by(date, .add = TRUE) %>%
dplyr::summarise(duration = sum(.data$duration), .groups = "drop")
close_db

Description

[Stable]
Copy zip files from a source destination to an origin destination where they do not yet exist. That is, it only updates the target folder from the source folder.

Usage

ccopy(from, to, recursive = TRUE)

Arguments

from A path to copy files from.
to A path to copy files to.
recursive Should files from subdirectories be copied?

Value

A message indicating how many files were copied.

Examples

## Not run:
ccopy("K:/data/myproject/", "~/myproject")
## End(Not run)

---

close_db

Close a database connection

documentation

Description

[Stable]
This is a convenience function that is simply a wrapper around dbDisconnect.

Usage

close_db(db)

Arguments

db A database connection to an m-Path Sense database.

Value

close_db returns invisibly regardless of whether the database is active, valid, or even exists.

See Also

open_db for opening an mpathsenser database.
copy_db

Copy (a subset of) a database to another database

Description

[Stable]

Usage

```r
copy_db(
  from_db,
  to_db = NULL,
  sensor = "All",
  path = getwd(),
  db_name = "sense.db"
)
```

Arguments

- `from_db`: A mpathsenser database connection from where the data will be transferred.
- `to_db`: A mpathsenser database connection where the data will be transferred to. If no new_db is specified, a path (and possibly a db_name) must be specified for `create_db` to create a new database.
- `sensor`: A character vector containing one or multiple sensors. See `sensors` for a list of available sensors. Use "All" for all available sensors.
- `path`: The path to the database. Use NULL to use the full path name in db_name.
- `db_name`: The name of the database.

Value

No return value, called for side effects.

coverage

Create a coverage chart of the sampling rate

Description

[Stable]

Only applicable to non-reactive sensors with 'continuous' sampling
coverage

Usage

coverage(
  db,
  participant_id,
  sensor = "All",
  frequency = mpathsenser::freq,
  relative = TRUE,
  offset = "None",
  start_date = NULL,
  end_date = NULL,
  plot = TRUE
)

Arguments

db 
  A valid database connection. Schema must be that as it is created by open_db.

participant_id 
  A character string of one participant ID.

sensor 
  A character vector containing one or multiple sensors. See sensors for a list of
  available sensors. Use ‘All’ for all available sensors.

frequency 
  A named numeric vector with sensors as names and the number of expected
  samples per hour

relative 
  Show absolute number of measurements or relative to the expected number?
  Logical value.

offset 
  Currently not used.

start_date 
  A date (or convertible to a date using as.Date) indicating the earliest date to
  show. Leave empty for all data. Must be used with end_date.

end_date 
  A date (or convertible to a date using as.Date) indicating the latest date to
  show. Leave empty for all data. Must be used with start_date.

plot 
  Whether to return a ggplot or its underlying data.

Value

A ggplot of the coverage results if plot is TRUE or a tibble containg the hour, type of measure (i.e.
  sensor), and (relative) coverage.

Examples

## Not run:
fix_json()
unzip()
freq <- c(
  Accelerometer = 720, # Once per 5 seconds. Can have multiple measurements.
  AirQuality = 1,
  AppUsage = 2, # Once every 30 minutes
  Bluetooth = 60, # Once per minute. Can have multiple measurements.
  Gyroscope = 720, # Once per 5 seconds. Can have multiple measurements.
  Light = 360, # Once per 10 seconds
)
create_db

Create a new mpathsenser database

Description

[Stable]

Usage

create_db(path = getwd(), db_name = "sense.db", overwrite = FALSE)

Arguments

path The path to the database.
db_name The name of the database.
overwrite In case a database with db_name already exists, indicate whether it should be overwritten or not. Otherwise, this option is ignored.

Value

A database connection using prepared database schemas.
**decrypt_gps**

Decrypt GPS data from a curve25519 public key

**Description**

[Stable]

By default, the latitude and longitude of the GPS data collected by m-Path Sense will be encrypted using an asymmetric curve25519 key to provide extra protection for these highly sensitive data. This function takes the entire location data set and decrypts its longitude and latitude columns using the provided key.

**Usage**

decrypt_gps(data, key)

**Arguments**

- **data**: A (lazy) tibble containing the GPS data
- **key**: A curve25519 public key

**Value**

A tibble containing the non-lazy, decrypted GPS data

**device_info**

Get the device info for one or more participants

**Description**

[Stable]

**Usage**

device_info(db, participant_id = NULL)

**Arguments**

- **db**: A database connection to an m-Path Sense database.
- **participant_id**: A character string identifying a single participant. Use `get_participants` to retrieve all participants from the database. Leave empty to get data for all participants.

**Value**

A tibble containing device info for each participant
first_date  

Extract the date of the first entry

Description

[Stable]
A helper function for extracting the first date of entry of (of one or all participant) of one sensor. Note that this function is specific to the first date of a sensor. After all, it wouldn’t make sense to extract the first date for a participant of the accelerometer, while the first device measurement occurred a day later.

Usage

first_date(db, sensor, participant_id = NULL)

Arguments

db  A database connection to an m-Path Sense database.
sensor  The name of a sensor. See sensors for a list of available sensors.
participant_id  A character string identifying a single participant. Use get_participants to retrieve all participants from the database. Leave empty to get data for all participants.

Value

A string in the format ’YYYY-mm-dd’ of the first entry date.

Examples

## Not run:
db <- open_db()
first_date(db, “Accelerometer”, “12345”)

## End(Not run)

fix_jsons  

Fix the end of JSON files

Description

[Stable]
When copying data directly coming from m-Path Sense, JSON files are sometimes corrupted due to the app not properly closing them. This function attempts to fix the most common problems associated with improper file closure by m-Path Sense.
**freq**

**Usage**

```r
fix_jsons(path = getwd(), files = NULL, recursive = TRUE, parallel = FALSE)
```

**Arguments**

- `path`: The path name of the JSON files.
- `files`: Alternatively, a character list of the input files
- `recursive`: Should the listing recurse into directories?
- `parallel`: A logical value whether you want to check in parallel. Useful for a lot of files.

**Value**

A message indicating how many files were fixed.

**Progress**

You can be updated of the progress by this function by using the `progress` package. See `progressr`'s vignette on how to subscribe to these updates.

**Examples**

```r
## Not run:
future::plan(future::multisession)
files <- test_jsons()
fix_jsons(files = files)
## End(Not run)
```

---

**freq**

*Measurement frequencies per sensor*

**Description**

A numeric vector containing (an example) of example measurement frequencies per sensor. Such input is needed for `coverage`.

**Usage**

```r
freq
```

**Format**

An object of class `numeric` of length 11.

**Value**

This vector contains the following information:
### geocode_rev

**Reverse geocoding with latitude and longitude**

**Description**

*Experimental*

This function allows you to extract information about a place based on the latitude and longitude from the OpenStreetMaps nominatim API.

**Usage**

```
geocode_rev(lat, lon, zoom = 18, email = "", rate_limit = 1)
```

**Arguments**

- `lat`: The latitude of the location (in degrees)
- `lon`: The longitude of the location (in degrees)
- `zoom`: The desired zoom level from 1-18. The lowest level, 18, is building level.
- `email`: If you are making large numbers of request please include an appropriate email address to identify your requests. See Nominatim's Usage Policy for more details.
- `rate_limit`: The time interval to keep between queries, in seconds. If the rate limit is too low, the OpenStreetMaps may reject further requests or even ban your entirely.

**Value**

A list of information about the location. See Nominatim's documentation for more details.

**Warning**

Do not abuse this function or you will be banned by OpenStreetMap. The maximum number of requests is around 1 per second. Also make sure not to do batch lookups, as many subsequent requests will get you blocked as well.
get_app_usage

Examples

# Frankfurt Airport
geocode_rev(50.037936, 8.5599631)

df <- get_app_usage(
  db, participant_id = NULL, start_date = NULL, end_date = NULL,
  by = c("Total", "Day", "Hour")
)

Description

[Experimental]

This function extracts app usage per hour for either one or multiple participants. If multiple days are selected, the app usage time is averaged.

Usage

gt_app_usage(
  db, participant_id = NULL, start_date = NULL, end_date = NULL,
  by = c("Total", "Day", "Hour")
)

Arguments

db A database connection to an m-Path Sense database.
participant_id A character string identifying a single participant. Use get_participants to retrieve all participants from the database. Leave empty to get data for all participants.
start_date Optional search window specifying date where to begin search. Must be convertible to date using as.Date. Use first_date to find the date of the first entry for a participant.
end_date Optional search window specifying date where to end search. Must be convertible to date using as.Date. Use last_date to find the date of the last entry for a participant.
by Either 'Total', 'Hour', or 'Day' indicating how to summarise the results.

Value

A data frame containing a column 'app' and a column 'usage' for the hourly app usage.
get_data

Generic helper function from extracting data from an m-Path Sense database

Description

[Stable]

This is a generic function to help extract data from an m-Path sense database. For some sensors that require a bit more pre-processing, such as app usage and screen time, more specialised functions are available (e.g. get_app_usage and screen_duration).

Usage

get_data(db, sensor, participant_id = NULL, start_date = NULL, end_date = NULL)

Arguments

db A database connection to an m-Path Sense database.
sensor The name of a sensor. See sensors for a list of available sensors.
participant_id A character string identifying a single participant. Use get_participants to retrieve all participants from the database. Leave empty to get data for all participants.
start_date Optional search window specifying date where to begin search. Must be convertible to date using as.Date. Use first_date to find the date of the first entry for a participant.
end_date Optional search window specifying date where to end search. Must be convertible to date using as.Date. Use last_date to find the date of the last entry for a participant.

Value

A lazy tbl containing the requested data.

Examples

```r
## Not run:
# Open a database
db <- open_db()

# Retrieve some data
get_data(db, "Accelerometer", "12345")

# Or within a specific window
get_data(db, "Accelerometer", "12345", "2021-01-01", "2021-01-05")

## End(Not run)
```
get_installed_apps  Get installed apps

Description

[Stable]

Extract installed apps for one or all participants. Contrarily to other get_* functions in this package, start and end dates are not used since installed apps are assumed to be fixed throughout the study.

Usage

get_installed_apps(db, participant_id = NULL)

Arguments

db A database connection to a mpathsenser database.

participant_id A character string identifying a single participant. Use get_participants to retrieve all participants from the database. Leave empty to get data for all participants.

Value

A tibble containing app names.

get_nrows  Get the number of rows per sensor in a mpathsenser database

Description

[Stable]

Usage

get_nrows(
  db,
  sensor = "All",
  participant_id = NULL,
  start_date = NULL,
  end_date = NULL
)
Arguments

- **db**: A database connection, as created by `create_db`.
- **sensor**: A character vector of one or multiple vectors. Use "All" for all sensors. See `sensors` for a list of all available sensors.
- **participant_id**: A character string identifying a single participant. Use `get_participants` to retrieve all participants from the database. Leave empty to get data for all participants.
- **start_date**: Optional search window specifying date where to begin search. Must be convertible to date using `as.Date`. Use `first_date` to find the date of the first entry for a participant.
- **end_date**: Optional search window specifying date where to end search. Must be convertible to date using `as.Date`. Use `last_date` to find the date of the last entry for a participant.

Value

A named vector containing the number of rows for each sensor.

---

**get_participants**  
*Get all participants*

Description

[Stable]

Usage

`get_participants(db, lazy = FALSE)`

Arguments

- **db**: A database connection, as created by `create_db`.
- **lazy**: Whether to evaluate lazily using `dbplyr`.

Value

A data frame containing all `participant_id` and `study_id`. 
get_processed_files  Get all processed files from a database

Description

[Stable]

Usage

get_processed_files(db)

Arguments

db  A database connection, as created by create_db.

Value

A data frame containing the file_name, participant_id, and study_id of the processed files.

generate_studies  Get all studies

Description

[Stable]

Usage

get_studies(db, lazy = FALSE)

Arguments

db  A database connection, as created by create_db.
lazy  Whether to evaluate lazily using dbplyr.

Value

A data frame containing all studies.
**haversine**

*Calculate the Great-Circle Distance between two points in kilometers*

**Description**

[Stable]

Calculate the great-circle distance between two points using the Haversine function.

**Usage**

```
haversine(lat1, lon1, lat2, lon2, r = 6371)
```

**Arguments**

- `lat1`: The latitude of point 1 in degrees.
- `lon1`: The longitude of point 1 in degrees.
- `lat2`: The latitude of point 2 in degrees.
- `lon2`: The longitude of point 2 in degrees.
- `r`: The average earth radius.

**Value**

A numeric value of the distance between point 1 and 2 in kilometers.

**Examples**

```r
fra <- c(50.03333, 8.570556) # Frankfurt Airport
ord <- c(41.97861, -87.90472) # Chicago O'Hare International Airport
haversine(fra[1], fra[2], ord[1], ord[2]) # 6971.059 km
```

**identify_gaps**

*Identify gaps in mpathsenser mobile sensing data*

**Description**

[Stable]

Oftentimes in mobile sensing, gaps appear in the data as a result of the participant accidentally closing the app or the operating system killing the app to save power. This can lead to issues later on during data analysis when it becomes unclear whether there are no measurements because no events occurred or because the app quit in that period. For example, if no screen on/off event occur in a 6-hour period, it can either mean the participant did not turn on their phone in that period or that the app simply quit and potential events were missed. In the latter case, the 6-hour missing period has to be compensated by either removing this interval altogether or by subtracting the gap from the interval itself (see examples).
**Usage**

```r
identify_gaps(
    db,
    participant_id = NULL,
    min_gap = 60,
    sensor = "Accelerometer"
)
```

**Arguments**

- `db`: A database connection to an m-Path Sense database.
- `participant_id`: A character string identifying a single participant. Use `get_participants` to retrieve all participants from the database. Leave empty to get data for all participants.
- `min_gap`: The minimum time (in seconds) passed between two subsequent measurements for it to be considered a gap.
- `sensor`: The name of a sensor. See `sensors` for a list of available sensors.

**Details**

While any sensor can be used for identifying gaps, it is best to choose a sensor with a very high, near-continuous sample rate such as the accelerometer or gyroscope. This function then creates time between two subsequent measurements and returns the period in which this time was larger than `min_gap`.

Note that the `from` and `to` columns in the output are character vectors in UTC time.

**Value**

A tibble containing the time period of the gaps. The structure of this tibble is as follows:

- `participant_id`: the participant_id of where the gap occurred
- `from`: the time of the last measurement before the gap
- `to`: the time of the first measurement after the gap
- `gap`: the time passed between from and to, in seconds

**Examples**

```r
# Not run:

## Find the gaps for a participant and convert to datetime

gaps <- identify_gaps(db, "12345", min_gap = 60) %>%
    mutate(across(c(to, from), ymd_hms)) %>%
    mutate(across(c(to, from), with_tz, "Europe/Brussels"))

# Get some sensor data and calculate a statistic, e.g. the time spent walking
# You can also do this with larger intervals, e.g. the time spent walking per hour

walking_time <- get_data(db, "Activity", "12345") %>%
    collect() %>%
    mutate(datetime = ymd_hms(paste(date, time))) %>%
    mutate(datetime = with_tz(datetime, "Europe/Brussels")) %>%
```
arrange(datetime) %>%
mutate(prev_time = lag(datetime)) %>%
mutate(duration = datetime - prev_time) %>%
filter(type == "WALKING")

# Find out if a gap occurs in the time intervals
walking_time %>%
  rowwise() %>%
  mutate(gap = any(gaps$from >= prev_time & gaps$to <= datetime))

## End(Not run)

---

**Description**

**[Stable]**

Import JSON files from m-Path Sense into a structured database. This function is the bread and butter of this package, as it creates (or rather fills) the database that (almost) all the other functions use.

**Usage**

```r
import(
  path = getwd(),
  db = NULL,
  dbname = "sense.db",
  overwrite_db = TRUE,
  sensors = NULL,
  batch_size = 24,
  backend = "RSQLite",
  recursive = TRUE,
  parallel = FALSE
)
```

**Arguments**

- **path**: The path to the file directory.
- **db**: Valid database connection.
- **dbname**: If no database is provided, a new database dbname is created.
- **overwrite_db**: If a database with the same dbname already exists, should it be overwritten?
- **sensors**: Select one or multiple sensors as in `sensors`. Leave NULL to extract all sensor data.
- **batch_size**: The number of files that are to be processed in a single batch.
**index_db**

Name of the database backend that is used. Currently, only RSQLite is supported.

**recursive**

Should the listing recurse into directories?

**parallel**

A value that indicates whether to do reading in and processing in parallel. If this argument is a number, this indicates the number of workers that will be used.

**Details**

Import is highly customisable in the sense that you can specify which sensors to import (even though there may be more in the files) and it also allows batching for a speedier writing process. If `parallel` is `TRUE`, it is recommended to `batch_size` be a scalar multiple of the number of CPUs the parallel cluster can use. If a single JSON file in the batch causes an error, the batch is terminated (but not the function) and it is up to the user to fix the file. This means that if `batch_size` is large, many files will not be processed. Set `batch_size` to 1 for sequential file processing (i.e. one-by-one).

Currently, only SQLite is supported as a backend. Due to its concurrency restriction, the `parallel` option is disabled. To get an indication of the progress so far, set one of the `handlers` using the `progressr` package, e.g. `progressr::handlers('progress')`.

**Value**

A message indicating how many files were imported. Imported database can be reopened using `open_db`.

**Progress**

You can be updated of the progress by this function by using the `progress` package. See `progressr`'s vignette on how to subscribe to these updates.

---

```r
index_db

Create indexes for a mpathsenser database
```

**Description**

[Stable]

**Usage**

```r
index_db(db)
```

**Arguments**

- `db` A database connection to an m-Path Sense database.

**Value**

No return value, called for side effects.
last_date

Extract the date of the last entry

Description

[Stable]

A helper function for extracting the last date of entry of (of one or all participant) of one sensor. Note that this function is specific to the last date of a sensor. After all, it wouldn’t make sense to extract the last date for a participant of the device info, while the last accelerometer measurement occurred a day later.

Usage

last_date(db, sensor, participant_id = NULL)

Arguments

db A database connection to an m-Path Sense database.
sensor The name of a sensor. See sensors for a list of available sensors.
participant_id A character string identifying a single participant. Use get_participants to retrieve all participants from the database. Leave empty to get data for all participants.

Value

A string in the format 'YYYY-mm-dd' of the last entry date.

Examples

## Not run:
db <- open_db()
first_date(db, "Accelerometer", "12345")
## End(Not run)

link

Link y to the time scale of x

Description

[Stable]

One of the key tasks in analysing mobile sensing data is being able to link it to other data. For example, when analysing physical activity data, it could be of interest to know how much time a participant spent exercising before or after an ESM beep to evaluate their stress level. link allows you to map two data frames to each other that are on different time scales, based on a pre-specified offset before and/or after. This function assumes that both x and y have a column called time containing DateTimeClasses.
Usage

```r
link(
  x,
  y,
  by = NULL,
  offset_before = 0,
  offset_after = 0,
  add_before = FALSE,
  add_after = FALSE,
  split = by
)
```

Arguments

- **x, y**: A pair of data frames or data frame extensions (e.g. a tibble). Both x and y must have a column called time.

- **by**: A character vector indicating the variable(s) to match by, typically the participant IDs. If NULL, the default, `*_join()` will perform a natural join, using all variables in common across x and y. Therefore, all data will be mapped to each other based on the time stamps of x and y. A message lists the variables so that you can check they’re correct; suppress the message by supplying by explicitly.

  To join by different variables on x and y, use a named vector. For example, `by = c('Var a', 'Var b')` will match x$a to y$b.

  To join by multiple variables, use a vector with length > 1. For example, `by = c('a', 'b')` will match x$a to y$a and x$b to y$b. Use a named vector to match different variables in x and y. For example, `by = c('a' = 'b', 'c' = 'd')` will match x$a to y$b and x$c to y$d.

- **offset_before**: The time before each measurement in x that denotes the period in which y is matched. Must be convertible to a period by `as.period`.

- **offset_after**: The time after each measurement in x that denotes the period in which y is matched. Must be convertible to a period by `as.period`.

- **add_before**: Logical value. Do you want to add the last measurement before the start of each interval?

- **add_after**: Logical value. Do you want to add the first measurement after the end of each interval?

- **split**: An optional grouping variable to split the computation by. When working with large data sets, the computation can grow so large it no longer fits in your computer’s working memory (after which it will probably fall back on the swap file, which is very slow). Splitting the computation trades some computational efficiency for a large decrease in RAM usage. This argument defaults to by to automatically suppress some of its RAM usage.

Details

y is matched to the time scale of x by means of time windows. These time windows are defined as the period between `x - offset_before` and `x + offset_after`. Note that either `offset_before`
or offset_after can be 0, but not both. The "interval" of the measurements is therefore the associated time window for each measurement of x and the data of y that also falls within this period. For example, an offset_before of minutes(30) means to match all data of y that occurred before each measurement in x. An offset_after of 900 (i.e. 15 minutes) means to match all data of y that occurred after each measurement in x. When both offset_before and offset_after are specified, it means all data of y is matched in an interval of 30 minutes before and 15 minutes after each measurement of x, thus combining the two arguments.

The arguments add_before and add_after let you decide whether you want to add the last measurement before the interval and/or the first measurement after the interval respectively. This could be useful when you want to know which type of event occurred right before or after the interval of the measurement. For example, at offset_before = "30 minutes", the data may indicate that a participant was running 20 minutes before a measurement in x. However, with just that information there is no way of knowing what the participant was doing the first 10 minutes of the interval. The same principle applies to after the interval. When add_before is set to TRUE, the last measurement of y occurring after the interval of x is added to the output data as the first row, having the time of x - offset_before (i.e. the start of the interval). When add_after is set to TRUE, the first measurement of y occurring after the interval of x is added to the output data as the last row, having the time of x + offset_after (i.e. the end of the interval). This way, it is easier to calculate the difference to other measurements of y later (within the same interval). Additionally, an extra column (original_time) is added in the nested data column, which is the original time of the y measurement and NULL for every other observation. This may be useful to check if the added measurement isn’t too distant (in time) from the others. Note that multiple rows may be added if there were multiple measurements in y at exactly the same time.

Value

A tibble with the data of x with a new column data with the matched data of y according to offset_before and offset_after.

Warning

Note that setting add_before and add_after each add one row to each nested tibble of the data column. Thus, if you are only interested in the total count (e.g. the number of total screen changes), remember to set these arguments to FALSE or make sure to filter out rows that do note have an original_time. Simply subtracting 1 or 2 does not work as not all measurements in x may have a measurement in y before or after (and thus no row is added).

| link_db | Link two sensors OR one sensor and an external data frame using an mpathsenser database |

Description

[Stable] This function is specific to mpathsenser databases. It is a wrapper around link but extracts data in the database for you.


Usage

\[
\text{link\_db}( \\
\text{db}, \\
\text{sensor\_one}, \\
\text{sensor\_two} = \text{NULL}, \\
\text{external} = \text{NULL}, \\
\text{offset\_before} = 0, \\
\text{offset\_after} = 0, \\
\text{add\_before} = \text{FALSE}, \\
\text{add\_after} = \text{FALSE}, \\
\text{participant\_id} = \text{NULL}, \\
\text{start\_date} = \text{NULL}, \\
\text{end\_date} = \text{NULL}, \\
\text{reverse} = \text{FALSE}, \\
\text{ignore\_large} = \text{FALSE} \\
) 
\]

Arguments

db A database connection to an m-Path Sense database.
sensor\_one The name of a primary sensor. See sensors for a list of available sensors.
sensor\_two The name of a secondary sensor. See sensors for a list of available sensors. Cannot be used together with external.
external Optionally, specify an external data frame. Cannot be used at the same time as a second sensor. This data frame must have a column called time.
offset\_before The time before each measurement in x that denotes the period in which y is matched. Must be convertible to a period by as\_period.
offset\_after The time after each measurement in x that denotes the period in which y is matched. Must be convertible to a period by as\_period.
add\_before Logical value. Do you want to add the last measurement before the start of each interval?
add\_after Logical value. Do you want to add the first measurement after the end of each interval?
participant\_id A character string identifying a single participant. Use get\_participants to retrieve all participants from the database. Leave empty to get data for all participants.
start\_date Optional search window specifying date where to begin search. Must be convertible to date using as\_Date. Use first\_date to find the date of the first entry for a participant.
end\_date Optional search window specifying date where to end search. Must be convertible to date using as\_Date. Use last\_date to find the date of the last entry for a participant.
reverse Switch sensor\_one with either sensor\_two or external? Particularly useful in combination with external.
ignore\_large Safety override to prevent long wait times. Set to TRUE to do this function on lots of data.
Value

A tibble with the data of sensor_one with a new column data with the matched data of either sensor_two or external according to offset. The other way around when reverse = TRUE.

See Also

link

link_gaps

Description

[Experimental]

Gaps in mobile sensing data typically occur when the app is stopped by the operating system or the user. While small gaps may not pose problems with analyses, greater gaps may cause bias or skew your data. As a result, gap data should be considered in order to inspect and limit their influence. This function, like link, allows you to connect gaps to other data (usually ESM/EMA data) within a user-specified time range.

Usage

link_gaps(
  data,
  gaps,
  by = NULL,
  offset_before = 0,
  offset_after = 0,
  raw_data = FALSE
)

Arguments

data A data frame or an extension to a data frame (e.g. a tibble). While gap data can be linked to any other type of data, ESM data is most commonly used.
gaps A data frame (extension) containing the gap data. See identify_gaps for retrieving gap data from an mpathsenser database. It should at least contain the columns from and to (both in a date-time format), as well as any specified columns in by.
by A character vector indicating the variable(s) to match by, typically the participant IDs. If NULL, the default, _join() will perform a natural join, using all variables in common across x and y. Therefore, all data will be mapped to each other based on the time stamps of x and y. A message lists the variables so that you can check they’re correct; suppress the message by supplying by explicitly. To join by different variables on x and y, use a named vector. For example, by = c('a' = 'b') will match x$a to y$b.
To join by multiple variables, use a vector with length > 1. For example, by = c('a', 'b') will match x$a to y$a and x$b to y$b. Use a named vector to match different variables in x and y. For example, by = c('a' = 'b', 'c' = 'd') will match x$a to y$b and x$c to y$d.

**offset_before**
The time before each measurement in x that denotes the period in which y is matched. Must be convertible to a period by \texttt{as.period}.

**offset_after**
The time after each measurement in x that denotes the period in which y is matched. Must be convertible to a period by \texttt{as.period}.

**raw_data**
Whether to include the raw data (i.e. the matched gap data) to the output as \texttt{gap_data}.

**Value**
The original data with an extra column \texttt{duration} indicating the gap during within the interval in seconds (if \texttt{duration} is \texttt{TRUE}), or an extra column called \texttt{gap_data} containing the gaps within the interval. The function ensures all durations and gap time stamps are within the range of the interval.

**See Also**
\texttt{bin_data} for linking two sets of intervals to each other; \texttt{identify_gaps} for finding gaps in the sampling; \texttt{add_gaps} for adding gaps to data;
participant_id  A character string identifying a single participant. Use get_participants to retrieve all participants from the database.

...  Unquoted names of columns of the sensor table to average over.

n  The number of observations to average over.

start_date  Optional search window specifying date where to begin search. Must be convertible to date using as.Date. Use first_date to find the date of the first entry for a participant.

eend_date  Optional search window specifying date where to end search. Must be convertible to date using as.Date. Use last_date to find the date of the last entry for a participant.

Value

A tibble with the same columns as the input, modified to be a moving average.

Examples

## Not run:
get_moving_average(db, "Light", "12345", mean_lux, max_lux, n = 5)

## End(Not run)

---

n_screen_on  Get number of times screen turned on

Description

[Experimental]

Usage

n_screen_on(
  db,
  participant_id,
  start_date = NULL,
  end_date = NULL,
  by = c("Total", "Hour", "Day")
)

Arguments

db  A database connection to an m-Path Sense database.

participant_id  A character string identifying a single participant. Use get_participants to retrieve all participants from the database. Leave empty to get data for all participants.
### n_screen_unlocks

Optional search window specifying date where to begin search. Must be convertible to date using `as.Date`. Use `first_date` to find the date of the first entry for a participant.

### end_date

Optional search window specifying date where to end search. Must be convertible to date using `as.Date`. Use `last_date` to find the date of the last entry for a participant.

### by

Either 'Total', 'Hour', or 'Day' indicating how to summarise the results. Defaults to total.

---

#### Value

In case grouping is by the total amount, returns a single numeric value. For date and hour grouping returns a tibble with columns 'date' or 'hour' and the number of screen on's 'n'.

---

#### Description

*Experimental*

#### Usage

```r
n_screen_unlocks(
    db,
    participant_id,
    start_date = NULL,
    end_date = NULL,
    by = c("Total", "Hour", "Day")
)
```

#### Arguments

- **db**
  - A database connection to an m-Path Sense database.
- **participant_id**
  - A character string identifying a single participant. Use `get_participants` to retrieve all participants from the database. Leave empty to get data for all participants.
- **start_date**
  - Optional search window specifying date where to begin search. Must be convertible to date using `as.Date`. Use `first_date` to find the date of the first entry for a participant.
- **end_date**
  - Optional search window specifying date where to end search. Must be convertible to date using `as.Date`. Use `last_date` to find the date of the last entry for a participant.
- **by**
  - Either 'Total', 'Hour', or 'Day' indicating how to summarise the results. Defaults to total.
**screen_duration**

Description

[Experimental]

Calculate the screen duration time where the screen was *unlocked* (i.e. not just on).

Usage

```r
screen_duration(
  db,
  participant_id,
  start_date = NULL,
  end_date = NULL,
  by = c("Hour", "Day")
)
```

**Value**

In case grouping is by the total amount, returns a single numeric value. For date and hour grouping returns a tibble with columns 'date' or 'hour' and the number of screen unlocks 'n'.

---

**open_db**

*Open an mpathsenser database.*

Description

[Stable]

Usage

```r
open_db(path = getwd(), db_name = "sense.db")
```

Arguments

- `path`: The path to the database. Use NULL to use the full path name in `db_name`.
- `db_name`: The name of the database.

Value

A connection to an mpathsenser database.

See Also

- `close_db` for closing a database; `copy_db` for copying (part of) a database; `index_db` for indexing a database; `get_data` for extracting data from a database.
Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>db</td>
<td>A database connection to an m-Path Sense database.</td>
</tr>
<tr>
<td>participant_id</td>
<td>A character string identifying a single participant. Use <code>get_participants</code> to retrieve all participants from the database. Leave empty to get data for all participants.</td>
</tr>
<tr>
<td>start_date</td>
<td>Optional search window specifying date where to begin search. Must be convertible to date using <code>as.Date</code>. Use <code>first_date</code> to find the date of the first entry for a participant.</td>
</tr>
<tr>
<td>end_date</td>
<td>Optional search window specifying date where to end search. Must be convertible to date using <code>as.Date</code>. Use <code>last_date</code> to find the date of the last entry for a participant.</td>
</tr>
<tr>
<td>by</td>
<td>Either 'Hour' or 'Day' indicating how to summarise the results. Leave empty to get raw screen duration per measurement.</td>
</tr>
</tbody>
</table>

Value

A tibble with either 'hour' and 'duration' columns or 'date' and 'duration' columns depending on the `by` argument. Alternatively, if no `by` is specified, a remote tibble is returned with the date, time, and duration since the previous measurement.

---

Description

[Stable]

A list containing all available sensors in this package you can work with. This variable was created so it is easier to use in your own functions, e.g. to loop over sensors.

Usage

```r
sensors
```

Format

An object of class character of length 25.

Value

A character vector containing all sensor names supported by `mpathsenser`.

Examples

```r
sensors
```
step_count  

Get step count

Description

[Experimental]

Extracts the number of steps per hour as sensed by the underlying operating system.

Usage

step_count(db, participant_id = NULL, start_date = NULL, end_date = NULL)

Arguments

- **db**: A database connection to an m-Path Sense database.
- **participant_id**: A character string identifying a single participant. Use `get_participants` to retrieve all participants from the database. Leave empty to get data for all participants.
- **start_date**: Optional search window specifying date where to begin search. Must be convertible to date using `as.Date`. Use `first_date` to find the date of the first entry for a participant.
- **end_date**: Optional search window specifying date where to end search. Must be convertible to date using `as.Date`. Use `last_date` to find the date of the last entry for a participant.

Value

A tibble with the 'date', 'hour', and the number of 'steps'.

test_jsons  

Test JSON files for being in the correct format.

Description

[Stable]

Usage

```r
test_jsons(
    path = getwd(),
    files = NULL,
    db = NULL,
    recursive = TRUE,
    parallel = FALSE
  )
```
unzip_data

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>The path name of the JSON files.</td>
</tr>
<tr>
<td>files</td>
<td>Alternatively, a character list of the input files.</td>
</tr>
<tr>
<td>db</td>
<td>A mpathsenser database connection (optional). If provided, will be used to check which files are already in the database and check only those JSON files which are not.</td>
</tr>
<tr>
<td>recursive</td>
<td>Should the listing recurse into directories?</td>
</tr>
<tr>
<td>parallel</td>
<td>A logical value whether you want to check in parallel. Useful when there are a lot of files. If you have already used plan, you can leave this parameter to FALSE.</td>
</tr>
</tbody>
</table>

Value

A message indicating whether there were any issues and a character vector of the file names that need to be fixed. If there were no issues, no result is returned.

Progress

You can be updated of the progress by this function by using the progress package. See progressr’s vignette on how to subscribe to these updates.

Description

[Stable]

Similar to unzip, but makes it easier to unzip all files in a given path with one function call.

Usage

```r
unzip_data(
  path = getwd(),
  to = NULL,
  overwrite = FALSE,
  recursive = TRUE,
  parallel = FALSE
)
```

Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>path</td>
<td>The path to the directory containing the zip files.</td>
</tr>
<tr>
<td>to</td>
<td>The output path.</td>
</tr>
<tr>
<td>overwrite</td>
<td>Logical value whether you want to overwrite already existing zip files.</td>
</tr>
</tbody>
</table>
**unzip_data**

**recursive**
Logical value indicating whether to unzip files in subdirectories as well. These files will then be unzipped in their respective subdirectory.

**parallel**
A logical value whether you want to check in parallel. Useful when there are a lot of files. If you have already used `future::plan`, you can leave this parameter to `FALSE`.

**Value**
A message indicating how many files were unzipped.

**Progress**
You can be updated of the progress by this function by using the `progress` package. See `progressr`'s vignette on how to subscribe to these updates.
Index

* datasets
  freq, 15
  sensors, 35
  activity_duration, 3
  add_gaps, 4, 31
  app_category, 5
  as.Date, 3, 11, 17, 18, 20, 29, 32, 33, 35, 36
  as.period, 27, 29, 31
  bin_data, 6, 31
  ccopy, 8
  close_db, 9, 34
  copy_db, 10, 34
  coverage, 10, 15
  create_db, 10, 12, 20, 21
  DateTimeClasses, 26
  dbDisconnect, 9
  dbplyr, 20, 21
  decrypt_gps, 13
  device_info, 13
  first_date, 3, 14, 17, 18, 20, 29, 32, 33, 35, 36
  fix_jsons, 14
  freq, 15
  geocode_rev, 16
  get_app_usage, 17, 18
  get_data, 3, 4, 18, 34
  get_installed_apps, 19
  get_nrows, 19
  get_participants, 3, 13, 14, 17–20, 20, 23, 26, 29, 32, 33, 35, 36
  get_processed_files, 21
  get_studies, 21
  handlers, 25
  haversine, 22
  identify_gaps, 4, 5, 22, 30, 31
  import, 24
  index_db, 25, 34
  last_date, 3, 17, 18, 20, 26, 29, 32, 33, 35, 36
  link, 26, 28, 30
  link_db, 28
  link_gaps, 5, 7, 30
  minutes, 28
  moving_average, 31
  n_screen_on, 32
  n_screen_unlocks, 33
  open_db, 9, 11, 25, 34
  plan, 37
  progress, 15, 25, 37, 38
  screen_duration, 18, 34
  sensors, 10, 11, 14, 18, 20, 23, 24, 26, 29, 31, 35
  step_count, 36
  tbl, 18
  test_jsons, 36
  unzip, 37
  unzip_data, 37